Event-B Decomposition for Parallel Programs
— *Extended Abstract* —[*]

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 an approach for developing a parallel program combining *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 separate 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

There are a number of methods for proving the correctness of parallel programs [4]. Our main contribution is an approach applying the technique of refinement and decomposition in Event-B [1]. The approach contains four steps as follows.

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

In the last step, each sub-model can be seen as a new abstract specification, hence application of steps 2, 3 and 4 can be repeated again. The novelty of our approach is in step 2 where we specify shared information between processes. This information has dual purpose. Firstly, it contains the necessary guarantee condition from each process to establish the final result. Secondly, it also gives the condition for which each process can rely on in further development. This decision, i.e. to have this step early in our development, takes advantage of 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 4.

[*] Part of this research was carried out within the European Commission ICT project 214158 DEPLOY (http://www.deploy-project.eu/index.html).

2 The Event-B Modelling Method

A development in Event-B [3] is a set of formal models. Event-B has a semantics based on transition systems and simulation between such systems, described in [2]. We will not describe in detail, just high-lights some important points for Event-B semantics.

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*.

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. *Proof obligations* serve to verify certain properties of machines. Given a machine, we need to prove the following obligations:

- **Invariant preservation**: invariants hold whenever variables change their values.
- **Feasibility**: For an event the action is feasible whenever the guard is enable.

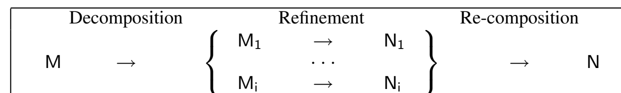
Machine refinement provides a means to introduce details about the dynamic properties of a model [3]. A machine CM can refine another machine AM. We call AM the *abstract* machine and CM the *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *gluing 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. 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*).

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 can occur.

2.1 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. More precisely, if one starts from an initial (large) model, say M, decomposition allows us to separate 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:



Generation of sub-models using shared variable decomposition Given a certain model M with events $\mathsf{e}_1(\mathsf{a})$, $\mathsf{e}_2(\mathsf{a},\mathsf{c})$, $\mathsf{e}_3(\mathsf{b},\mathsf{c})$, $\mathsf{e}_4(\mathsf{b})$,[1] 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 certain *variable distribution*. This distribution can be derived directly from the information about the partitioning of events and the set of variables that they access.

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 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 $(\mathsf{ext_})\mathsf{e}_2$ 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 :| ∃a'·Q(t, a, c, a', c')
  end
```

More detail on shared variable decomposition in Event-B can be found in [1].

3 Example. The 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 $ARRAY$, if there is one, satisfies some property P. Otherwise, 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.

The pseudo-code for the main program is given below (on the left) and for each process (presented here $process_1$ on the right)

```
index1, index2 := min(PART1), min(PART2);
publish1, publish2 := M + 1, M + 1;
process1 || process2;
result := min({publish1, publish2})
```

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

The machine-checked version of the development can be found on the web [5]. We summarise 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.

[1] Note that the variables appeared in brackets denote those that are *accessed* by these events.

4 Related Work

The problem of verifying the *FindP* program has been tackled using different methods, e.g., Owicki/Gries' *interference-free* [9] and Jones' *rely/guarantee* approach [7].

The work of Owicki/Gries [9] extends Hoare's deductive system for sequential programs [6] 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 decomposing, S and T are some events of the models and the post-condition Q are just some invariants. For example, S are some events belonging to $process_1$ and T are events belonging to $process_2$, 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 becoming external events) maintain the new invariants.

This is somewhat not surprising, since in our approach, the role of external events is to keep 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.

Comparing to the Owicki/Gries approach, our method is closer to the *rely/guarantee* approach of Jones [7]. 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 2.1 guarantees that the resulting sub-models satisfy the parallel composition rule. This is the advantage of our approach over *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 that values of some variables decrease monotonically [8].

5 Conclusion

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 [2] and is used in many other examples. Applying this approach without using decomposition, the two processes are developed together, hence the development also has higher complexity comparing to our approach.

The key point in 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.

References

1. J-R. Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.
2. J-R. Abrial. *Modeling in Event-B: System and Software Design*. CUP, 2009. To appear.
3. J-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamentae Informatica*, 2006.
4. 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.
5. T.S. Hoang. FindP development using decomposition. http://deploy-eprints.ecs.soton.ac.uk/154/, 2009.
6. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 1969.
7. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 1983.
8. C.B. Jones. Splitting atoms safely. *Theor. Comput. Sci.*, 2007.
9. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 1976.