An Event-B Development Process for the Distributed BIP Framework

Badr Siala, Tahar Bhiri, Jean-Paul Bodeveix, M Filali

An Event-B Development Process for the Distributed BIP Framework

Badr Siala[1,2], Mohamed Tahar Bhiri[1], Jean-Paul Bodeveix[2], and Mamoun Filali[2(✉)]

[1] Université de Sfax, Sfax, Tunisia
siala@irit.fr, tahar_bhiri@yahoo.fr
[2] IRIT CNRS UPS Université de Toulouse, Toulouse, France
{bodeveix,filali}@irit.fr

Abstract. We present a refinement-based methodology to design correct by construction distributed systems specified as Event-B models. Starting from an Event-B machine, the studied process proposes successive steps in order to split and schedule the computation of complex events and then to map them on subcomponents. The specification of these steps is done through two domain specific languages. From these specifications, two refinements are generated. Eventually, a distributed code architecture is also generated. The correctness of the process relies on the correctness of the refinements and the translation. We target the distributed BIP framework.

1 Introduction

In this paper, we are concerned with providing tool support to assist system design using a safe refinement-based process. The considered systems will be seen as a collection of interacting actors. The first levels of the process provides a centralized view of the system behavior. It will be built by taking into account system requirements incrementally, in the form of a series of abstract machines written in Event-B [3]. Then, we propose dedicated, user guided, refinement generators to take into account the distributed nature of the designed system. As a result, we obtain a set of interacting machines of which composition is proven to conform to the abstract levels. The system can then be executed on a distributed platform via a translation to the BIP (Behavior, Interaction, Priority) language [5]. By now, it should be clear that our aim is not to fully automate the distribution process but to assist it. While keeping modest, the difference is similar to that between a model checker where the proof of a judgement is automatic and a theorem proving assistant where the user ha s to compose basic strategies in order to make his proof. Actually, while a theorem proving assistant helps to construct the proof of a goal, we intend to help in the elaboration of a distributed model through refinement patterns [16].

The semantics of Event-B and BIP are based on labeled transition systems thereby promoting their coupling. Event-B is used for the formal specification and the decomposition of initially centralized reactive systems. BIP is used for

the implementation and the deployment of distributed systems specified and verified in Event-B. The skeleton of the BIP code is automatically generated from Event-B.

Sections 2 and 3 present Event-B composition/decomposition techniques and the component-based model BIP. Section 4 proposes our development process of distributed systems by coupling Event-B and BIP. This process is illustrated by Fig. 1. Section 5 relates our distributed systems development approach to existing work. We conclude the paper in Sect. 6 and present some perspectives.

Fig. 1. Process steps

2 Event-B

The Event-B method allows the development of correct by construction systems and software [3]. To achieve this, it supports natively a formal development process based on a refinement mechanism with mathematical proofs. Figure 2 illustrates a refinement step where a machine M0 using a context C0 is refined by a machine M1 using an extension C1 of C0. Contexts define abstract data types through sets, constants and axioms while machines define symbolic labelled transition systems through variables and events specifying their evolution while preserving invariant properties.

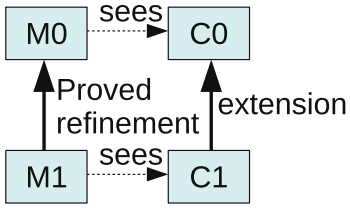

Fig. 2. Event-B development step

As a running example, we will consider the electronic hotel key system case study [15][1]. The context (Listing 1.1) introduces basic data structures: guests, rooms and cards defined as ordered pairs of keys[2]. State variables (Listing 1.2) declare the current key of a room (`currk`), the rooms owned by a guest (`owns`), the cards issued by the hotel and cards owned by a guest.

```
context chotel
sets
  ROOM GUEST KEY
constants
  CARD
axioms
  @crd CARD = KEY × KEY
end
```

Listing 1.1. Hotel context

```
machine hotel sees chotel
variables
   currk owns issued cards
invariants
  @currk_ty currk ∈ ROOM → KEY
  @owns_ty owns ∈ ROOM → ℙ(GUEST)
  @issued_ty issued ∈ ℙ(CARD)
  @cards_ty cards ∈ GUEST → ℙ(CARD)
```

Listing 1.2. Hotel state variables

[1] The full code is available in https://dl.dropboxusercontent.com/u/98832434/hotelrefinements.html.

[2] prj1 returns the left projection of an ordered pair.

The dynamics of the system is described by events, one of which, named `register` being given in Listing 1.3.

This is a non-deterministic event, parametrized by the variables g for the incoming guest, r for the room to be chosen and c for the card to be issued. The `where` part specifies which of these triples are allowed: the room should be free (g1), the card should not have been issued (g2) and the card should open the door (g3). The `then` part specifies how the state space is updated: the current key of the room will be the second key of the card (a1), the card has been issued (a2), is owned by the guest (a3) which owns the room (a4).

```
event register
any g r c
where
  @tg g ∈ GUEST
  @tr r ∈ ROOM
  @tc c ∈ CARD
  @g1 owns(r) = ∅
  @g2 c ∉ issued
  @g3 prj1(c) = currk(r)
then
  @a1 currk(r) := prj2(c)
  @a2 issued := issued∪{c}
  @a3 cards(g) := cards(g)∪{c}
  @a4 owns(r) := {g}
end
```

Listing 1.3. Hotel register event

Recently, Event-B has been enhanced by reuse techniques such as genericity [17], abstraction [13], composition and decomposition [4,18]. In this paper, we are mainly concerned by composition and decomposition. They allow the formal combination of specifications through the refinement mechanism. Two methods of composition/decomposition were identified for Event-B: shared variable [19] and shared event [18]. Shared variable composition/decomposition is suitable for shared-memory parallel systems whereas shared event composition/decomposition is suitable for message-passing distributed systems. In this paper, we limit ourselves to the shared event composition/decomposition approach inspired by CSP where processes synchronize on the same event and may exchange messages. In Event-B, subcomponents (sub-specifications) can synchronize through shared events and exchange data specified by the common value of their parameters.

2.1 Shared Event Composition

The shared event composition of Event-B machines is represented by a new construct called **composed machine** [18]. This operation requires the disjointness of the sets of state variables of the machines to be composed. It is defined as a machine merging subcomponents' properties: conjunction of invariants, union of variables and parallel synchronisation of events. The composition of two events which have common parameters p is defined as follows [18]:

```
E1 ≙ any p,x where G(p,x,m1) then S(p,x,m1) end
E2 ≙ any p,y where H(p,y,m2) then T(p,y,m2) end
E1||E2 ≙ any p,x,y where G(p,x,m1)∧H(p,y,m2) then S(p,x,m1)||T(p,y,m2) end
```

where x, y, p are sets of parameters from the events `E1` and `E2` and m_1 and m_2 are the variables of the two subcomponents. Sending a value v can be modeled by using a guard of the form $p = v$. The other guards will constrain the sent value either at the sending point or at the receiving point. This design pattern originating from CSP has been proposed by Butler for action systems [7] and in [18] for Event-B.

The **composed machine** is supposed to satisfy the Event-B standard Proof Obligations (POs) related to invariants and refinements. Moreover, during the

composition of several subcomponents, it is possible to add a composition invariant relating the states of subcomponents.

Like CSP parallel composition, Event-B shared event composition is monotonic under refinement [18]. Actually, the composition of refined subcomponents is a refinement of the composition of initial subcomponents.

Semantics. In the following we state the semantics of the product CM of machines M_i as a labelled transition over the variables of the subcomponents.

$$\frac{(e = \|_{i\in I}\, M_i.e) \in CM \quad (e = \textbf{ any } X_i \textbf{ where } G_i(X_i) \textbf{ then } S_i(X_i)) \in M_i \quad \bigwedge_{i\in I} G_i(X_i \lhd p)(v_i),\ \bigwedge_{i\in I} v_i' = S_i(X_i \lhd p)(v_i),\ \bigwedge_{i\notin I} v_i' = v_i}{\langle v_1, \ldots, v_n\rangle \xrightarrow{e(p)} \langle v_1', \ldots, v_n'\rangle}$$

where

- v_i is the valuation of the variables of the component M_i,
- p is the valuation of the union of the parameters of the component events.

2.2 Shared Event Decomposition

Decomposition is a mean to master the complexity (divide and conquer) or to introduce architectural aspects (see Sect. 4). It can be seen as the inverse of composition where an Event-B model is split into several simpler subcomponents. Concretely, decomposition is specified by a set of subcomponent names and a partition of variables, each class being mapped to a subcomponent. An important point is that the composition of subcomponents refines the initial centralized model. However, decomposition fails if a guard or an action refers to variables mapped to different locations. Within the scope of distributed systems, we propose a support to help solving these problems. Decomposition can also fail if the synthesized typing invariant is not strong enough. It could to badly formed expression where some partial functions are applied outside their definition domain. We do not consider this problem.

2.3 Shared Event Composition/Decomposition Tool

The Rodin platform provides an interactive tool [19] as a plugin allowing the shared event composition/decomposition of Event-B specifications. Composition is defined by editing a *composed machine* which designates the subcomponents and defines synchronization events as a product of subcomponent events. Conversely, decomposition is built by naming subcomponents and mapping variables on them. In case of success, the tool generates a machine for each subcomponent and a composed machine. Given that the decomposition of the invariants depends on the scope of the variables, invariants containing variables distributed over several subcomponents are discarded.

3 The BIP Component-Based Model

The BIP language [5] allows to build component-based systems. To achieve this, it offers a means to describe atomic components and composition operators describing composite components. In BIP, an architecture is a hierarchical model consisting of a structured collection of components obtained by composition of atomic components which represent the leaves of the hierarchical model.

3.1 Atomic Components

An atomic BIP component declares data, ports and a behavior. Data variables (**data**) are typed. Ports (**port**) give access to some variables and constitute the component **interface**. The behavior is defined by a port, a guard and a variable update function.

According to the component-based paradigm, a BIP component is a design-time concept (a type) and a runtime concept (an instance). This is also true for ports. Listings 1.4 and 1.5 present, respectively, the port types and an atomic component `ty_Desk` produced by our BIP code generator (see Sect. 4.3).

```
port type ty_empty_port()
port type ty_register_Desk (INT register_g, INT  register_c)
port type ty_register_Guest (INT register_g, INT register_c)
```

Listing 1.4. Port types

```
atom type ty_Desk()
  /* state variables */
  data INT currk ...
  /* temporary variables */
  data INT register_g
  /* port instances */
  export port ty_empty_port compute_register_r ()
  export port ty_register_Desk register(register_g ,register_c)
  place P0
  initial to P0 do /* initialize variables */
  /* transitions */
  on compute_register_c from P0 to P0 provided register_g_computed
  on register from P0 to P0 provided register_g_computed do /* action */
end
```

Listing 1.5. Atomic component ty_Desk

3.2 Coordination Between BIP Components

The component-based model BIP has three layers called Behavior, Interaction and Priority. The *Behavior* layer describes the behavior of atomic components (see Sect. 3.1) whilst layers *Interaction* and *Priority* describe the architectural aspects of a component-based system. This separation between behavioral and architectural aspects is an asset in BIP [5]. The synchronization constraints between BIP components are expressed through interactions defined by the **connector** construct whereas scheduling constraints between these interactions are expressed through the *Priority* concept.

BIP Connectors. A connector is simultaneously a *design-time* and a *runtime* concept. A BIP connector is defined by:

- a set of ports $\{p_1, ..., p_n\}$ of subcomponents involved in an interaction.
- an optional port p with variables exported by the connector allowing to compose the connectors.
- a set of interactions which are subsets of $\{p_1, ..., p_n\}$. Every interaction can be annotated by a guard, an *upstream transfer functions* (`up`) and *downstream transfer functions* (`down`). The guards of the interactions involve variables in the scope of ports and connector variables. In this work, we limit ourselves to simple connectors restricted to data transfer (Sect. 4).

For example, Listing 1.6 defines two connector types[3]. The first one denotes a pure synchronization and the second one a synchronization with data exchange.

```
connector type ty_compute_register_r(ty_empty_port Desk,ty_empty_port Guest)
  define Desk Guest
  on Desk Guest down {}
end
connector type ty_register(ty_register_Desk Desk,ty_register_Guest Guest)
  define Desk Guest
  on Desk Guest down {
    Guest.register_c=Desk.register_c;Desk.register_g=Guest.register_g;
  }
end
```

Listing 1.6. Connector types

Composite Component. In BIP, a composite component is both present at design-time and runtime. It includes the following elements:

- atomic or composite components declared by the keyword **component**;
- connectors which connect the components forming the composite component declared by the keyword **connector**;
- priority rules declared by the keyword **priority**;
- exported ports that define the interface of the composite component.

Listing 1.7 presents a composite component. It contains two atomic components and a connector for coordinating them.

```
compound type ty_hotel_decomposition ()
  component ty_Desk Desk()
  component ty_Guest Guest()
  connector ty_register register(Desk.register, Guest.register)
  ...
end
```

Listing 1.7. The Hotel root component type

[3] produced by our BIP code generator in Sect. 4.3.

3.3 BIP Execution and Operational Semantics

The BIP execution engine starts with the calculation of executable interactions (Interaction layer). Then, it schedules these interactions, taking into account the priority constraints (Priority layer). Finally, the transitions of the atomic components involved in the interaction are executed (Behavior layer). We now give the operational semantics of the composition of a set of components $(C_i)_{i\in 1..n}$ connected through a set of connectors γ. First, we sum up the syntax of components and connectors as follows:

- $C_i = \langle \Sigma_i, P_i, X_i, \rightarrow_i \rangle$ where Σ_i are the locations of C_i, P_i its set of ports, X_i its set of variables, $\mathcal{G}(X_i)$ is a set predicates over X_i, $\mathcal{A}(X_i)$ is a set of actions over X_i and $\rightarrow_i \subseteq \Sigma_i \times P_i \times \mathcal{G}(X_i) \times \mathcal{A}(X_i) \times \Sigma_i$ its transitions labelled by a guard and an action. We will write $\sigma_i \xrightarrow{p_i/g_i/a_i}_i \sigma_i'$ for an element of $\rightarrow_i$.
- $\gamma \subseteq \{\langle I \subseteq 1..n, (p_i(x_i))_{i\in I} \in \Pi_{i\in I} P_i(X_i), p, G, (D_i)_{i\in I}, U\rangle\}$ is a set of connectors where for a given connector, I is the set indexes of interacting components, $(p_i(x_i))_{i\in I}$ the selected set of ports (one in each component) with their view x_i on component variables, p the outbound port, G the connector guard, D_i the set of *down* functions specifying the update of subcomponent states and U the *up* function specifying the outbound port data.

Then, the operational semantics of the composition is defined by the following transitions over locations and valuations v_i of the component variables. A connector over enabled ports is selected. The *down* actions D_i of the connector are performed before the local action a_i of each component.

$$\frac{\begin{array}{c}\langle I, (p_i)_{i\in I}, p, G, (D_i)_{i\in I}, U\rangle \in \gamma \\ \bigwedge_{i\in I} \sigma_i \xrightarrow{p_i/g_i/a_i}_i \sigma_i' \ \wedge \ \bigwedge_{i\notin I} \sigma_i' = \sigma_i \\ (\bigwedge_{i\in I} g_i(v_i)) \ \wedge \ G(\langle x_i \triangleleft v_i \mid i \in I\rangle) \\ \bigwedge_{i\in I} v_i' = a_i(v_i \ \lessdot\!\!+ \ D_i(\langle x_j \triangleleft v_j \mid j \in I\rangle)) \ \wedge \ \bigwedge_{i\notin I} v_i' = v_i\end{array}}{\langle (\sigma_1, v_1), \ldots, (\sigma_n, v_n)\rangle \xrightarrow{p(U(\langle x_i \triangleleft v_i \mid i\in I\rangle))} \langle (\sigma_1', v_1'), \ldots, (\sigma_n', v_n')\rangle}$$

For readability reasons, priorities are not taken into account. We should add that the fired interaction is not hidden by ready interactions having a lower priority.

3.4 The BIP Tool-Chain

The BIP tool-chain includes translators from other languages to BIP, formal verification tools and code generators from a BIP model. The BIP language features a static checker called D-Finder [5]. It is a compositional verification tool (invariants, deadlock). Likewise, the BIP language has a runtime verification tool [11]. The code generators take the BIP model and generate single-threaded or multi-threaded code that can be executed and analyzed [14].

4 Towards a Distribution Process

Our goal is to provide a process for guiding the user refinements in order to map an initial "centralized" design (as explained in Sect. 2) on a distributed architecture. The proposed process can be seen as a continuation of the basic methodology which captures requirements as successive refinements of an initial specification. However, as we target a system engineering process, our aim is not to propose a fully automatic distribution tool. For example, in the hotel case study, the behavior of the guest should be mapped on a `Guest` component. Figure 1 illustrates the proposed process. It is based on three steps: a splitting step which splits events in order to allow the incremental and local resolution of non-determinism, a mapping step which introduces *components* and *mappings* of variables over these components and a distributed code generation step.

We reuse the *shared event decomposition* plugin [18]. However, it does not apply on models where guards or actions access variables mapped on different components as the tool would not know how to split them. Moreover, even if each guard or action refers to only one variable, the resulting components produced by this tool would not be usable. Consider two variables a and b mapped on components C_1 and C_2 and the event `ev`:

`ev` $\triangleq$ `any p where @g1: a > p @g2: p < b then p1 := p end`

Applying [18] is possible: each of C_1 and C_2 gets a copy of ev with respectively g_1 and g_2 as their unique guard, but this leads to another problem: we get two synchronized events specifying constraints over the parameter p. Their separate refinement could lead to incompatible choices and thus to a deadlock resulting from the assembly. The proposed transformations allow the user to avoid this problem by guiding the refinement process. For this purpose, the user can provide parameters to automatic refinement tools. As a result, the two constraints will be located on the same component, while variables will be possibly mapped to distinct subcomponents. Transformations are organized in three steps presented in Sects. 4.1, 4.2 and 4.3.

Moreover, as an implementation constraint, we consider that BIP connectors should not perform computations. Data usage in connectors will thus be restricted to data transfer. This property will lead to a specific refinement of the Event-B model during the mapping processing step (see Sect. 4.2). These steps can be automatically performed given some user annotations. In order to support such a process, we consider two domain specific languages (DSL), one for specifying event parameters computation order and the other for specifying the mapping of machine variables and possibly the location of guard computations. The transformation steps are explicitly specified through the proposed DSLs. These two specifications are used to generate refined models and projections to subcomponents automatically. The correctness of the refinements ensures the correctness of the development. Our process, applied to our example, is illustrated by Fig. 3.

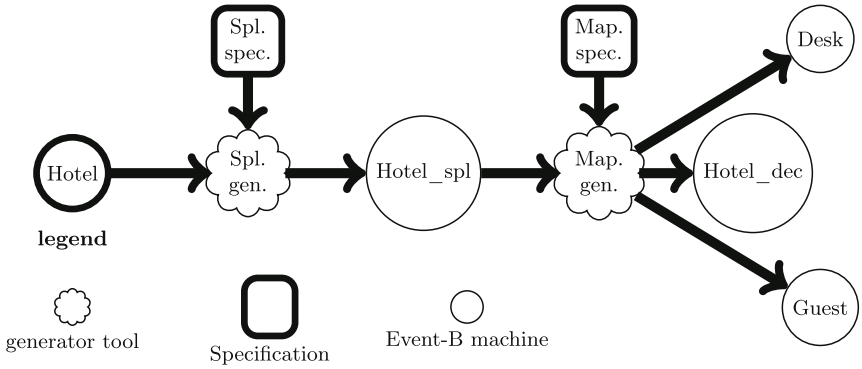

Fig. 3. Hotel transformations

4.1 The Event Splitting Step

The splitting step allows the user to inject heuristics for computing event parameters specified by a set of constraints: an event can be split in order to allow the incremental resolution of its non-determinism. This transformation can be useful if the event is non-deterministic and intended to be shared by several sub-components. Non-determinism will be constrained to occur on local events so that data exchanged will be locally computed before. This step is guided by the user as he may want to control the order in which non-determinism is resolved[4].

The Event Splitting Plugin. Figure 4 illustrates the profile of the transformation implemented as a Rodin plugin. It takes as input an Event-B machine and a *splitting specification*, whose structure is described by a domain specific language.

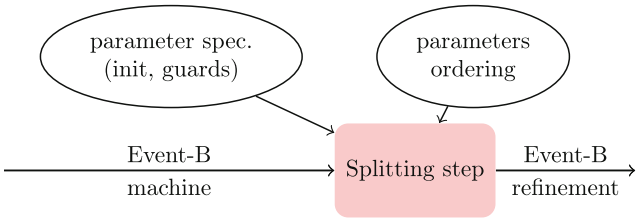

Fig. 4. Event splitting step

```
event ev when p1 ... pn parameter p init v with g1 ... gm
         when ...        parameter  ...
```

We specify for some of the model events, e.g. ev, the parameters (p) to be computed, the parameters on which it depends (p_i), the default value v of p (for typing purposes) and the guards (g_i) acting as the specification of the value of p. The plugin generates a refinement of the input machine.

Such a specification provides a partial order on event parameters. It is used to schedule newly introduced events aiming at computing and storing in a state variable the value of their associated parameter. Ordering constraints are implemented through the introduction of one boolean variable for each parameter, its *computed state*. The machine invariant is extended by the properties of the newly introduced variables: if a variable has been computed, its specification, given by

[4] We consider here that non-determinism is only introduced through event parameters.

its guards, is satisfied. When all the parameters of an event have been computed as state variables, the event itself can be fired. The progress of parameters computation is ensured by a *variant* defined as the number of parameters remaining to be computed. More precisely, the previous specification for parameter p of event ev will produce the following machine contents:

```
machine generated refines input_machine
variables
 ev_p  ev_p_computed //witness and status for parameter p of event ev
invariants
  @ev_gi ev_p_computed ⇒ gi // where p is replaced by ev_p
variant  // count of the remaining parameters to compute
  {FALSE ↦ 1, TRUE ↦ 0}(ev_p_computed) + ...
events
  event INITIALISATION extends INITIALISATION
  then
    @ev_p ev_p := v
    @ev_p_comp ev_p_computed := FALSE
  end

  convergent event compute_ev_p // computes parameter p of event ev
  any p where
   @gi gi // guards acting as p specification
   @pi ev_pi_computed = TRUE //parameters, p depends on, have been computed
   @p ev_p_computed = FALSE // p remains to be computed
  then
    @a ev_p := p  //computed value stored in state variable ev_p
    @computed ev_p_computed := TRUE // makes the variant decrease
  end

  event ev refines ev
  when
    @p_comp ev_p_computed = TRUE
  with
    @p p = ev_p // parameter p of inherited event is refined to ev_p
  then
    @pi ev_pi_computed := FALSE // for all ev_pi with updated guards
    ...   // replace p by ev_p in actions of the refined event
  end
end
```

Listing 1.8. Generated machine for the splitting refinement

An important point is that we get a refinement of the input machine. It should be proved by the user by discharging the standard proof obligations generated by Rodin and has actually been proved for the hotel example. Three main properties should be established: convergent events refine skip as they do not modify inherited state variables and preserve the invariant. They cannot be launched indefinitely as they make the variant (a natural number) decrease. Lastly, the event `ev` is refined as new state variables which take place of the parameters of the inherited event satisfy their guards. The refined invariant is also preserved thanks to the reset of the *computed state* of parameters which depend on guards using updated variables. We can also prove that absence of deadlock is preserved: if the guards of an abstract event are true, the parameters of this event can be or have been computed and lastly the refined event itself can be launched.

Application to Our Example. With respect to our example, the `register` event (see Listing 1.3) has three parameters: `g,r,c`. We specify that the parameter `g` should be computed first as the arrival of a guest is supposed to trigger the various actions. Then, a room is chosen in `r` and its associated card is computed in `c`. For each parameter, we specify its initial value and the name of guards which constitute its specification. The dependencies for the `register` event (see Listing 1.3) are specified as follows:

```
splitting hotel_splitted
refines hotel
events
  event register
    parameter g init g0 with tg // tg does't depend on r,c
    when g parameter r init r0 with tr g1 // fired after computation of g
    when g r parameter c init c0 with tc g2 g3 // fired after g,r
end
```

Listing 1.9. Splitting specification

4.2 The Mapping Step

The aim of this step is to set a distributed implementation over subcomponents of an Event-B centralized model. As for the splitting step, the mapping step takes as input a machine and a *mapping specification* described using a dedicated domain specific language. The user can thus provide a set of subcomponent names and declare a mapping from machine variables and possibly event guards to subcomponents. Then, the tool generates a refinement of the input machine and one projection machine for each subcomponent. This step has two phases: the first one, called the *replication phase*, replicates the variables over the components in order to allow a local access to remote variables; the second one, called the *projection phase*, isolates each component as such. The first phase generates a refinement of the input machine which is in turn refined by the product of its projections, thanks to the shared event decomposition mechanism [19].

The Replication Phase. Given the mapping of machine variables to subcomponents, this phase builds a refinement of the input machine by introducing local copies of distant variables accessed by guards. It maps each guard or action to a component and performs some renaming.

We suppose in the following that variables v_i are mapped on components C_i. The convergent events are shared by source (C_i) and destinations (C_j) of variables remotely accessed by guards. Refinements of inherited events are shared by the sources (C_i) of local copies (on C_j) of variables accessed by guards and by components (C_k) owning variables remotely accessed by actions. Figure 5 presents a component-based view of the transformed model. The focus is put on event *ev* of

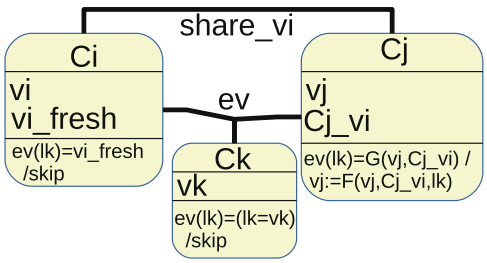

Fig. 5. Local copies and distant access

component C_j. Its guard reads the local copy of v_i while the action has remote access to v_k. Event synchronization ensures the local copy of v_i is up-to-date and gives access to v_k by constraining the event parameter (lk in the figure, `local_vk` in the code pattern).

Listing 1.10 presents the transformation pattern focused on component C_i. The resulting machine should refine the input machine. This is for the moment verified by discharging the proof obligations generated by Rodin. As previously, we plan to establish this result at the meta-level and the arguments will be very similar to those given for the splitting transformation.

```
machine generated refines input_machine
variables
  vi // inherited variables, on Ci
  Cj_vi // copy of vi mapped on Cj (used by a Cj guard)
  vi_fresh // true if vi has been copied, on Ci
invariants
  @Cj_vi_f vi_fresh = TRUE ⇒ Cj_vi = vi // copy is synchronized
variant
  {FALSE ↦ 1, TRUE ↦ 0}(vi_fresh) + ...
events
  convergent event share_vi // shared by Ci and Cj
  any local_vi
  where
    @g vi_fresh = FALSE // on Ci
    @l local_vi = vi // on Ci
  then
    @to_Cj Cj_vi := local_vi // on Cj
    @done vi_fresh := TRUE // on Ci
  end

  event ev refines ev // shared by Ci,Cj,Ck
  any local_vk
  where
    @vj_access local_vk = vk // on Ck, access to remote variables
    @vi_fresh vi_fresh = TRUE // on Ci, copy to Cj has been done
    @g [vi := Cj_vi]g // inherited guard on Cj, access to local copy of vi
  then
    @a vj := [vi := Cj_vi || vk := local_vk]e // on Cj
  end
end
```

Listing 1.10. replication phase

Furthermore, as for the splitting plugin, the freshness of copies is reset when the source variable is updated by an action.

The Projection Phase. It generates a machine for each component, as would do the *shared event decomposition* plugin [19]. However, thanks to the replication phase, guards and actions over remote variables are now accepted. For component C_j, we get the following code template:

```
machine Cj
variables vj Cj_vi
invariants  // keep only those referring vj and Cj_vi
events
  event share_vi // sync with Ci event, import vi
  any local_vi  then
    @to_Cj Cj_vi := local_vi
  end

  event share_vj // sync with Cl event, export vj
  any local_vj then
    @to_Cl local_vj := vj
  end

  event ev
  any local_vk // read by some Cj action
  where
    @vj_fresh vj_fresh = TRUE // needed by Cl, vj has been exported
    @g [vi := Cj_vi]g // mapped on Cj, access to copy of vi
  then
    @a vj := [vi := Cj_vi;vk=local_vk]e
  end
end
```

Listing 1.11. Projection phase

We have to note that some invariants may be lost: we only keep those who refer variables local to the considered component. It means that the correctness of the resulting machines (i.e. the fact that events preserve the remaining invariants) should be proven. If this is not possible, invariants should be added by the user. However, the composition of the projections, as defined in [19], to which lost invariants are added is, by construction, the machine we had before decomposition. As a consequence, thanks to the monotony of composition, the design process can be pursued on each component machine.

Application to Our Example. Listing 1.12 specifies hotel subcomponents and the mapping of the variables `currk owns issued` on the component `Desk` and the variable `cards` on the component `Guest`.

```
components Desk Guest
mappings
  variables currk owns issued ↦ Desk;
  variable cards ↦ Guest;
```

Listing 1.12. Hotel components and mapping specification

4.3 The Code Generation Step

This step assumes that the input Event-B model conforms to a subset of Event-B, we called Event-B0, which plays the role of the subset B0 of the B language that is translated to C. In the considered subset, shared events should be those resulting from the application of the replication phase of the mapping step. Furthermore, we suppose that subcomponent machines do not need to be refined. Events should be deterministic (parameters should have a value) and use a subset

of the Event-B expression and predicate languages for which their exists a direct mapping to their BIP counterparts. For this purpose, we require that used set expressions and predicates have been refined to calls to a `set` library [10] of which signature has a C implementation within the BIP framework. Here, we present how the architectural part of the BIP code is generated. The generator takes as input the mapping specification (subcomponent names, variable and guards mappings) and the refined machine produced by the mapping step.

Port Type Generation. For each shared event and each component of which variables are referenced by this event, we generate a port type taking as parameter the type of exported variables (variables mapped to this component and used by guards or actions mapped to other components). A port type for synchronisation purpose only is generated for all events that do not export variables. Listing 1.5 provides port types generated for our example.

Connector Type Generation. For each event which uses variables of several components, we generate a connector type taking as parameters ports specified by the previously introduced port types. They are supposed to be synchronous. They define a `down` action which copies (via the ports) variables of one component to their copies located in components which need them. Listing 1.6 illustrates the application of this rule in our example.

Subcomponent Skeleton Generation. For each subcomponent, we generate an atomic BIP component. It contains:

- variables mapped to this component as well as variables of other components referenced by guards or actions mapped to this component.
- instances of the port types associated to this component
- for each event, a transition synchronized on the corresponding port instance, and the BIP translation of guards and actions mapped to this component.

As an illustration, Listing 1.5 gives an extract of the atomic component type `ty_Desk` generated by our plugin.

Composite Component Generation. The root component contains an instance of each subcomponent and connector. Each connector instance takes as parameter a port instance defined in one of the concerned subcomponents. Listing 1.7 provides the code of our example resulting from this step.

The generated BIP architecture should for now be completed manually by the data types and behaviors of atomic components. To achieve this, we envision to use the Theory component [10] of the Rodin platform. Indeed, the Theory component allows to develop proved mathematical theories (datatypes, operators, rewrite rules, inference rules). This allows the extension of Event-B by useful data structures such as arrays, linked lists and hash tables.

5 Related Work

Over the last years, several formalisms such as process algebra, input/output automata, UNITY and TLA$^+$ have been proposed to model and mostly to reason

over concurrent and distributed systems. However, to the best of our knowledge, their effective use within development frameworks leading to a distributed implementation has not yet been a general tendency. The automatic generation of source code from formal specifications is supported by few formal methods such as B and Event-B. In [6], an approach is developed allowing the generation of efficient code from B formal developments by using an imperative intermediate language B0. Several Event-B source code generators have been proposed [9,12,20]. Indeed, an Event-B model can represent sequential, concurrent or distributed code as well as reactive, distributed or hybrid systems. The work described in [20] proposes a set of plugins for the Rodin development tool that automatically generate imperative sequential code from an Event-B formal specification. These works do not take into account Event-B composition. Whereas the works described in [9] generate concurrent Ada code restricted to binary synchronization. The automatic refinement of B machines is also possible thanks to the Bart tool [8]. Also, in Event-B, the atomicity decomposition plugin [16] defines a DSL to parametrize the refinement generator. However, the refinement pattern is dedicated to event splitting and does not apply to our problem.

6 Conclusion

In this paper, we have presented a distribution process for system designs formally expressed as Event-B models. Starting from an Event-B machine, the studied process proposes successively the splitting step and the mapping step. The specification of these two steps is done through two domain specific languages. Eventually, a distributed Event-B model and a distributed BIP code architecture are also automatically generated. As we said in the introduction, our primary aim is provide tools to assist the user in the design of distributed systems. Providing a fully automatic process is not in our objectives as we target system engineering and requirements may provide constraints in functions/data to component mapping. Each proposed step generates refinements. The proof obligations generated by Rodin for these refinements remain to be discharged in order to assert the correctness of the developed model.

As future work, we envision to enhance the tooling of our process. Currently, the splitting and mapping steps have been implemented with the xtext [2] language infrastructure, the refinements and the BIP code have been generated with the accompanying xtend language [1] which provides support for writing code generators[5]. We are interested in achieving a distributed code generator plugin for the Rodin platform by taking into account types and the translation of Event-B expression and predicate languages.

We are also interested in studying how the proof obligations generated by the refinements can be discharged definitively at the meta level. In the long term, we seek to enrich the set of transformations and to provide a library of certified transformations dedicated to the development of distributed systems for various architectures.

[5] The generated code is available at https://dl.dropboxusercontent.com/u/98832434/hotelrefinements.html.

References

1. Java 10, today! http://www.eclipse.org/xtend/. Accessed 16 Jan 2006
2. Language engineering for everyone! https://eclipse.org/Xtext. Accessed 16 Jan 2006
3. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
4. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundam. Inf. **77**(1–2), 1–28 (2007)
5. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
6. Bert, D., Boulmé, S., Potet, M.-L., Requet, A., Voisin, L.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003)
7. Butler, M.: A CSP approach to action systems. Ph.D. thesis, Oxford University (1992)
8. Clearsy. Bart (b automatic refinement tool). http://tools.clearsy.com/wp-content/uploads/sites/8/resources/BART_GUI_User_Manual.pdf
9. Edmunds, A., Butler, M.: Tasking Event-B: An extension to Event-B for generating concurrent code. Event Dates: 2nd April 2011, February 2011
10. Edmunds, A., Butler, M.J., Maamria, I., Silva, R., Lovell, C.: Event-B code generation: type extension with theories. In: ABZ Proceedings, pp. 365–368 (2012)
11. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. Softw. Syst. Model. **14**(1), 173–199 (2015)
12. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Heidelberg (2014)
13. Fürst, A., Hoang, T.S., Basin, D., Sato, N., Miyazaki, K.: Formal system modelling using abstract data types in Event-B. In: Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 222–237. Springer, Heidelberg (2014)
14. Jaber, M.: Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP. Theses, Université Joseph-Fourier - Grenoble I, October 2010
15. Nipkow, T.: Verifying a hotel key card system. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 1–14. Springer, Heidelberg (2006)
16. Salehi Fathabadi, A., Butler, M., Rezazadeh, A.: A systematic approach to atomicity decomposition in Event-B. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 78–93. Springer, Heidelberg (2012)
17. Silva, R., Butler, M.: Supporting reuse of Event-B developments through generic instantiation. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 466–484. Springer, Heidelberg (2009)
18. Silva, R., Butler, M.: Shared event composition/decomposition in Event-B. In: Aichernig, B.K., Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects. LNCS, vol. 6957, pp. 122–141. Springer, Heidelberg (2011)
19. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for Event-B. Softw. Pract. Experience **41**(2), 199–208 (2011)
20. Singh, N.K.: EB2ALL: an automatic code generation tool. In: Singh, N.K. (ed.) Using Event-B for Critical Device Software Systems, pp. 105–141. Springer, London (2013)