Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems

D. Diaconescu[2], I. Leustean[2], L. Petre[1], K. Sere[1], and G. Stefanescu[2]

[1] Åbo Akademi University, Finland
[2] University of Bucharest, Romania

Abstract. The state-based formal method Event-B relies on the concept of correct stepwise development, ensured by discharging corresponding proof obligations. The register-voice interactive systems (rv-IS) formalism is a recent approach for developing software systems using both structural state-based as well as interaction-based composition operators. One of the most interesting feature of the rv-IS formalism is the structuring of the components interactions. In order to study whether a more structured (rv-IS inspired) interaction approach can significantly ease the proof obligation effort needed for correct development in Event-B, we need to devise a way of integrating these formalisms. In this paper we propose a refinement-based translation from Event-B to rv-IS, exemplified with a file transfer protocol modelled in both formalisms.

1 Introduction

Event-B [2,9,14,15,16,17,18] is a state-based formalism dedicated to the refinement-based development of parallel and distributed systems. This amounts to developing an abstract model into more concrete ones, so that we are sure that a more concrete model correctly develops a more abstract one. A central advantage of Event-B is the associated Rodin tool platform [27,3] employed in discharging the proof obligations that ensure this correct development. In addition to providing a user interface for editing Event-B models, the proving process is closely integrated with the modelling process, encouraging proof-based model improvement. Event-B is currently successfully integrated in several industrial developments, for instance at Space Systems Finland [13] and at SAP [8].

The register-voice interactive systems [24,25,21,11,12,23] (rv-IS) formalism is a recent approach for developing software systems using both structural state-based as well as interaction-based composition operators. Interactive computation [29] is an important computer science topic, often related to human-computer interaction, the particular case when one of the interacting entities is human. While able to deal with such cases as well, the rv-IS formalism is more oriented to the process-to-process interaction. There are already many successful formalisms for this, including Petri nets [26], process algebras [5], π-calculus [20,19], dataflow networks [6,7], etc. The approach used in this paper integrates a dataflow-like interaction model with a classical state-based computation model. One of the most interesting feature of the rv-IS formalism is the structuring of the component interactions.

Our aim is to study whether a more (rv-IS inspired) structured approach of an interactive, modular system has any effect on the correct development as designed in Event-B. More precisely, we are interested in uncovering whether the proof obligations are significantly eased when a certain structure is assumed in the model. For this, we need to devise an integration of Event-B and rv-IS, up to a level where the key features of each formalism can be easily translated into the other. We have set up the following working plan for integrating the Event-B and the rv-IS formalisms:

1. Define a notion of refinement in rv-IS models based on a combination of the refinement of state-based systems and of Broy-style refinement of dataflow-based interactive systems.
2. Define a translation EB2IS from Event-B models to structured rv-IS models.
3. Prove the translation EB2IS preserves refinement.
4. Use one of the known translations to pass from structured rv-IS models to unstructured rv-IS models, e.g, the translation in [12].
5. Define a refinement preserving translation IS2EB from unstructured rv-IS models to Event-B models.
6. Use these translations EB2IS and IS2EB to: (1) improve the discharging of proof obligations in Event-B based on rv-IS structural operators and associated decomposition techniques; (2) get tool support to develop and analyze rv-IS models.

In this paper we present a double-folded contribution. First, we introduce a refinement-preserving translation EB2IS from Event-B models to structured rv-IS models. Second, we argue our translation by analyzing an example: we present three refinement steps for modeling a simple file transfer protocol in Event-B and show the associated refined structured rv-IS models. This means we are addressing items 2. and 3. in our working plan above. We have already addressed item 1. in [10] and item 4. in [12].

We proceed as follows. In Section 2 we outline Event-B and rv-IS. In Section 3 we introduce a general translation from Event-B models to rv-IS models and briefly put forward the concept of rv-IS refinement. In Section 4 we present an example of a file transfer protocol and in Section 5 we conclude the paper.

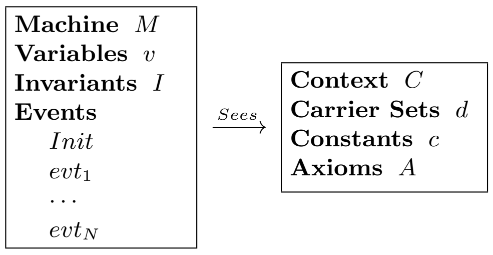
2 Preliminaries

In this section we overview the formalisms to integrate to the extent needed in this paper.

2.1 Event-B

Event-B [2] is a state-based formal method focused on the stepwise development of correct systems. This formalism is based on Action Systems [4,28] and the B-Method [1]. In Event-B, the development of a model is carried out step by step from an abstract specification to more concrete specifications.

The general form of an Event-B model is illustrated in the side figure. Models in Event-B consist of *contexts* and *machines*. A context describes the static part of a model, containing sets and constants, together with axioms about these. A machine describes the dynamic part of a model, containing variables, invariants (boolean predicates on the variables), and events, that evaluate (via event *guards*) and modify (via event *actions*) the variables. The guard of an event is an associated boolean predicate on the variables, that determines if the event can execute or not. If the event can execute, then we say it is *enabled*. The action of an event is a parallel composition of either deterministic or non-deterministic assignments. Upon executing the initializing event $Init$, computation proceeds by a repeated, non-deterministic choice and execution of an enabled event. If none of the events is enabled then the system deadlocks. The relationship *Sees* between a machine and its accompanying context denotes a structuring technique that allows the machine access to the contents of the context.



A machine M and a context C in Event-B

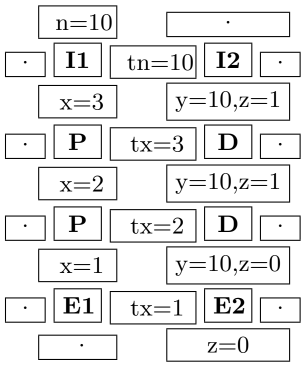
The semantics of events is defined using *before-after (BA) predicates* [2]. A before-after predicate describes a relationship between the system states before and after the execution of an event. The semantics of a whole Event-B model is formulated as a number of *proof obligations*, expressed in the form of logical sequents. The full list of proof obligations can be found in [2]. Every Event-B model should satisfy the event feasibility and invariant preservation properties. The feasibility of an event means that, whenever the event is enabled, its BA predicate is well-defined, i.e., there is some reachable after-state. Each event should also preserve the given model invariant. The formal semantics provides us with a foundation for establishing correctness of Event-B specifications.

System Development. Event-B employs a top-down refinement-based approach to formal system development. Development starts from an abstract system specification that models some essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into an abstract specification. These new events correspond to stuttering steps that are not visible in the abstract specification. We call such model refinement as *superposition refinement.* Moreover, Event-B formal development supports *data refinement*, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants are called *gluing invariants.*

In order to prove the correctness of each step of the development, a set of proof obligations needs to be discharged. Thus, in each development step we have mathematical proof that our model is correct. The model verification effort and, in particular, the automatic generation and proving of the required proof obligations, are significantly facilitated by the provided tool support – the Rodin platform [27,3].

2.2 Register-voice interactive systems

The rv-IS formalism is built on top of register machines, closing them with respect to a space-time duality transformation. Specifically, we use the model, the core programming language, the specification formalism and the analysis techniques developed for modeling, programming and reasoning about interactive computing systems by the last author and coworkers in the recent years, see [24,25,21,11,12,23]. In the following, we shortly overview the approach.



```
module I1
{listen nil}{read n}{
  tn:tInt; x:Int;
  tn = n; x = sqrt(n);
}{speak tn}{write x}
```

```
module I2
{listen tn}{read nil}{
  y,z:Int;
  y = tn; z = 1;
}{speak nil}{write y,z}
```

```
module P
{listen nil}{read x}{
  tx:tInt;
  tx = x; x = x-1;
}{speak tx}{write x}
```

```
module D
{listen tx}{read y,z}{
  if(y % tx = 0){
   z = 0;};
}{speak nil}{write y,z}
```

```
module E1
{listen nil}{read x}{
  tx:tInt; tx = x;
}{speak tx}{write nil}
```

```
module E2
{listen tx}{read y,z}{
  null;
}{speak nil}{write z}
```

Fig. 1. A scenario and the modules of the **Prime** rv-IS program

Scenarios. A *scenario* is a two-dimensional rectangular area filled in with identifiers and enriched with data around each identifier. In our interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline. This means that a process sends a message to the right, then it resumes its execution. *(Memory) states* are placed at the north and at the south borders of the identifiers and *(interaction) classes* are placed at the west and at the est borders of the identifiers. In the the left-hand side of Fig. 1 we illustrate an rv-IS scenario for deciding whether the number 10 is prime. We explain this example in detail at the end of this section.

Spatio-temporal specifications. A spatio-temporal specification combines constraints on both spatial and temporal data. For the spatial data, we use the common data structures and their natural representations in memory. For representing temporal data we use streams: a *stream* is a sequence of data ordered in time and is denoted as $a_0 \frown a_1 \frown \ldots$, where $a_0, a_1, \ldots$ are the data laying on the stream at time $0, 1, \ldots$, respectively.

A *voice* is defined as the time-dual of a register. Voices are simple temporal structures, represented on streams, that hold natural numbers. The value of a voice can be modified in a location and then propagated within the system. A voice can be "listened" at various locations, at each location the piece of stream representing the voice displaying a particular value. Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. Most of usual data structures have natural temporal representations. Examples includes timed booleans, timed integers (denoted `tInt`), timed arrays, timed lists, etc.

The notation $\otimes$ is used for the product of memory states, while $^\frown$ for the product of interaction classes; $\mathbb{N}^{\otimes k}$ denotes $\mathbb{N} \otimes \ldots \otimes \mathbb{N}$ (k terms) and $\mathbb{N}^{\frown k}$ denotes $\mathbb{N}^\frown \ldots {}^\frown\mathbb{N}$ (k terms); the associated "star" operations are denoted as $(\_\_^{\otimes})^*$ and $(\_\_^{\frown})^*$.

A simple *spatio-temporal specification* $S : (m,p) \to (n,q)$ is a relation $S \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$, where m (resp. p) is the number of input voices (resp. registers) and n (resp. q) is the number of output voices (resp. registers). More general spatio-temporal specifications may be introduced using complex interface types, not only registers and voices.

Syntax of structured rv-programs. The *type* of a *structured rv-program* P, denoted by

$$P : (w(P), n(P)) \to (e(P), s(P)),$$

collects the types at the west, north, east and south borders of its scenarios. In general, these are relatively complex types built up from boolean and integer types - see the concrete types used in Agapia v0.1 programming language [11].

The syntax of structured rv-programs is defined as follows:

```
P ::= null | X | P % P | P # P | P $ P | if(C) then {P} else {P}
      | while_t(C) {P} | while_s(C) {P} | while_st(C) {P}
```

The starting blocks for the construction of structured rv-programs are called *modules*. The syntax of a module is given as follows:

```
module module_name
{listen temporal_variables}{read spatial_variables}{
  code
}{speak temporal_variables}{write spatial_variables}
```

where the `read` (resp. `listen`) instruction collects the spatial (resp. temporal) input and the `write` (resp. `speak`) instruction returns the spatial (resp. temporal) output. The `code` consists in instructions similar to the C code.

The operations on structured rv-programs are briefly described below. More details and examples may be found in [24,11,12].

1. **Composition:** Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types agree. They can also be composed diagonally by mixing the horizontal and vertical composition.
   (a) For two programs $\mathtt{P_i} : (w_i, n_i) \to (e_i, s_i)$, $i = 1, 2$, the *horizontal composition* $\mathtt{P}_1\#\mathtt{P}_2$ is well-defined only if $e_1 = w_2$; the type of the composite is $(w_1, n_1 \otimes n_2) \to (e_2, s_1 \otimes s_2)$.
   (b) Similarly, the *vertical composition* $\mathtt{P}_1\%\mathtt{P}_2$ is well-defined only if $s_1 = n_2$; the type of the composite is $(w_1{}^\frown w_2, n_1) \to (e_1{}^\frown e_2, s_2)$.
   (c) The *diagonal composition* $\mathtt{P}_1\$\mathtt{P}_2$ is a derived operation - it connects the east border of $\mathtt{P}_1$ to the west border of $\mathtt{P}_2$ and the south border of $\mathtt{P}_1$ to the north border of $\mathtt{P}_2$; it is defined only if $e_1 = w_2$ and $s_1 = n_2$; the type of the composite is $(w_1, n_1) \to (e_2, s_2)$.

2. **If:** For the "if" operation, given two programs with the same type $\mathtt{P}, \mathtt{Q} : (w, n) \to (e, s)$, a new program `if(C) then {P} else {Q}` $: (w, n) \to (e, s)$ is constructed, for a condition `C` involving both, the temporal variables in w and the spatial variables in n.
3. **While:** There are three while statements, each being the iteration of the corresponding composition operation.
   (a) For a program $\mathtt{P} : (w, n) \to (e, s)$, the *temporal while* statement `while_t(C){P}` is defined if $n = s$ and C is a condition on the variables in $w \cup n$. The type of the result is $((w^\frown)^*, n) \to ((e^\frown)^*, n)$.
   (b) The case of *spatial while* `while_s(C){P}` is similar.
   (c) If $\mathtt{P} : (w, n) \to (e, s)$, the statement `while_st(C){P}` is defined if $w = e$ and $n = s$ and C is a condition on $w \cup n$. The type of the result is $(w, n) \to (e, s)$.

Operational semantics of structured rv-programs. The operational semantics is given in terms of scenarios. Scenarios are built up with the following procedure:

1. Each cell of the associated grid has as label a module name.
2. An area around a cell may have additional information. For example, if a cell has the information `x` = `2`, that means that in that area `x` is updated to be 2.
3. The scenario is built from the current rv-program by reducing it to simple compositions of spatio-temporal specifications w.r.t. the syntax of the program, until we reach basic blocks, e.g. modules.

Example. We illustrate the operational semantics by considering an example:

```
(I1 # I2) % while_t(x > 1){P # D} % (E1 # E2)
```

This is a structured rv-program **Prime** verifying if a number `n` is prime. Its modules are listed in the right-hand side of Fig. 1.

Our rv-IS program has two processes: one generates all the numbers in the set $\{\lfloor\sqrt{\mathtt{n}}\rfloor, \ldots, 2\}$ (the `P` module) and the other checks if a number is a divisor of `n` (the `D` module) as well as updates a variable `z`. Modules `I1` and `I2` are used for initializations and `E1` and `E2` for ending. At the end of the program, if the variable `z` is `1`, then the number `n` is prime.

In order to show how we can construct a scenario for the rv-IS program above we consider a concrete example for `n = 10`. The corresponding scenario is presented in the right-hand side of Fig. 1. In the first line of the scenario we initialize the processes with the needed information. Module `I1` reads the value `n = 10`, provides the first process with the square root of `n`, i.e., `x = 3`, and declares a temporal variant of `n`, namely `tn = 10`. This is used by module `I2` to initialize the process with the initial value of `n`, namely `y = 10`; in this module we also set `z = 1` (hence initially, we assume `n` is prime). In the next step, module `P` produces a temporal data `tx = 3` (`tx` is equal with the data `x` of the first process) and decreases `x`. Module `D` verifies if `tx` is a divisor of `y` and, if it is, then it resets the value of `z` to `0`. We repeat these steps until the variable `x` becomes `1`. The last line contains ending modules that only change the interfaces.

The scenarios may be constructed in various ways. For instance, programs building the scenarios by columns [10] exhibit a dataflow computation style.

3 From Event-B to structured rv-IS

In this section we introduce a general method for translating an Event-B system specification into an rv-IS specification. The method actually produces an rv-program, whenever the transformations used to define the actions of the events can be implemented with a code written in the rv-module code syntax. We also describe shortly our approach to the refinement of rv-IS [10]. In this paper, we are concerned with the events of a certain system, not with its invariants.

An Event-B model can be seen as a set of events of the form presented in the box on the right, where for each `i`, `Event-i` is the name of the event, `Grd-i` is the guard and `Act-i` is the action, so that `Grd-i` and `Act-i` are sets of predicates, respectively actions. We denote with `Ainit` the actions of the event `Init`.

```
Event-i
 when
  Grd-i
 then
  Act-i
 end
```

The rv-IS specification associated to an Event-B model captures not only the model, but also the semantic rules used for its execution. In order to construct a structured rv-IS specification from an Event-B model, we define a *manager* that decides which event can take place at each time. For each event `Event-i`, we construct two modules `Gi` and `Ei` - modules `Gi` are used by the manager in order to decide which event to be triggered, while modules `Ei` are used by the manager to describe the state changes caused by the event.

In Event-B the memory is shared by all the events, hence in the associated rv-IS specification we need to simulate this common memory. Therefore, after each action, the manager must update the variables of all the processes.

```
1:   (I # for_s(j=1,N){ID})
2: $ (Mg # for_s(j=1,N){Gj})
3: $ while_st(ten ≠ ∅) {
4:        (Me # for_s(j=1,N){Ej})
5:      $ (Mu # for_s(j=1,N){U}))
6:      $ (Mg # for_s(j=1,N){Gj})
7:   }
```

Table 1. The formula for EB2IS translated model

Assume that the Event-B model to translate has `N` events, in addition to the `Init` event. We define the set $\mathtt{Ev} = \{\mathtt{Ei} \mid \mathtt{i} = \overline{\mathtt{1,N}}\}$ and we denote by `C` the set of all the constants and by `V` the set of all the variables of the Event-B model.

The general format of the corresponding rv-IS specification is presented in Table 1. (The `for_s` statement is derived from `while_s` in the natural way.)

Module `I` contains all the initializations from the event `Init` in Event-B and module `ID` provides the same variables to all the processes involved in the program. The manager uses the modules `Mg`, `Me` and `Mu` to simulate the behavior in Event-B and to decide which event can take place next. In line 2, the manager constructs the set `ten` of enabled events by checking their guards; the module `Gj` checks the guard of the event `Event-j`. While we have at least one enabled event, we start to simulate its behavior. In line 4, the manager chooses one event from the list of enabled events at the current moment and starts to search for the process modeling the execution of this event. Module `Ej` modifies in the system with respect to actions `Act-j` if `Event-j` is the chosen one. In line 5, the manager updates the variables in all the processes with respect to the new modifications. After this, we repeat the procedure until no more events can occur, as described in line 6.

`module I {listen nil}{read C,V}{ Ainit; tV = V ∪ C; }{speak tV}{write nil}`	`module Gi {listen ten}{read V}{ if(Grd-i){ten=ten∪{Event-i};}; }{speak ten}{write V}`	`module Mg {listen tV}{read nil}{ ten = ∅; }{speak ten}{write nil}`
`module ID {listen tV}{read nil}{ V = tV; }{speak tV}{write V}`	`module Ei {listen tk,tV}{read V}{ if(tk=Event-i){Act-i; tV=V;} }{speak tk,tV}{write V}`	`module Me {listen ten}{read nil}{ tk :∈ ten; tV = ∅; }{speak tk,tV}{write nil}`
`module U {listen tV}{read V}{ V = tV; }{speak tV}{write V}`		`module Mu {listen tk,tV}{read nil}{ null; }{speak tV}{write nil}`

Table 2. Modules for EB2IS translation

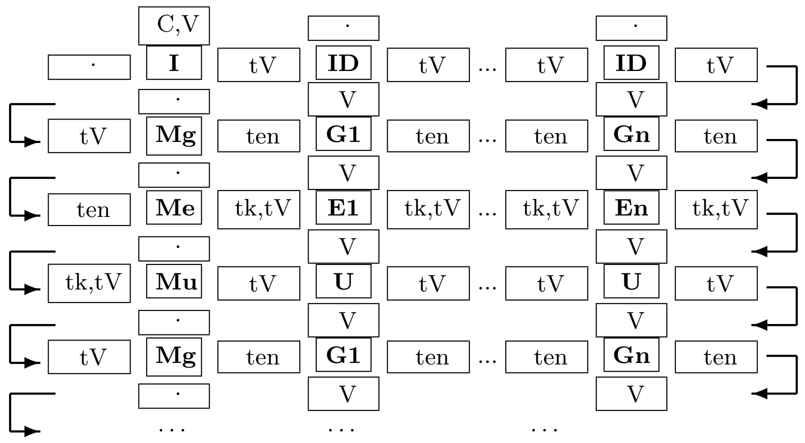


Fig. 2. Scenarios for an rv-IS obtained from an Event-B model

The behavior of the manager is split in the following actions: search for the ‘chosen’ event (lines 2 and 6), modify the system with respect to the actions of the ‘chosen’ event (line 4), and update the variables of all processes (line 5). In order to take the next action, the manager needs information from the previous action, therefore we must compose the parts of the program diagonally. The modules of the associated rv-IS specifications are described in Table 2.

In this translation the manager decides in an nondeterministic fashion which event can take place next; however, in module `Me` we can implement any method for deciding this. The manager described above chooses one single event (`tk :∈ ten`; `tk` is a single token). In a more general implementation, the manager is free to choose a set of events that can take place at a certain moment of time, by constructing `tk` to be a set. In such a case, we have to avoid written conflicts for updated variables occurring in more than one event. A general scenario for the program above is presented in Fig. 2.

Refinement of register-voice interactive systems We associate a graph $Gr(S)$ to a scenario S, with the following procedure: (1) we give a proper name to each tuple (w, n, e, s) of data surrounding a scenario cell; a cell is called an *identity* if $(e = w \vee e = n) \wedge (s = n \vee s = w)$; (2) we replace the identifiers of the cells by these names; (3) the graph $Gr(S)$ has as nodes the scenario non-identity cells and as edges connections via identity nodes of their $w/n/e/s$ ports.

Two scenarios $S1$ and $S2$ are *equal up-to-stuttering* of states and classes if the graphs $Gr(S1)$ and $Gr(S2)$ are isomorphic. $S2$ *up-to-stuttering includes* $S2$ if $Gr(S1)$ and $Gr(S2)$ have the same nodes and the edges of $Gr(S1)$ are included in the edges of $Gr(S2)$.

For two rv-IS models $IS1$ and $IS2$, we say $IS2$ is a *refinement* of $IS1$ if: (1) up to a connecting relation between the states and classes of $IS1$ and $IS2$, each scenario of $IS2$ up-to-stuttering includes a scenario of $IS1$; (2) if a scenario of $IS2$ is related to a scenario of $IS1$ and the latter may be extended in $IS1$, then the former may be extended in $IS2$.

As an example, consider the scenarios in Fig. 3. If `Y,Z,U,W` are identity nodes so that `b=d, B=C=D=E` and `F=H`, then $S1$ is up-to-stuttering equal to $S2$.

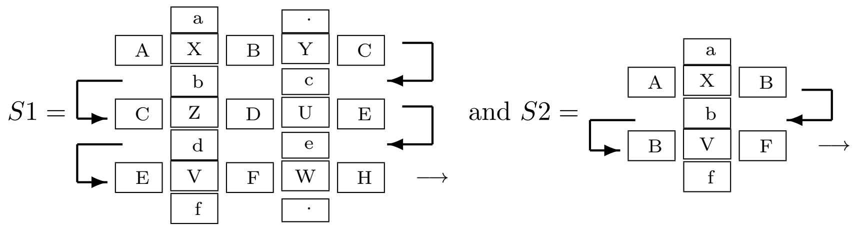


Fig. 3. Two up-to-stuttering equal scenarios

4 An example - a simple file transfer protocol

In this section we translate an Event-B model into an rv-IS specification.

The model is that of a classical file transfer protocol, also described in [2]. The file to be transferred is sequential, i.e. composed of a finite number of items arranged in a specific order. The file has to be sent from one agent - the sender, to another one - the receiver. The transferred file should be equal to the original one. The protocol is distributed, realized by two distinct modules that exchange various kinds of messages and reside in different sites.

We develop the protocol in three steps. Initially, we are interested only in the final result of the protocol, not in how it is achieved. The file in this model is transmitted in one shot and the agents do not reside on different sites. In the first refinement, we transmit the file piece by piece between the two agents. The main difference with respect to the initial model is that we separate the sender and the receiver. They are still not completely independent, since the receiver can still access the sender's memory. In the second refinement, the sender and the receiver are completely independent from each other and the receiver has no longer access to the sender's memory. In this stage, the two agents communicate only through messages: the sender is sending messages that are read by the receiver and the receiver responds to these messages by returning an acknowledgement message to the sender. The distributed nature of the protocol is therefore revealed in this final refinement step.

Initial model We assume a nonempty set D (the carrier set) and two constants: n is a positive number and f is a total function from $\{1, \ldots, n\}$ to D. Informally, f is the file to be transferred, the constant n represents the length of the file f, while D contains the data that can be stored in the file f. We represent the file f as a total function with elements in D. The result of the protocol is a variable g, the file transferred to the receiver. Since we construct g step by step, we model g as a partial function from $\{1, \ldots, n\}$ to D.

In the initial model, we say nothing about the internal structure of the file f. In order to transfer the file, we have an event `receive` that chooses randomly a partial function g with values in D, until this function is equal to f. When we obtain such a function g, then we can assume that the file f was sent to the receiver's site.

The Event-B events of this initial model FTP-EB1 are the following:

FTP-EB1 ::=

```
init
  $g :\in \mathbb{N} \leftrightarrow D$
```

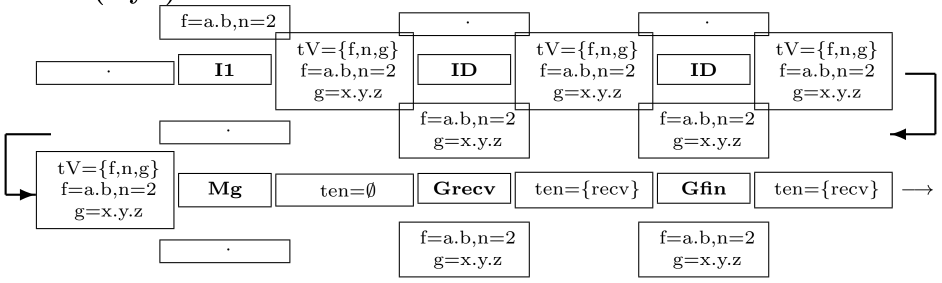
```
receive
  when
    $g \neq f$
  then
    $g :\in \mathbb{N} \leftrightarrow D$
  end
```

```
final
  when
    $g = f$
  then
    skip
  end
```

In order to construct an rv-IS specification FTP-IS1, let us consider the following set of events $\texttt{Ev} = \{\texttt{Erecv}, \texttt{Efin}\}$. The specification is presented in Table 3, where the involved modules `I, Grecv, Erecv, Gfin, Efin` are described in Table 4. In the initial model we have the modules `I, Grecv, Erecv, Gfin, Efin` subscribed by 1, in the second model these modules are subscripted by 2, and in the final model these modules are subscripted by 3.

Let us analyze a simple case: suppose that `f` contains only two characters, say `f=a.b`; thus `n=2`. A typical scenario for the FTP-IS1 specification is built up using partial scenarios illustrated below. In the presentation, `g=x.y.z, g=s.t, ..., g=a.b` is just a sequence of random assignments for `g`. Alternatively, one can consider the case where the lucky assignment `g=a.b` never occurs.

Init1(x.y.z) =



The first scenario `Init1(x.y.z)` (above) is an initialization step that starts with the given data `f,n`. The random assignment `g=x.y.z` generates the initial data for all the processes associated to the events, i.e., for the `recv` and `fin` processes. In addition, the scenario starts to check the validity of the guards: in this case the guard of the `recv` event is true and `recv` is exported on the last line.

```
FTP-IS1 =
   (I1 # ID # ID)
  $ (Mg # Grecv1 # Gfin1)
  $ while_st(ten ≠ ∅) {
          (Me # Erecv1 # Efin1)
        $ (Mu # U # U)
        $ (Mg # Grecv1 # Gfin1)
   }
```

```
FTP-IS3 =
   (I3 # ID # ID # ID)
  $ (Mg # Grecv3 # Gsend3 # Gfin2)
  $ while_st(ten ≠ ∅) {
          (Me # Erecv3 # Esend3 # Efin1)
        $ (Mu # U # U # U)
        $ (Mg # Grecv3 # Gsend3 # Gfin2)
   }
```

Table 3. Formulas for FTP-IS1 and FTP-IS3 specifications

`module I1` `{listen nil}{read f,n}{` `g :∈ ℕ ↔ D;` `tV = {f,n,g};` `}{speak tV}{write nil}`	`module I2` `{listen nil}{read f,n}{` `g = ∅; r = 1;` `tV = {f,n,g,r};` `}{speak tV}{write nil}`	`module I3` `{listen nil}{read f,n}{` `g = ∅; r = 1 ; s = 1;` `d :∈ D; tV = {f,n,g,r,d};` `}{speak tV}{write nil}`
`module Grecv1` `{listen ten}{read V}{` `if(g ≠ f){` `ten = ten ∪ {Erecv1};};` `}{speak ten}{write V}`	`module Grecv2` `{listen ten}{read V}{` `if(r ≤ n){` `ten = ten ∪ {Erecv2};};` `}{speak ten}{write V}`	`module Grecv3` `{listen ten}{read V}{` `if(s = r+1){` `ten = ten ∪ {Erecv3};};` `}{speak ten}{write V}`
`module Erecv1` `{listen tk,tV}{read V}{` `if(tk = Erecv1){` `g :∈ ℕ ↔D;` `tV = V;};` `}{speak tk,tV}{write V}`	`module Erecv2` `{listen tk,tV}{read V}{` `if(tk = Erecv2){` `g = g ∪ {r ↦ f(r)};` `r = r+1; tV = V;};` `}{speak tk,tV}{write V}`	`module Erecv3` `{listen tk,tV}{read V}{` `if(tk = Erecv3){` `g = g ∪ {r↦d}; r = r+1;` `tV = V;};` `}{speak tk,tV}{write V}`
`module Gfin1` `{listen ten}{read V}{` `if(g = f){` `ten = ten ∪ {Efin1};};` `}{speak ten}{write V}`	`module Gfin2` `{listen ten}{read V}{` `if(r = n+1){` `ten = ten ∪ {Efin2};};` `}{speak ten}{write V}`	`module Gsend3` `{listen ten}{read V}{` `if(s=r ∧ r≠n+1){` `ten = ten ∪ {Esend3};};` `}{speak ten}{write V}`
`module Efin1` `{listen tk,tV}{read V}{` `if(tk = Efin1){` `tV = V;};` `}{speak tk,tV}{write V}`	`Efin2 = Efin1` `Mg, Me, Mu, U` and `ID` are as in Table 2	`module Esend3` `{listen tk,tV}{read V}{` `if(tk = Esend3){` `d = f(s); s = s+1;` `tV = V;};` `}{speak tk,tV}{write V}`

Table 4. Modules for FTP-IS1 to FTP-IS3 specifications

The second scenario `DoRecv1(x.y.z;s.t)` (next page) corresponds to the application of the `recv` event, resulting in a state change from `g=x.y.z` to `g=s.t`. Hopefully, after a number of such steps, the random assignment leads to `g=a.b`: in that case, the exported guard in the last line is `fin`, not `recv`. If `fin` holds, then the last scenario `DoFin1` (next page) applies. In this part, the `fin` event has no actions, so nothing changes in the states. Therefore, this repeats forever.

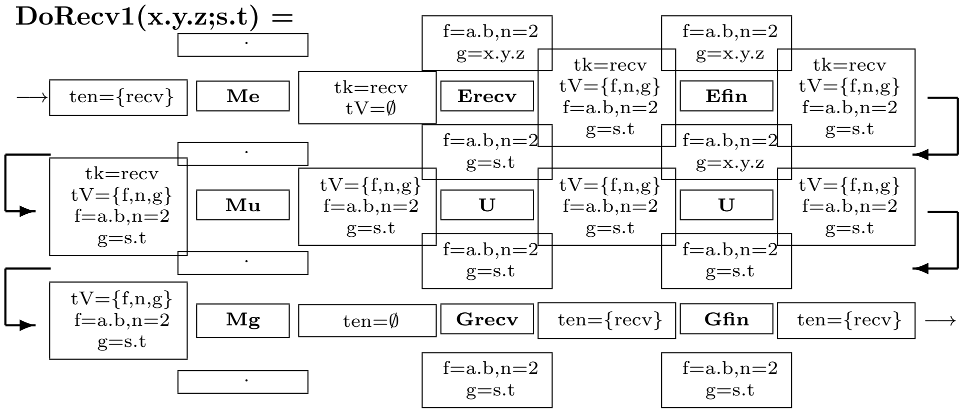
First refinement In the first refinement, we modify the event `receive` in order to send concrete parts of the file f. The event `receive` will no longer produce files randomly until it obtains one equal with f. Instead, it sends one element of the file f at each step. For this we introduce a new variable r which models an index of the file f. At each step, the r-th element of f is copied in the file g of the receiver's site. The file transfer is finished when r is greater than n.

The refinement FTP-EB2 of our model in Event-B has the following events:

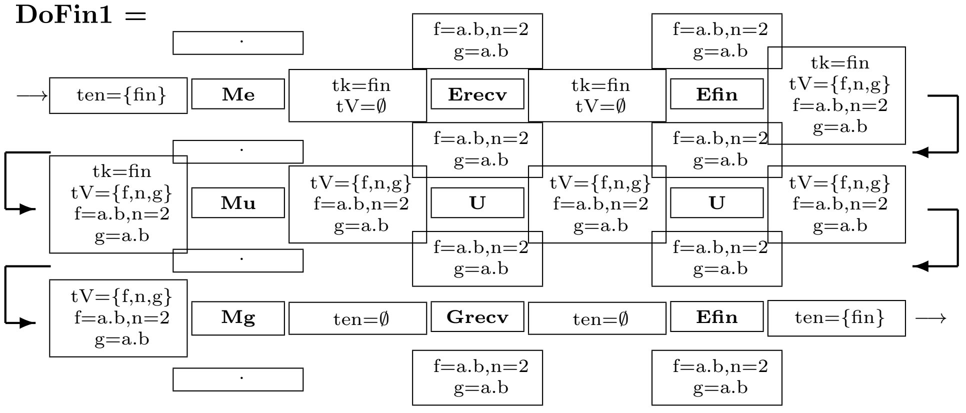
FTP-EB2 ::=

```
init
  $g := \emptyset$
  $r := 1$

receive
  when
    $r \leq n$
  then
    $g := g \cup \{r \mapsto f(r)\}$
    $r := r + 1$
  end

final
  when
    $r = n + 1$
  then
    skip
  end
```

DoRecv1(x.y.z;s.t) =



DoFin1 =



The corresponding rv-IS specification FTP-IS2 uses the same formula as in the case of the initial model, but with modules `I1, Grecv1, Gfin1, Erecv1` slightly changed: they are replaced by the new modules `I2, Grecv2, Gfin2, Erecv2` listed in Table 4.

Let us analyze the above case again: suppose that `f=a.b,n=2`. The running scenario is unique (deterministic) this time and consists of an initial action, followed by `n` times repeated `recv` actions, followed by repeated `fin` actions. A detailed presentation appears in [10].

Second refinement In the last refinement step we split the event `receive` in Event-B into two corresponding events, `send` and `receive`. The indexes s and r model the positions of the current file item (to be) sent and the next position where a file item is to be received, respectively. The event `send` models the activity of the sender, that forms a message d to be sent by copying in d the file item at position s. The event `receive` models the activity of the receiver, that stores the message d as the file item at position r. Hence, we now have a distributed file transfer protocol where the sender and the receiver communicate by sending message d and sharing variables r and s.

FTP-EB3 ::=

```
init
  $g = \emptyset$
  $s = 1$
  $r = 1$
  $d :\in D$

send
  when
    $s = r$
    $r \neq n + 1$
  then
    $d = f(s)$
    $s = s + 1$
  end

receive
  when
    $s = r + 1$
  then
    $g = g \cup \{r \mapsto d\}$
    $r = r + 1$
  end

final
  when
    $r = n + 1$
  then
    skip
  end
```

For presenting the associated rv-IS specification FTP-IS3, we fix the following set of events $\texttt{Ev} = \{\texttt{recv}, \texttt{send}, \texttt{fin}\}$, also adding the subscript 3 to indicate that we are at the third modeling level. The specification is presented in Table 3 and the modules in Table 4. The scenarios can be constructed in a similar way as shown for FTP-IS1. The particular case when `f=a.b`, with scenarios named `Init3`, `DoSend3(a)` and `DoRecv3(a)` is discussed in [10].

Refinement preservation. The state space of FTP-IS1 is $S1 = \{\texttt{f}, \texttt{n}, \texttt{g}\}$, of FTP-IS2 is $S2 = \{\texttt{f}, \texttt{n}, \texttt{g}, \texttt{r}\}$ and of FTP-IS3 is $S3 = \{\texttt{f}, \texttt{n}, \texttt{g}, \texttt{r}, \texttt{s}, \texttt{d}\}$. The class space of FTP-IS2 is $C2 = \{\texttt{ten}, \texttt{tk}, \texttt{tV} = (\texttt{f}, \texttt{n}, \texttt{g}, \texttt{r})\}$, departing from that of FTP-IS1 $C1 = \{\texttt{ten}, \texttt{tk}, \texttt{tV} = (\texttt{f}, \texttt{n}, \texttt{g})\}$ by the type of `tV`. The class space of the last model FTP-IS3 is $C3 = \{\texttt{ten}, \texttt{tk}, \texttt{tV} = (\texttt{f}, \texttt{n}, \texttt{g}, \texttt{r}, \texttt{s}, \texttt{d})\}$. FTP-IS3 has a new event `send` and the sets used for $\texttt{ten}, \texttt{tk}$ are larger, including this new element.

Proposition 1. *(a) FTP-IS2 is a refinement of FTP-IS1; and (b) FTP-IS3 is a refinement of FTP-IS2.*

Proof: (Outline) For (a), let $\rho = (\rho_s, \rho_c)$ be a relation between the states and classes of FTP-IS2 and FTP-IS1, where $\rho_s : S2 \to S1$ and $\rho_c : C2 \to C1$ are the natural projections that abstract `r` away. If `Scen` is a scenario in FTP-IS2, then $\rho(\texttt{Scen})$ is a scenario in FTP-IS1. For (b), let $\rho = (\rho_s, \rho_c)$ be a relation between the FTP-IS3 and FTP-IS2, where $\rho_s : S3 \to S2$ and $\rho_c : C3 \to C2$ are the natural projections that abstract $\texttt{s}, \texttt{d}, \texttt{send}$ away. If `Scen` is a scenario in FTP-IS3, then $\rho(\texttt{Scen})$ is a scenario in FTP-IS2 up to sub-scenario stuttering corresponding to the application of the macro-steps associated to the `send` event and to the column corresponding to the `send` event. Indeed, this latter sub-scenarios have no visible effect on the states and classes of FTP-IS2. These arguments demonstrate condition (1) in the refinement definition at the end of Section 3.

Condition (2) in this definition is valid for case (a): if `Scen1` is a partial scenario in FTP-IS2 and the scenario $\rho(\texttt{Scen1})$ can be extended in FTP-IS1, then the same is true for `Scen1` in FTP-IS2. A similar property holds for (b). □

5 Conclusions

Our motivation in this paper is based on one quintessential feature of Event-B and its associated Rodin platform. Modeling in Event-B is semantically justified by proof obligations. Every update of a model generates a new set of proof obligations in the background. It is this interplay between modeling and proving that sets Event-B apart from other formalisms. Without proving the required obligations, we cannot be sure of correctness of a model. The proving effort thus encourages the developer to structure formal model development in such a way that manageable proof obligations are generated at each step. This leads to very abstract initial models so that we can gradually introduce into a system model various facets of the system. Such a development method fits well when we have to describe complex algorithms.

However, it is not obvious how to structure the development of a model, what to model in the initial specification, and what other details to introduce in each of the following refinements. This is especially true when considering the generated proof obligations, because differently structured developments generate different sets of obligations. Several structuring mechanisms have been presented before for Event-B, for instance in [9], to address the complexity of system development. The problem of structuring the development has also been observed before in the efforts to develop the `Flow`-plugin in the Rodin platform [27], to address the event ordering and enabledness conditions of a model. In this paper we bring forward the enabledness of events as well as the choice of the event to execute next, via the manager modules `Me,Mu,Mg` in rv-IS. Each event is seen as an independent process that is activated when enabled. The interactions between events are 'normalized' to sharing the variables, but in fact new values (hence interactions) occur only upon the execution of an enabled event. This puts forward a clear separation between computation and communication and is resemblant of employing Event-B||CSP in [22], to demonstrate (with the help of the same case study) an explicit approach to control flow.

The main contribution of our paper consists in the definition of a translation EB2IS from Event-B models to structured rv-IS models. Moreover, we provide evidence that the translation preserves refinement, by considering a refinement chain of relatively complex Event-B models and the corresponding translated chain of rv-IS models. As refinement is the fundamental feature of Event-B, this argues in favor of our proposed translation.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Design.* Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 6:447-466, 2010.
4. R. J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.
5. J.A. Bergstra, A. Ponse, and S.A. Smolka (Eds.). *Handbook of Process Algebra.*. Elsevier, 2001.
6. M. Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44:850-891, 1997.

7. M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258:99-129, 2001.
8. J. Bryans and W. Wei. Formal Analysis of BPMN Models Using Event-B. In *S. Kowalewski and M. Roveri (Eds.), Proc. FMICS 2010.* LNCS 6371, pp. 33-49, Springer, 2010.
9. M. Butler. Decomposition Structures for Event-B. In *Proc. IFM 2009*, LNCS 5423, pp. 20-38. Springer, 2009.
10. D. Diaconescu, I. Leustean, L. Petre, K. Sere, and G. Stefanescu. *Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems.* TUCS Technical Reports No. 1028, http://tucs.fi. December 2011.
11. C. Dragoi and G. Stefanescu. AGAPIA v0.1: A programming language for interactive systems and its typing systems. In: *Proc. FINCO/ETAPS 2007.* ENTCS 203, pp. 69-94. Elsevier, 2008.
12. C. Dragoi and G. Stefanescu. On compiling structured interactive programs with registers and voices. In *Proc. SOFSEM 2008.* LNCS 4910, pp. 259-270. Springer, 2008.
13. A. S. Fathabadi, A. Rezazadeh, and M. Butler. Applying Atomicity and Model Decomposition to a Space Craft System in Event-B. In: *Proc. 3rd NASA FM Symposium.* LNCS Vol. 6671, pp. 328-342. Springer, 2011.
14. T. S. Hoang, A. Fürst and J.-R. Abrial. Event-B Patterns and Their Tool Support. In: *Proc. SEFM'09*, pp.210-219. IEEE, 2009.
15. A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky. Patterns for Refinement Automation. In: *Proc. FMCO 2009.* LNCS 6286, pp. 70-88. Springer, 2010.
16. A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In: *Proc. ABZ 2010.* LNCS 5977, pp. 174-188. Springer, 2010.
17. M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Refinement-Based Modeling of 3D NoCs. In: *Proc. FSEN'11.* LNCS 7141. Springer, 2011.
18. M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Formal Modeling of Multicast Communication in 3D NoCs. In: *Proc. DSD'11*, pp. 634-642. IEEE, 2011.
19. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.
20. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes I and II. In *Information and Computation*, Vol. 100, No. 1, pp. 1-77, 1992.
21. A. Popa, A. Sofronia and G. Stefanescu. High-level structured interactive programs with registers and voices. *JUCS*, 13:1722-1754, 2007.
22. S. Schneider, H. Treharne, and H. Wehrheim. Bounded Retransmission in Event-B||CSP: a Case Study. In: *ENTSC*, 280: 69-80, 2011.
23. A. Sofronia, A. Popa, and G. Stefanescu. Undecidability Results for Finite Interactive Systems. In: *ROMJIST*, 12:265-279, 2009. Also: Arxiv, CoRR 1001.0143, 2010.
24. G. Stefanescu. Interactive systems with registers and voices. *Fundamenta Informaticae*, 73:285-306, 2006.
25. G. Stefanescu. Towards a Floyd logic for interactive rv-systems. In: *Proc. ICCP 2006*, pp. 169-178. TU Cluj-Napoca, 2006.
26. URL http://www.petrinets.info/
27. URL RODIN tool platform, http://www.event-b.org/platform.html.
28. M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. In: *FMSD*, 13:5-35. Kluwer, 1998.
29. P. Wegner. Interactive foundations of computing. *TCS*, 192:315-351. 1998.