Event-B Decomposition for Parallel Programs [⋆]

Thai Son Hoang and Jean-Raymond Abrial

Deparment of Computer Science,
Swiss Federal Institute of Technology Zurich (ETH-Zurich),
CH-8092, Zurich, Switzerland
htson@inf.ethz.ch, jrabrial@neuf.fr

Abstract. We present here a case study developing a parallel program. The approach that we use combines *refinement* and *decomposition* techniques. This involves in the first step to abstractly specify the aim of the program, then subsequently introduce shared information between sub-processes via refinement. Afterwards, decomposition is applied to split the resulting model into sub-models for different processes. These sub-models are later independently developed using refinement. Our approach aids the understanding of parallel programs and reduces the complexity in their proofs of correctness.
Keywords: Event-B, parallel programs, decomposition, refinement.

1 Introduction

We consider here programs that use several co-operating parallel processes in order to compute the intended final result. Proving correctness of such programs is a difficult task because of the interleaved execution of many sub-statements from different processes. These sub-statements may be executed in an unpredictable order. As a result, techniques such as program testing do not give us sufficient confidence about the correctness of these programs, since no execution leading to an error might appear during tests. To achieve correctness, it is therefore necessary to develop these programs and prove them formally.

There are a number of methods for proving the correctness of parallel programs [11]. Our main contribution is an approach applying the technique of refinement and decomposition in Event-B [2], which reduce the complexity of the verification process (more information in Section 5.1). The approach contains four steps as follows.

1. Start with an abstract specification *in-one-shot* giving the purpose of the program.
2. Refine this abstract specification by introducing details about the *shared variables*.
3. Decompose the model in the previous step to split the model into several (abstract) sub-models for processes.
4. Refine each sub-model from the previous step independently.

In the last step, each sub-model can be seen as a new abstract specification and hence application of steps 2, 3 and 4 can be repeated again. The novelty of our approach is in

⋆ Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (http://www.deploy-project.eu/index.html). We thank Matthias Schmalz, Christoph Sprenger and David Basin for their comments on drafts of this paper.

step 2 where we specify shared information between processes. This information has two purposes. Firstly, it contains the necessary guarantee condition from each process to establish the final result. Secondly, it also gives the condition on which each process can rely on in further development. This decision to have this step early in our development takes advantage of our decomposition technique and results in simpler models and reduces the complexity of proving programs. This is the main advantage of our method over existing approaches. More information on related work is in Section 5.1.

The rest of the paper is structured as follows. Section 2 gives an overview of the Event-B method and the concept of (shared variable) decomposition. Section 3 introduces the *FindP* program and its formal development using our approach is presented in Section 4. Section 5 compares our approach with some existing methods for developing parallel programs and draws some conclusions.

2 The Event-B Modelling Method

A development in Event-B [6] is a set of formal models. The models are built from expressions in a mathematical language, which are stored in a repository. When presenting our models, we will do so in a pretty-printed form e.g., adding keywords and following layout conventions to aid parsing. Event-B has a semantics based on transition systems and simulation between such systems, described in [3]. We will not describe in detail the Event-B semantics here and instead just illustrate some of the proof obligations that are important for our development.

Event-B models are organised in terms of the two basic constructs: *contexts* and *machines*. Contexts specify the static part of a model whereas machines specify the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are similar to types [6]. Axioms constrain carrier sets and constants, whereas theorems express properties derivable from axioms. In the following, we further describe machines and machine refinement.

2.1 Machines

Machines specify behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, *events*, and *variants*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by events. Each event is composed of a *guard* $G(t, v)$ (the conjunction of one or more predicates) and an *action* $S(t, v)$, where t are the *parameters* of the event.[1] The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the term "**any** t **where** $G(t, v)$ **then** $S(t, v)$ **end**". We use the short form "**when** G(v) **then** S(v) **end**" when the event does not have any parameters, and we write "**begin** S(v) **end**" when, in addition, the event's guard equals *true*. A dedicated event of the last form is used for *initialisation*.

[1] When referring to variables v and parameters t, we usually allow for multiple variables and parameters, i.e., they may be "vectors".

The action of an event is composed of one or more *assignments* of the form

$$x \ := \ E(t,v) \tag{1}$$
$$x \ :\in \ E(t,v) \tag{2}$$
$$x \ :\mid \ Q(t,v,x') \quad , \tag{3}$$

where x is a variable contained in v, $E(t,v)$ is an expression, and $Q(t,v,x')$ is a predicate. Assignments of the form (1) are *deterministic*, whereas the other two forms are *nondeterministic*. In (2), x (which must be a single variable) is assigned an element of a set. In (3), Q is a "before-after predicate", which relates the values x (before the action) and x' (afterwards). (3) is the most general form of assignment and nondeterministically selects an after-state x' satisfying Q and assigns it to x. Variables other than x are unchanged by the above assignments. There is also a side condition on the action of an event: the variables on the left-hand side of the assignments contained in the action must be disjoint.

Proof obligations serve to verify certain properties of machines. We only describe the proof obligation for invariant preservation. Formal definitions of all proof obligations are given in [3]. *Invariant preservation* states that invariants hold whenever variables change their values. Obviously, this does not hold a priori for any combination of events and invariants and therefore must be proved. For each event and each invariant, we must prove that the invariant is *re-established* after the event is carried out. More precisely, under the assumption of the invariants and the event's guard, we must prove that the invariant still holds in any possible state after the event's execution.

Similar proof obligations are associated with a machine's initialisation event. The only difference is that there is no assumption that the invariant holds. For brevity, we do not treat initialisation differently from other events. The required modifications of the associated proof obligations are straightforward.

2.2 Machine Refinement

Machine refinement provides a means to introduce details about the dynamic properties of a model [6]. For more details on the theory of refinement, we refer to the Action System formalism [7], which has inspired the development of Event-B. Here we sketch some central proof obligations for machine refinement.

A machine CM can refine another machine AM. We call AM the *abstract* machine and CM the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing invariant* $J(v,w)$, where v are the variables of the abstract machine and w are the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Let the abstract event ea and concrete event ec be:

$$\mathsf{ea} \ \widehat{=} \ \textbf{any}\ t\ \textbf{where}\ G(t,v)\ \textbf{then}\ S(t,v)\ \textbf{end} \tag{4}$$
$$\mathsf{ec} \ \widehat{=} \ \textbf{any}\ u\ \textbf{where}\ H(u,w)\ \textbf{then}\ T(u,w)\ \textbf{end} \quad . \tag{5}$$

Somewhat simplified, we say that ec refines ea if the guard of ec is stronger than the guard of ea (*guard strengthening*), and the gluing invariant $J(v,w)$ establishes a simulation of ec by ea (*simulation*). Proving guard strengthening just amounts to proving an implication. For simulation, we must prove that ec can be "simulated" by ea. More precisely, under the assumption of the invariants and of the concrete guard $H(u, w)$ we must show that it is possible to choose a value for the abstract parameter t such that the abstract guard holds and the gluing invariant $J(v, w)$ is re-established. The possible values for the abstract parameter are given as *witness* in ec with the keyword **with**.

In the course of refinement, *new events* are often introduced into a model. New events must be proved to refine the implicit abstract event SKIP, which does nothing. Moreover, it may be proved that new events do not collectively diverge. In other words, the new events cannot take control forever and hence one of the old events eventually occurs. We will not go into details for *convergent* proof obligation in this paper.

We have used the *Rodin Tool* [4] for our formal development. This is an industrial-strength tool for creating and analysing Event-B models. It includes a proof-obligation generator and support for interactive and semi-automated theorem proving.

2.3 Shared Variable Decomposition

The idea of decomposition is to split a large model into smaller sub-models which can be handled more comfortably than the whole: one should be able to refine these sub-models independently [2]. More precisely, if one starts from an initial (large) model, say M, decomposition allows us to split this model into several sub-models $\mathsf{M}_1 \cdots \mathsf{M}_i$. These sub-models can then be refined independently yielding $\mathsf{N}_1 \cdots \mathsf{N}_i$. The correctness of the decomposition technique guarantees that the model N, obtained by re-composing $\mathsf{N}_1 \cdots \mathsf{N}_i$, is a refinement of the original model M. This process is illustrated in the following diagram:

$$\begin{array}{|ccc|} \hline \text{Decomposition} & \text{Refinement} & \text{Re-composition} \\ \mathsf{M} \quad \rightarrow & \left\{ \begin{array}{ccc} \mathsf{M}_1 & \rightarrow & \mathsf{N}_1 \\ & \cdots & \\ \mathsf{M}_i & \rightarrow & \mathsf{N}_i \end{array} \right\} & \rightarrow \quad \mathsf{N} \\ \hline \end{array}$$

Generation of sub-models using shared variable decomposition Given a model M with events $\mathsf{e}_1(\mathsf{a})$, $\mathsf{e}_2(\mathsf{a}, \mathsf{c})$, $\mathsf{e}_3(\mathsf{b}, \mathsf{c})$, $\mathsf{e}_4(\mathsf{b})$,[2] we would like to decompose M into two separate models: M_1 dealing with events e_1 and e_2; and M_2 dealing with events e_3 and e_4.

By giving the above *event partition*, we must also perform a *variable distribution*. This distribution can be derived directly from the information about the partitioning of events and the set of variables that they access. In our example, M_1 must have variables a and c, while M_2 must have variables b and c. As a result, c becomes a *shared variable* between the two models which *cannot be data-refined*. In contrast, the variables a and b are private variables of M_1 and M_2 and can be data-refined by their corresponding sub-refinements.

Moreover, in each sub-model, we need to have a number of *external events* to simulate how shared variables are handled in the non-decomposed model. These events are

[2] Note that the variables appeared in brackets denote those that are *accessed* by these events, e.g. appearing in guard or action of the corresponding event.

abstract versions of the corresponding internal events and use only the shared variables. In our example, M_1 will have an external event corresponding to e_3 (beside the internal events e_1 and e_2). Symmetrically, M_2 will have an external event corresponding to e_2. Similar to shared variables, *external events* cannot be further refined.

We also present a practical construction of the external event given its original event. This is illustrated below for an external event (ext_)e_2 in sub-model M_2. Intuitively, this event is the *projection* of the original event, i.e. e_2, on the state of the sub-model M_2.

```
e₂
  any  $t$  where
    $G(t, a, c)$
  then
    $a, c :| Q(t, a, c, a', c')$
  end
```

```
(ext_)e₂
  any  $t, a$  where
    $G(t, a, c)$
  then
    $c :| \exists a' \cdot Q(t, a, c, a', c')$
  end
```

3 Example: FindP Program

Our running example is a standard problem in the literature for parallel programs. The purpose of the *FindP* program is to find the first index k of an array $ARRAY$ that satisfies some property P, if there is one. If this index does not exist, i.e. none of the array elements satisfy P, the program returns $M + 1$, where M is the size of the array.

We are interested in the solution using two parallel processes to independently investigate the array that was given by Rosen [20]. The processes in the original program works on the sets of even and odd indices separately. We present here a slightly generalised version of it where the two processes work on any two different parts of the array, denoted as $PART1$ and $PART2$, which cover the entire domain of the array, and are not necessarily disjoint.

The main idea of each process is to independently evaluate the value of the array in ascending order and to publish the first value that it finds. Moreover, from time to time, a process looks at the value that is published by the other process in order to know if it needs to continue the search or if it can terminate early.

The pseudo-code for the main program is given below. Here $index1$, $index2$ are the two local indices, and $publish1$, $publish2$ are the published results of the processes. In the end, when both processes terminate, the result taken is the minimum of the two published results.

```
$index1, index2 := min(PART1), min(PART2);$
$publish1, publish2 := M + 1, M + 1;$
$\mathbf{process_1} \;||\; \mathbf{process_2};$
$result := min(\{publish1, publish2\})$
```

The pseudo-code for each process (presented here $process_1$) is as follows. Each process needs to continue only if its local index is smaller than both published results (as indicated by the guard of the loop). If this is the case, the process evaluates the value of the array at the current index and performs appropriate actions: publishing the current index or moving to the next index, if possible.

```
while $index1 < min(\{publish1, publish2\})$ do
  if $ARRAY(index1) = \text{TRUE}$ then $publish1 := index1$
  else $index1 :=$ the-next-index-in-PART1-or-M+1 end
end
```

The key interaction between the two processes appears in the guard of the loop. Here the guard of $process_1$ refers to the published result of $process_2$, which in the meantime could be modified. In other words, $process_1$ needs to read the published value of $process_2$ into some local variable before making the comparison using this local variable. The unfolded version of the $process_1$ is as follows. Our formal development in later sections is guided towards this version of the processes.

```
1 : $(read)$         $read1 := publish2;$
2 :                  **if** $index1 < min(\{publish1, read1\})$ **then**
                        **if** $ARRAY(index1) = \mathrm{TRUE}$ **then**
$(found)$                  $publish1 := index1;$  $goto\ 3;$
                        **else**
$(inc)$                    $index1 :=$ the-next-index-in-PART1-or-M+1;  $goto\ 1;$
                        **end**
                     **else**
$(not\_found)$          $goto\ 3$
                     **end**
3 : $(end)$
```

Here we make some assumptions on the atomicity. They are similar to the atomicity assumptions made by Abrial/Cansell [5].

- We have a number of shared variables (e.g. the published values). They are the variables that are written by one process and read by the other process. They are the shared variable with respect to the *read* process.
- We have a number of local variables (e.g. the local indices).
- The events involving only local tests and actions can be performed concurrently.
- There is an elementary atomic action for reading the value of a shared variable into a local variables, e.g. $local\_variable := shared\_variable$.
- We extend the above atomic action to contain possible local test and local action.

```
**when** $local\_test$ **then**
   $local\_variable := shared\_variable$
   $local\_action$
**end**
```

Different atomicity assumptions will lead to different *unfolded* versions of our program here. But this will not effect the applicability of our approach.

4 Formal Development

The machine-checked version of the development can be found on the web[3]. We first present our strategy for developing this program as follows.

Initial model specifies the result of the algorithm directly.
First refinement introduces the local indices of processes.
Decomposition step splits the model into sub-models corresponding to different processes: $main$, $process_1$, $process_2$.

We continue with further refinement steps for $process_1$ ($process_2$ should be developed in symmetrical fashion). Futher development of the $main$ process is straightforward and is not of our interest here.

First sub-refinement introduces the local index of the process.
Second sub-refinement introduces the read value of the process.
Third sub-refinement introduces the address counter for scheduling of events.

[3] URL: http://deploy-eprints.ecs.soton.ac.uk/154/

4.1 Initial Context and Model

The context defines an array of Booleans representing our abstract view.

constants: $ARRAY, M$

axioms:
axm0_1 : $M \in \mathbb{N}_1$
axm0_2 : $ARRAY \in 1\,..\,M \to BOOL$

The initial model contains only one integer variable called $result$. There is only one event final (beside the initialisation) to specify the result of the program *in-one-shot*. The aim of the program is encoded in the guard as constraints for parameter k.

```
final
  any  $k$  where
    $k \in 1\,..\,M+1$
    $\forall j \cdot j \in 1\,..\,k-1 \Rightarrow ARRAY(j) = \mathrm{FALSE}$
    $k \neq M+1 \Rightarrow ARRAY(k) = \mathrm{TRUE}$
  then
    $result := k$
  end
```

4.2 First Refinement

The first refinement introduces the idea of using two processes. Here the context needs to be extended to include the notion of two different non-empty parts of the array.

constants: $PART1, PART2$

axioms:
axm1_1 : $PART1 \cup PART2 = 1\,..\,M$
axm1_2 : $PART1 \neq \varnothing \ \wedge\ PART2 \neq \varnothing$

At this point, the necessary information about the two sub-processes in order to obtain the final result of the program is whether or not they already terminate and the published results of the two processes. They are represented by a pair of variables, namely *finish1* and *publish1* (respectively *finish2* and *publish2*) for $process_1$ (respectively $process_2$). Initially *finish1* (respectively *finish2*) is given the value FALSE, i.e. the process has not yet terminated; and *publish1* (respectively *publish2*) is assigned the value $M+1$, i.e. the process has not yet found a result.

We first look at the refinement of the final event with the new set of variables. This event is carried out when the two processes have finished and the result taken is just the minimum of the two published values.

```
final
  refines  $final$
  when
    $finish1 = \mathrm{TRUE} \ \wedge\ finish2 = \mathrm{TRUE}$
  with
    $k = min(\{publish1, publish2\}$
  then
    $result := min(\{publish1, publish2\})$
  end
```

In order to prove the refinement of the final event with respect to its abstract version, we need to give a witness for the disappearing parameter k of the abstraction. Here the parameter k is exactly the minimum of the two published values. Given the witness, the *simulation* proof obligation becomes trivial since both the abstract and concrete events assign equivalent expressions to the variable $result$.

We still need to prove *guard strengthening*. This requires us to give some invariants for the newly introduced variables. The invariants are symmetric for $process_1$ and $process_2$, hence we only give the five invariants associated with $process_1$ here.

```
invariants:
  inv1_1  $publish1 \neq M + 1 \Rightarrow finish1 = \text{TRUE}$
  inv1_2  $publish1 \neq M + 1 \Rightarrow publish1 \in PART1$
  inv1_3  $publish1 \neq M + 1 \Rightarrow ARRAY(publish1) = \text{TRUE}$
  inv1_4  $publish1 \neq M + 1 \Rightarrow (\forall i \cdot i \in PART1 \land i < publish1 \Rightarrow ARRAY(i) = \text{FALSE})$
  inv1_5  $finish1 = \text{TRUE} \land publish1 = M + 1 \Rightarrow$
            $(\forall i \cdot i \in PART1 \land i < publish2 \Rightarrow ARRAY(i) = \text{FALSE})$
```

inv1_1 states that if $process_1$ has published some result then it must have terminated. This also means the process can publish at most once.

inv1_2–inv1_4 states that $process_1$ *cannot lie*: if it publishes some result then this must be the smallest index that it can find within $PART1$.

inv1_5 states that in the case where $process_1$ terminates without publishing any values, it has given up because it cannot find any better result than the other process $process_2$. The two possibilities for $process_1$ to terminate are:
- it has searched all the indices in $PART1$ and did not find any result, or
- it looks at the published value of the $process_2$ and knows that it cannot find a better (smaller) result.

In both situations, the invariant holds trivially.

We now abstractly construct the events to model the effect of the two processes on the new variables. These events correspond to the two cases in which a process can terminate. Here, we consider the events corresponding to $process_1$ only.

The first case is when $process_1$ finds a result within $PART1$ and terminates. Here $\texttt{publish1} = \texttt{M} + 1$ is a theorem, which is the consequence of the first guard $finish1 = FALSE$ and invariant **inv1_1**. The other case is when $process_1$ terminates without publishing any value.

```
found_1
  any  $k$  where
    $finish1 = \text{FALSE}$
    $k \in PART1$
    $ARRAY(k) = \text{TRUE}$
    $\forall i \cdot i \in PART1 \land i < k \Rightarrow ARRAY(i) = \text{FALSE}$
    $\texttt{publish1} = \texttt{M} + 1$
  then
    $finish1, publish1 := \text{TRUE}, k$
  end
```

```
not_found_1
  when
    $finish1 = \text{FALSE}$
    $\forall i \cdot i \in PART1 \land i < publish2 \Rightarrow$
        $ARRAY(i) = \text{FALSE}$
  then
    $finish1 := \text{TRUE}$
  end
```

4.3 Decomposition

In the previous refinement step, we introduced the *interface* of the processes, i.e. the shared variables and events describing how these variables can be changed, which guarantees the correctness of the program. At this point, we want to develop in details each process independently. We apply the technique of decomposition (shared variable) as described earlier in Section 2.3. There will be three different processes: $main$ (final), $process_1$ (found1, not_found1) and $process_2$ (found2, not_found2).

As a result, we have three different sub-models, one for each process. Amongst these sub-models, the development $main$ is straightforward and is not of our interest here. We concentrate on the sub-model for $process_1$ ($process_2$ is symmetric).

The sub-model for $process_1$ contains three shared variables: *finish1*, *publish1* and *publish2* and no private variables. This process does not refer to either *result* (the global result) or *finish2* (if the other process has finish or not). According to the event distribution, this model of $process_1$ has two internal events, namely found_1 and not_found_1, which are the exact copy of the original events. The other events become external which need to be generated as follows. We present the original events on the left and the corresponding external events for $process_1$ on the right.

```
final
  when
    $finish1 = \mathrm{TRUE}$
    $finish2 = \mathrm{TRUE}$
  then
    $result := min(\{publish1, publish2\})$
  end
```

```
(ext_)final
  any  $finish2$  where
    $finish1 = \mathrm{TRUE}$
    $finish2 = \mathrm{TRUE}$
  then
    SKIP
  end
```

```
found_2
  any  $k$  where
    $finish2 = \mathrm{FALSE}$
    $k \in PART2$
    $ARRAY(k) = \mathrm{TRUE}$
    $\forall i \cdot i \in PART2 \land i < k \Rightarrow$
      $ARRAY(i) = \mathrm{FALSE}$
    $\mathtt{publish2} = \mathtt{M} + 1$
  then
    $finish2, publish2 := \mathrm{TRUE}, k$
  end
```

```
(ext_)found_2
  any  $k, finish2$  where
    $finish2 = \mathrm{FALSE}$
    $k \in PART2$
    $ARRAY(k) = \mathrm{TRUE}$
    $\forall i \cdot i \in PART2 \land i < k \Rightarrow$
      $ARRAY(i) = \mathrm{FALSE}$
    $\mathtt{publish2} = \mathtt{M} + 1$
  then
    $publish2 := k$
  end
```

```
not_found_2
  when
    $finish2 = \mathrm{FALSE}$
    $\forall i \cdot i \in PART1 \land i < publish2 \Rightarrow$
      $ARRAY(i) = \mathrm{FALSE}$
  then
    $finish2 = \mathrm{TRUE}$
  end
```

```
(ext_)not_found_2
  any  $finish2$  where
    $finish2 = \mathrm{FALSE}$
    $\forall i \cdot i \in PART1 \land i < publish2 \Rightarrow$
      $ARRAY(i) = \mathrm{FALSE}$
  then
    SKIP
  end
```

4.4 Further (sub-)refinements

In this section, we present the sketch of the further development of $process_1$. The refinement steps are all typical super-position refinement where more details about the actual process are introduce at each step as mention early in the start of Section 4. We do not present in detail the proofs of the correctness of the refinement steps here.

Introducing the local index In the first sub-refinement for $process_1$, we introduce the index that the process is currently checking. This is represented by the new variable *index1*. The following invariants state that this process investigates only the part of the array belongs to $PART1$ in ascending order and it cannot skip any index.

invariants:
inv2_1 $index1 \neq M + 1 \Rightarrow index1 \in PART1$
inv2_2 $\forall k \cdot k \in PART1 \land k < index1 \Rightarrow ARRAY(k) = \mathrm{FALSE}$

The internal event not_found_1 is unchanged. It trivially maintains the new invariants since it only modifies variable *finish1*. The same applies to external events, i.e. (ext_)final, (ext_)found_2, (ext_)not_found_2 (which are always unchanged during refinement), since they do not refer to variable $index1$.

We now refine the internal event found_1 to use $index1$: We also introduce a new event inc_1 to model the case where the value at the current index is FALSE and hence $process_1$ moves to the next index.

```
found_1
  refines  $found\_1$
  when
    $finish1 = \text{FALSE}$
    $index1 \neq M + 1$
    $ARRAY(index1) = \text{TRUE}$
  with
    $k = index1$
  then
    $finish1, publish1 := \text{TRUE}, index1$
  end
```

```
inc_1
  any  $i$  where
    $ARRAY(index1) = \text{FALSE}$
    $i \neq M + 1 \Rightarrow i \in PART1$
    $index1 < i$
    $\forall j \cdot j \in PART1 \land index1 < j \Rightarrow i \leq j$
  then
    $index1 := i$
  end
```

For event found_1, the information from the witness $k = index1$ and the two invariants declared above guarantees that this is a correct refinement of the abstract event. For event inc_1, the parameter i is the smallest index in $PART1$ that is greater than $index1$, or $M + 1$ if such an index does not exist. The proof that this event maintains the invariants is intuitive and can be found in our technical report [12].

Introduce the read value In this refinement, we introduce the read value of process represented by variable *read1*. The constraint for this variable is expressed by invariant **inv3_1**: its value is either $M + 1$ or the published value of the other process, i.e. $publish2$. A new event read_1 is introduced to model the situation when $process_1$ reads the published value of $process_2$. This event sets the value of *read1* to *publish2* and hence clearly maintains the invariant **inv3_1**.

```
invariants:
  inv3_1  $read1 \neq M + 1 \Rightarrow read1 = publish2$
```

```
read1
  begin
    $read1 := publish2$
  end
```

The only change to event inc1 is two extra guards: $index1 < read1$ and $index1 < publish1$. Since this event does not change variables *read1* and *publish2*, it preserves the invariant **inv3_1** trivially.

The event found_1 is refined by replacing the guard $index1 \neq M + 1$ with the following two guards: $index1 < read1$ and $index1 < publish1$. Since both $publish1$ is either $M + 1$ or belongs to $PART1$, $publish1$ is no greater than $M + 1$. Together with the guard $index1 < publish1$, $index1$ is strictly smaller than $M + 1$. Hence the proof obligation for guard strengthening holds trivially.

We refine the remaining internal event not_found_1 by replacing the guard $\forall i \cdot i \in PART1 \land i < publish2 \Rightarrow ARRAY(i) = \text{FALSE}$ with $index1 < read1 \Rightarrow publish1 \neq M + 1$. We do not go into detail of the proof why this is a correct guard strengthening, but refer the readers to our technical report [12]

For the external events, even though they are not refined, we must prove that they maintain the invariant **inv3_1**. In this case, we must consider those events that modify variable *publish2*. In our development, this is event (ext_)found_2. The important part for our proof in this event is the theorem in the guard, i.e. `publish2` $=$ `M` $+$ `1`, and the action $publish2 := k$. According to the action, we have to prove that $read1 \neq M+1 \Rightarrow read1 = k$, under the assumption of the invariants and the guards. From the theorem in guard $publish2 = M + 1$ and invariant **inv3_1**, we have $read1 = M + 1$ (since if it is not, then we have $publish2 = read1 \neq M + 1$). Hence $read1 \neq M + 1 \Rightarrow read1 = k$ holds trivially.

Introduce the address counter In this last sub-refinement of $process_1$ we introduce the address counter in order to obtain the unfolded program as described in Section 3. The resulting internal events (with some refinement for guards) are as follows. These events conform with the notion of atomicity mentioned earlier.

```
read1
  when
    $address1 = 1$
  then
    $address1, read1 := 2, publish2$
  end
```

```
not_found_1
  when
    $address1 = 2$
    $\neg(index1 < min(\{publish1, read1\}))$
  then
    $address1, finish1 := 3, \mathrm{TRUE}$
  end
```

```
found_1
  when
    $address1 = 2$
    $index1 < min(\{publish1, read1\})$
    $ARRAY(index1) = \mathrm{TRUE}$
  then
    $address1 := 3$
    $finish1 := \mathrm{TRUE}$
    $publish1 := index1$
  end
```

```
inc_1
  any  $i$  where
    $address1 = 2$
    $index1 < min(\{publish1, read1\})$
    $ARRAY(index1) = \mathrm{FALSE}$
    $i \neq M + 1 \Rightarrow i \in PART1$
    $index1 < i$
    $\forall j \cdot j \in PART1 \land index1 < j \Rightarrow i \leq j$
  then
    $address1, index1 := 1, i$
  end
```

4.5 Proof Statistics

The proof statistics for the development is in the table below. We only take into account the number of obligations for sub-refinement models once, since the refinements for both process $process_1$ and $process_2$ are symmetric. We can use techniques such as pattern or generic instantiation in order to reuse the sub-development without reproving again. In the table, 50% of the proof obligations are in the model before decomposing. This indicates that this refinement is the most important and difficult step in our approach.

Model	Number POs	Auto.(%)	Manual (%)
Initial context	0	0 (N/A)	0 (N/A)
Initial model	3	3 (100%)	0 (0%)
First extended context	0	0 (N/A)	0 (N/A)
First refinement	46	44 (96%)	2 (4%)
First sub-refinement	14	10 (71%)	4 (29%)
Second sub-refinement	6	5 (83%)	1 (17%)
Third sub-refinement	22	16 (73%)	6 (27%)
Total	91	78 (86%)	13 (14%)

5 Related Work and Conclusion

5.1 Related Work

The problem of verifying the *FindP* program has been tackled using different methods, notably using Owicki/Gries' *interference-free* [19] and Jones' *rely/guarantee* approach [14,15]. Moreover, the *FindP* program has been used as an illustrated example for the formalisation of these two approaches in Isabelle/HOL [18].

The work of Owicki/Gries [19] extends Hoare's deductive system for sequential programs [13] in order to prove the correctness of parallel programs. Their proofs of correctness for parallel statements centre around the notion of *interference-free* which is defined as follows. Given a proof of Hoare's triple $\{P\}\ S\ \{Q\}$ and a statement T with precondition $pre(T)$, T does not interfere with $\{P\}\ S\ \{Q\}$ if

InfFree1 $\{Q \wedge pre(T)\}\ T\ \{Q\}$, i.e. T maintains the post-condition Q, and
InfFree2 for any sub-statement S' of S, $\{pre(S') \wedge pre(T)\}\ T\ \{pre(S')\}$.

Within our approach, the above two conditions are verified during the development of the model at various refinement levels. At the abstract level before decomposition, S and T are some events of the models and the post-condition Q are just some invariants. For example, S are events belonging to $process_1$, T are events belonging to $process_2$, and Q are the invariants that state the outcome of $process_1$, e.g. **inv1_1–inv1_5**. We have to prove that these invariants are maintained by any events T and this corresponds to condition **InfFree1**.

Furthermore, during the sub-refinement of a process, sub-statements S' of S are introduced. At the same time, new invariants are added and these invariants correspond to the preconditions $pre(S')$ in the proof of $\{P\}\ S\ \{Q\}$ using Hoare's deductive system. Hence the condition **InfFree2** is verified by proving that events T (now external events) maintain the new invariants.

This is not too surprising, since in our approach, the role of external events is to keep track of the information about the possible changes on shared variables by different processes. During the refinement of a sub-process, we need to take into account the effect of these external events so that they do not "interfere" with the development of this sub-process. The main advantage of our approach over the work from Owicki/Gries is that these external events are at the abstract level rather than concrete statements as defined in the *interference-free* conditions. This reduces the complexity of the verification process.

Compared to the Owicki/Gries approach, our method is closer to the *rely/guarantee* approach of Jones [14]. The approach extends the notion of Hoare's triple $\{P\}\ S\ \{Q\}$ to encode the rely condition R and guarantee condition G. By definition, a condition $\{P, R\}\ S\ \{G, Q\}$ is satisfied by S if: under the assumptions that S starts in state satisfies the precondition P, and any external transition satisfies the rely condition R; then S ensures that any internal transition of S satisfies the guarantee condition G, and if S terminates then the final state satisfies postcondition Q.

We focus on an example rule for parallel composition.

$$\textbf{PAR-I}\quad \frac{\begin{array}{ll} R \vee G_1 \Rightarrow R_2 & (\textbf{RG1}) \\ R \vee G_2 \Rightarrow R_1 & (\textbf{RG2}) \\ G_1 \vee G_2 \Rightarrow G & (\textbf{RG3}) \\ \{P, R_1\} S_1 \{G_1, Q_1\} & (\textbf{RG4}) \\ \{P, R_2\} S_2 \{G_2, Q_2\} & (\textbf{RG5}) \end{array}}{\{P, R\}\ S_1 \mid\mid S_2\ \{G, Q_1 \wedge Q_2\}}$$

The rule is interpreted as follows. Statement $S_1 \mid\mid S_2$ satisfies $\{P, R\}\ S_1 \mid\mid S_2\ \{G, Q_1 \wedge Q_2\}$ if the following conditions are met. Firstly, both "global" rely condition R and the guarantee condition of one statement ensure the rely condition of the other (**RG1** and **RG2**). Secondly, both guarantee conditions of the two statements ensure the global guarantee condition G (**RG3**). Lastly, S_1 and S_2 independently satisfy their corresponding rely/guarantee condition (**RG4** and **RG5**)

Note that both rely and guarantee conditions are relations over two states. They are indeed similar to events in Event-B which correspond to a relations over pre-/post-states. Moreover, the implication between rely/guarantee conditions is the same as event refinement. Within our approach, a pair of internal/external events encodes rely/guarantee conditions where the rely condition corresponds to the external event and the guarantee condition corresponds to the internal event. The generation of external events guarantees that they are the abstractions of the corresponding internal events. In fact, our generation of sub-models as described in Section 2.3 guarantees that the resulting sub-models satisfy the parallel composition rule. This is the advantage of our approach over the *rely/guarantee* method. In fact the external events are the strongest possible condition that the other process can rely on. In practise, the rely/guarantee conditions could be more abstract, e.g. requires only that the value of some variables decrease monotonically [16]. Moreover, rely/guarantee is usually used for composition rather than decomposition as in [1].

The decomposition technique also appears in many other approaches, with similar intuition: Breaking a specification into smaller pieces and reasoning about them independently. For example, in the work of Abadi/Lamport [1], this is captured by their *Decomposition Theorem* and a generalised version of it. The most important idea in their approach is to find some properties E (also called *environment*) of the other processes assumed by a process. However, in another study, Lamport claimed that decomposition might not be that useful [17]. One of the argument is the difficulty in inventing the *environment* properties and checking the hypotheses of the decomposition theorem. In our approach, we *derive* these properties from the overall purpose of the program using refinement (step 2 of our approach). This is also the reason why we consider the class of parallel programs that achieve some intended result.

Stepwise refinement has been considered for developing parallel systems in Action System in early work of Back/Sere [8,9]. The shared variable decomposition in Event-B corresponds to their notion of *concurrent action system* (in contrast to *distributed action system* with shared actions). However, the approach presented in [8] based on the notion of refining atomicity introduces the notion of parallelism quite late in the development (almost as the last step of the refinement chain). The reason for this delay is that the decision for implementing the system as concurrent action system or distributed action system can be made as late as possible. In our example, we have this decision of using shared variables in advance. Hence we can take the advantage of having the decomposition early to reduce the complexity. We consider the use of shared variables as a part of the design process of the program rather than an implementation detail.

5.2 Conclusion

We have presented a method for developing parallel programs using refinement and decomposition techniques. Refinement gives us the possibility to abstractly define the aim of the programs which helps us to understand the purpose of these programs. Decomposition allows us to reduce the complexity of the development by separately developing sub-processes while keeping track of minimum information on what other processes can do. Our approach should be applicable to all programs that use several parallel processes in order to obtain a certain goal.

Our approach introduces the possible *interaction* between processes early in the development in order to take the advantage of decomposition. This is different from the approach where one develops processes according to the implementation of the process with possible *cheating* (e.g. one process directly looks into the value of the other process), and subsequently refines the model until there is no more cheating. This approach has been proposed in [3] and is used in many other examples. Applying this approach without using decomposition, the two processes are developed together, and hence the development also has higher complexity comparing to our approach.

The key aspect of our development using decomposition lies in the model that is being decomposed, where we have to abstractly specify the effect of the two future processes on shared variables. We use the overall intended result of the program to help us to *derive* the requirement on the future processes. Furthermore, as a result of using step-wise refinement, we can develop sub-processes using different implementations as long as they satisfy the abstraction. As an example, we can also "implement" the two processes (inefficiently) by not checking the published values of the other processes or having more fine-grained version of atomicity.

For future work, we would like to apply our method to other standard parallel programs (not necessarily ones with intended final result) known from literature, such as "bounded buffer", "partition of set" or "bubble-lattice sort", which have been studied using other approaches [10]. Our approach should not only be used for verification a posteriori but also for finding proofs of correctness for such systems.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 1995.
2. J-R. Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.
3. J-R. Abrial. *Modeling in Event-B: System and Software Design*. CUP, 2009. To appear.
4. J-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *ICFEM 2006*, 2006.
5. J-R. Abrial and D. Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *J. UCS*, 2005.
6. J-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica*, 2006.
7. R-J. Back. Refinement calculus, part II: Parallel and reactive programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX Workshop*, pages 67–93, 1989.
8. R-J. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.*, 1989.
9. R-J. Back and K. Sere. Superposition refinement of parallel algorithms. In *FORTE*, 1991.
10. H. Barringer. *A Survey of Verification Techniques for Parallel Programs*. LNCS. Springer, 1985.
11. W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. CUP, 2001.
12. T.S. Hoang. Event-B development of the FindP program. Technical Report 653, ETH Zurich, November 2009.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 1969.
14. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.
15. C.B. Jones. The role of proof obligations in software design. In *TAPSOFT, V.2*, LNCS, 1985.
16. C.B. Jones. Splitting atoms safely. *Theor. Comput. Sci.*, 2007.
17. L. Lamport. Composition: A way to make proofs harder. In *COMPOS*, 1997.
18. L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2001.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 1976.
20. B. K. Rosen. Correctness of parallel programs: The Church-Rosser approach. *Theor. Comput. Sci.*, 1976.