SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **00**:1–7 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Decomposition Tool for Event-B

Renato Silva[1], Carine Pascal[2], Thai Son Hoang[3], Michael Butler[1]

[1] *University of Southampton, UK*
[2] *Systerel, Aix-en-Provence, France*
[3] *ETH Zurich, Zurich, Switzerland*

SUMMARY

Two methods have been identified for Event-B model decomposition: shared variable and shared event. The purpose of this paper is to introduce the two approaches and the respective tool support in the Rodin platform. Besides alleviating the complexity for large systems and respective proofs, decomposition allows team development in parallel over the same Event-B project which is very attractive in the industrial environment.

KEY WORDS: Event-B, Decomposition, Shared Event, Shared Variable, Formal Methods

Introduction

The "top-down" style of development used in Event-B [2] allows the introduction of new events and data-refinement of variables during refinement steps. A consequence of this development style is an increasing complexity of the refinement process when dealing with many events and state variables. The main purpose of the model decomposition is precisely to address such difficulty by cutting a large model into smaller components. Two methods have been identified for the Event-B decomposition: shared variable [3, 1] and shared event [11, 12]. We propose a plug-in developed for the Rodin platform [22] supporting these two methods for Event-B. Since decomposition is monotonic [11], the generated sub-components can be further refined independently. Therefore we can introduce team developments: several developers share parts of the same model and work independently in parallel. Moreover decomposition also partitions proof obligations (POs) which are expected to be discharged more easily in the sub-components. Next we introduce briefly the formalism used during the description of our work: Event-B.

Contract/grant sponsor: R. Silva is sponsored by a Fundação Ciência e Tecnologia (FCT-Portugal) scholarship.
Contract/grant sponsor: Part of this research was carried out within the European Commission ICT DEPLOY (http://www.deploy-project.eu); contract/grant number: 214158

Event-B Language

Event-B is a formal methodology that uses mathematical techniques based on set theory and first order logic supporting system development with abstract specification. An abstract Event-B specification is divided into a static part called *context* and a dynamic part called *machine*. A machine *sees* as many contexts as desired. A context consists of sets (collection of elements or a type definition), constants and assumptions (axioms) of the system. A machine contains the state (global) variables whose values are assigned in *events*. Events can only occur when enabled by their *guards* being true and as a result *actions* are executed. Events can have *parameters* (local variables) used by guards or actions. *Invariants* defines the dynamic properties of the specification and POs are generated to verify that these properties are maintained before and after an event is enabled. The predicates (axioms, invariants, guards) can also be defined as *theorems*. Theorems are proved from other invariants and axioms of seen contexts [3].

An abstract Event-B specification can be refined by adding more details in order to make it closer to an implementation (concrete). A context *extends* an abstract context by adding sets, constants or axioms. The abstract context properties are still assumed. Refinement of a machine consists of refining existing events. The relation between variables in the concrete and abstract model is given by a *gluing invariant*. POs are generated to ensure that this invariant is preserved in the concrete model. It is possible to add new events that refine *skip* which may be declared as convergent, meaning they do not cause divergence. The convergence is proved if each new event decreases a *variant*. The variant must be well-founded and may be an integer or a finite set.

Decomposition Styles

In order to explain the decomposition we will use the inverse operation (composition) as a subterfuge. Composition can be described as the capacity to model the interaction of partial specifications (sub-components) generating larger/full specifications. The interaction of partial specifications occurs through shared state [3], events' synchronisation [14] or a combination of both [5]. Shared state composition allows the interaction of sub-components by state sharing. Because variables usually define the state of a system, this composition is also known as shared variable. When specifying an automated teller machine (ATM) system, *user* and *cash machine* can have separated partial specifications. Both partial specifications can define variables to describe the used debit/credit cards for the transactions. The composition of these two specifications can interact through *shared variables*: the variables representing the cards. The other variables used only in a single partial specification are called *private variables*. A shared event composition allows sub-components to interact through synchronised events in parallel; moreover sub-components can communicate using shared parameters which is useful for modelling message passing systems. A constraint of the latest is disallow variable sharing. Returning to the ATM system, the partial specification *user* can have an event that defines the personal identification number (PIN) of the card:*user_defines_PIN* and *cash machine* contains an event that changes the card PIN: *change_PIN_card*. A shared event composition of the partial specifications originates a new event *user_change_PIN* that allows the introduction

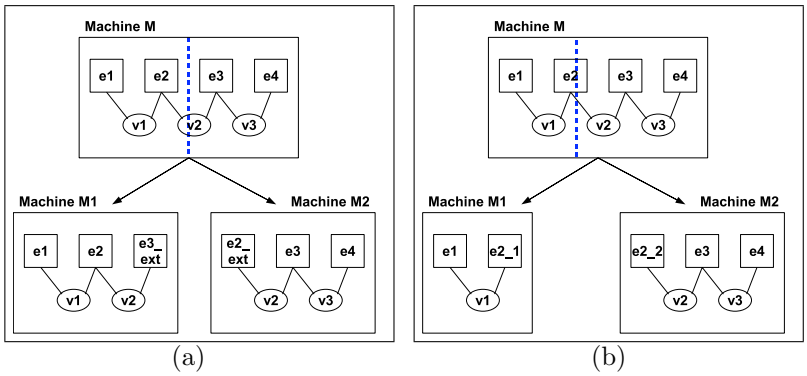


Figure 1. Shared Variable decomposition on the left and shared event decomposition on the right

of a new PIN for a particular card. Such event could be specified by composing events *user_define_PIN* and *change_PIN_card*. Both shared state and shared event composition allow the individual refinement of the partial specifications. Another option is to define the full specification of the ATM system using a mix of both composition styles. The *decomposition* of a specification means finding two or more components (sub-components) whose *composition* refines some abstract machine. Two styles where identified for the decomposition of a specification: *shared variable* and *shared event* based on the composition approach. Like in the composition, the decomposed sub-components can be further refined [1, 12].

We describe the two decomposition styles using Fig. 1. Machine M has events *e1* to *e4* and variables *v1* to *v3*. The solid lines connect variables used by events. In Fig. 1(a), M is decomposed using shared variable decomposition: events in the original component are selected and partitioned among the sub-components. In Fig. 1(a) the decomposition is represented by the dashed line in M: *e1* and *e2* are allocated to sub-component *M1* and *e3* and *e4* are allocated to sub-component *M2*. Consequently, *v1* and *v3* are private variables of *M1* and *M2* respectively. On the other hand, variable *v2* is a shared variable between *e2* and *e3*. To keep the same behaviour of the shared variable among the sub-components, additional events (called *external events*) are introduced. They simulate how shared variables are handled in the other sub-components: external event *e3_ext* is added to $M1$ and *e2_ext* is added to $M2$. A restriction of the shared state approach is that the shared variables and respective external events in the sub-components cannot be refined and must be always present in the the further refinements. The re-composition of the (refined) sub-components should always be possible resulting in a refinement of the original system [1].

In Fig. 1(b), M is decomposed using the shared event decomposition: variables in the original component are selected and partitioned among the sub-components. The decomposition is represented by the dashed line in M: *v1* is placed in *M1* and *v2*, *v3* are placed in *M2*. Events using variables allocated to different sub-components (*e2* shares *v1* and *v2*) must be split into partial versions of the non-decomposed event. A partial version of an event consists in a copy

of original event restricted to a variable (or variables): only parameters, guards and actions referring to the specified variable are preserved from the original. *e2_1* in *M1* is a partial version of *e2* restricted to variable *v1* and similarly, *e2_2* is restricted to *v2*. Sub-components can be further refined independently. For the application of each of the styles, shared event approach is suitable for developing message-passing distributed systems while shared variable approach is suitable for designing parallel algorithms [9].

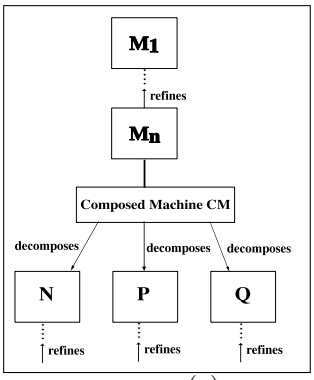
Limitations

For the shared variable decomposition, the partition of events is always possible in the sense that it is always possible to generate sub-components. However that decomposition might be less significant despite being possible: a large number of shared variables may not be of much interest in particular for further refinements that become more complex and do not benefit the development. The point of decomposition is important, since if it is done too early in the development, the sub-component might be too abstract and will not be able to be refined (without knowing more about the other sub-components). If the system is decomposed too late, e.g. when the system is already concrete, it will not benefit from the approach anymore.

For the shared event decomposition, the partition of variables is not always possible for all developments. Due to the restriction of shared variables, it might be necessary to have a "preparation refinement step" to solve complex predicates (invariants, guards, axioms) or assignments (actions) by separating variables allocated to different sub-components. If that step is not done, these complex predicates/assignments are automatically flagged by the tool and the user's intervention is required to explicitly make the separation (such operations cannot be done automatically). Another limitation is that we do not allow the overlapping of elements in the sub-components which sometimes may be useful. Even in the shared variable approach, the overlapped (shared) elements cannot be further refined independently.

Decomposition Tool

The Rodin Platform [22] is the result of an EU research project. It is a software toolset, based on modern software programming tools developed to use Event-B notation. It is open source, based on Eclipse Platform [15] and it works has a complement for rigorous modelling developments [13]. The aim is to benefit industry by permitting the integration of any necessary functionality in the same tool. Rodin contains a Static Checker that analyses Event-B components for syntactical errors (well-formedness and typing of models). There is a Proof Obligation Generator for generating PO and these obligations can be discharged by a theorem prover. An important Rodin feature is the high level of extensibility reflected by, for instance, the ability to contribute plug-ins. Plug-ins are components providing a certain type of service within the context of the Eclipse workbench. By components here we mean objects that may be configured into a system at system deployment time [15], such as the default theorem prover (B4free [4]) or model checking systems (ProB [21]). The decomposition tool described here is also implemented as a plug-in for the Rodin platform.



(a)

COMPOSED MACHINE CM
TYPE SHARED EVENT
REFINES M_n
INCLUDES
N P Q
EVENTS
evt_1 **REFINES** M_n.evt_1
Combines Events N.evt_1 ‖ P.evt_1
...
evt_n **REFINES** M_n.evt_n
Combines Events P.evt_n ‖ Q.evt_n
END

(b)

Figure 2. Decomposition tool diagram for a machine M_n and composed machine CM using the shared event approach

The input for the decomposition is a machine of a given Rodin project selected by the end-user. After the selection of the decomposition style and decomposition configuration, the tool generates the sub-components automatically. Below are the steps to be followed in order to decompose machine M_n in Fig. 2(a):

1. The end-user selects a machine M_n to decompose.
2. The end-user defines sub-components to be generated: N, P, Q ....
3. The end-user selects the decomposition style to use:

   Shared Variable: The end-user selects the events to be allocated to sub-components.

   Shared Event: The end-user selects the variables to be allocated to sub-component.

4. The end-user can opt to decompose the seen contexts into the sub-components similarly to the machine decomposition.
5. Sub-components are generated according to the decomposition configuration.
6. The decomposition configuration is stored as a composed machine.
7. Sub-components N, P, Q ... can be further refined.

The decomposition style to be used depends on the input system and on the end-user's preference. The decomposition generates sub-components according to the configuration: events/variables partition, sub-components stored in new projects or in the same one; context is to be decomposed or copied. That configuration is stored persistently in a composed machine [23] for future reuse or editing as seen in Fig. 2. "Replaying" the decomposition might require additional storing mechanisms. We intend to address this issue in the future. The configuration is performed through the Rodin Graphical User Interface. A limitation of the tool is to select which invariants are allocated to which sub-component. Currently, a solution is to introduce these invariants into the model before decomposition, e.g. into M_n, as theorems.

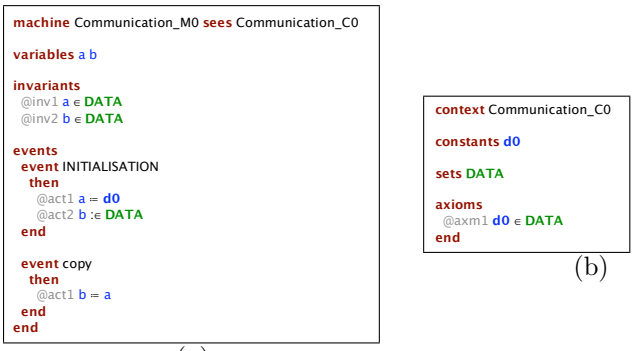


Figure 3. Diagrams corresponding to the *Communication* example

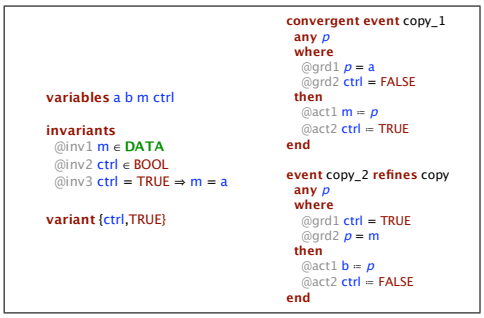


Figure 4. Excerpt of machine *Communication_M1*, refinement of *Communication_M0*

Decomposing a Communication protocol

We demonstrate the use of the decomposition tool through an example: a communication process. The abstract model called *Communication* can be seen in Fig. 3.

The variable a is initialised with the constant *d0* and variable b is assigned any value non-deterministically. The initial model contains only the event *copy* that copies the value of a to variable b in one single step. A refinement of *Communication* (*Communication_M1*) introduces a middleware entity that copies the value of a to b in two steps: the value of a is stored temporarily in the variable m (middleware) before being copied to b as seen in Fig. 4. Note that a control variable *ctrl* is introduced to ensure that the value of m is valid to be copied to b.

Invariant *inv3* expresses that when the variable *ctrl* is true, the value of the middleware m corresponds to the value of the source a. This invariant can be seen as a requirement of the

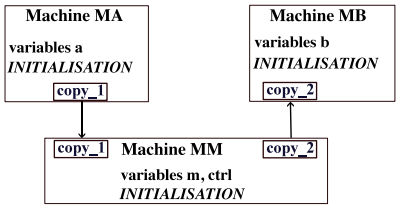


Figure 5. Decomposition of *Communication_M1* into machines *MA*,*MB* and *MM*

refinement of event *Communication_M0.copy* by *Communication_M1.copy_2*. The convergent event *copy_1* requires a variant that guarantees that this event cannot be executed forever. The variant is expressed as $\{ctrl, TRUE\}$ which means that eventually the control variable *ctrl* will be TRUE and in that case *copy_1* event is not executed since the respective guard is disable.

Depending on the chosen decomposition style and configuration, the system *Communication_M1* can be decomposed into different number of sub-components as seen in the following sections.

Shared Event Decomposition of *Communication*

From the modeller's point of view, the decomposition starts by defining which sub-components will be generated. The following step defines the partition of variables over the sub-components. The rest of the model decomposition (events, parameters, invariants, contexts) is a consequence of the variables allocation as defined below. For the shared event decomposition, we decompose *Communication_M1* in three parts: *MA*, *MB* and *MM* as seen in Fig. 5.

Using the decomposition tool, we define the partition as follows: variable a is allocated to machine *MA*, variables m and *ctrl* to machine *MM* and variable b to machine *MB*. It follows that event *copy_1* is split between *MA* and *MM* and event *copy_2* is split between *MB* and *MM*. A diagram of the use of the tool can be seen in Fig. 6 and the resulting machines can be seen in Fig. 7.

Next we describe the steps for a machine decomposition focusing on invariants, events, variant and contexts. The initial partition of variables between the sub-components defines the rest of the decomposition as detailed below.

Invariants: The decomposition of the invariants depends on the scope of the variables. The tool only maintains the invariants related with variable type definition as seen for *inv1* and *inv2* in *Communication_M1* (Fig. 4). The other invariants depend on the input of the user since they might be a constraint of the composed component and not a requirement of the sub-component. For instance, invariant *inv3* in *Communication_M1* $ctrl = TRUE \Rightarrow m = a$ contains three variables: *ctrl*, m and a. According to the defined decomposition configuration, *ctrl* and m are variables of *MM* and a is a variable

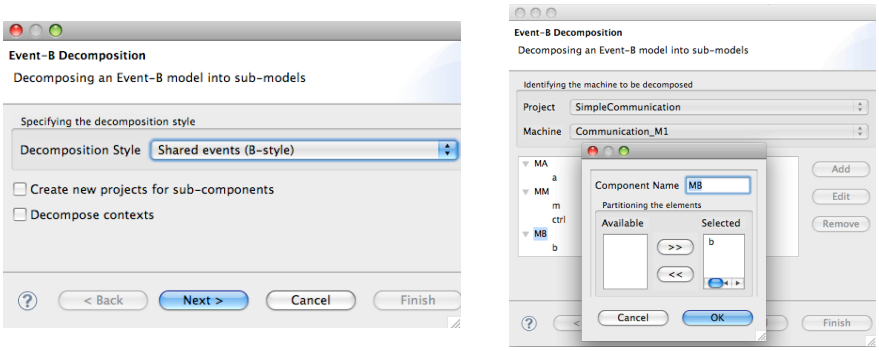


Figure 6. Shared event decomposition on *Communication_M1* using the tool

```
machine MA

sees Communication_C0

variables a

invariants
    @inv1 a ∈ DATA

events
  event INITIALISATION
    then
      @act1 a ≔ d0
  end

  event copy_1
    any
      p
    when
      @grd1 p = a
  end

end
```

(a)

```
machine MM

sees Communication_C0

variables m ctrl

invariants
    @inv1 m ∈ DATA
    @inv2 ctrl ∈ BOOL

events
  event INITIALISATION
    then
      @act1 m :∈ DATA
      @act2 ctrl ≔ FALSE
  end

  event copy_1
    any p
    where
      @grd1 p ∈ DATA
      @grd2 ctrl = FALSE
    then
      @act1 m ≔ p
      @act2 ctrl ≔ TRUE
  end

  event copy_2
    any p
    where
      @grd1 ctrl = TRUE
      @grd2 p = m
    then
      @act2 ctrl ≔ FALSE
  end
end
```

(b)

```
machine MB

sees Communication_C0

variables b

invariants
    @inv1 b ∈ DATA

events
  event INITIALISATION
    then
      @act2 b :∈ DATA
   end

  event copy_2
    any
      p
    when
      @grd1 p ∈ DATA
    then
      @act1 b ≔ p
  end
end
```

(c)

Figure 7. Machines *MA*, *MM* and *MB*

of *MA*. This suggests that *inv3* can be a constraint of the composition of the sub-components and not a constraint of the individual sub-components. As a result, *inv3* in *Communication_M1* is not part of any of the sub-components. Alternatively when an invariant clause is demanded and uses variables placed outside the scope of a sub-component, a further refinement of the composed component might be required to make an explicit separation of the variables.

Events: The partition of the events depends on the partition of the variables. For instance, variables m and *ctrl* are part of *MM* so their initialisation is allocated to that sub-component. Event *copy_1* in machine *Communication_M1* is split between *MA* and *MM* and has a parameter p. When the decomposition occurs, that parameter is shared between the decomposed events and the parameter type information is added. The guard of a decomposed event inherits the guard on the composed event according to the variable partition. Variable a is not within the scope of machine *MM* so only the type of p is defined in the guard of *MM.copy_1*. As mentioned in the limitation of the approach, a guard or action involving variables of different sub-components need to be explicitly separated.

Variant: Variant is only necessary when new events are introduced in a refinement. Decomposed events in sub-components are inherited from the composed component so no new events are introduced meaning that variants are not required.

Contexts: The context *Communication_C0* used in the example is shared between all the machines. That context (and possible others) can be flatted into a single context and decomposed. On the one hand, decomposing contexts can inadvertently remove relevant information. On the other hand, not decomposing it can add too many (not relevant and unnecessary) hypothesis which is not beneficial for the proofs: on the contrary, it might be harmful and complicate the proving process. Therefore, the context decomposition is optional as it varies with the system being modelled.

Shared Variable Decomposition of *Communication*

Before using share variable decomposition we do a further refinement. The idea is to store the values after copying into a simple database. We represent the database fields (*REGID* and *PRIORITY*:*LOW*, *MEDIUM* or *HIGH*) in the context *Communication_C1* as seen in Fig. 8(b). We introduce a boolean variable *processQueue* that is true when a new value is received and need to be processed. A new event *enqueueDB* is also introduced to store a new register in the database based on the received value as seen in Fig. 8.

Now we decompose *Communication_M2* by shared variable decomposition. The copy of the values and respective processing (storing in the database) are separated into machines *MCopy* and *MProcess*. Using the decomposition tool as seen in Fig. 9 we allocate events *copy_1* and *copy_2* to *MCopy* and the event *enqueueDB* to *MProcess*.

The first step is to select which variables are accessed for each sub-component and afterwards separate shared variables from privates ones for each sub-component. The shared variables are used in events *copy_2* and *enqueueDB*: *processQueue* and *b*. All the other variables are private to each sub-component. The invariants depend on the initial separation of variables. The following step is to separate/create the private/external events: the event partition according to the decomposition is applied; *copy_2* and *enqueueDB* use shared variables and as a consequence it is required an external event in the other sub-component. An external event *copy_2* is created in *MProcess* using the shared variables. The other variables used by the original *copy_2* become parameters in the external event as they are not in the scope of that sub-component (*ctrl* and m). The event *enqueueDB* is similarly built. The resulting machines can be seen in Fig. 10.

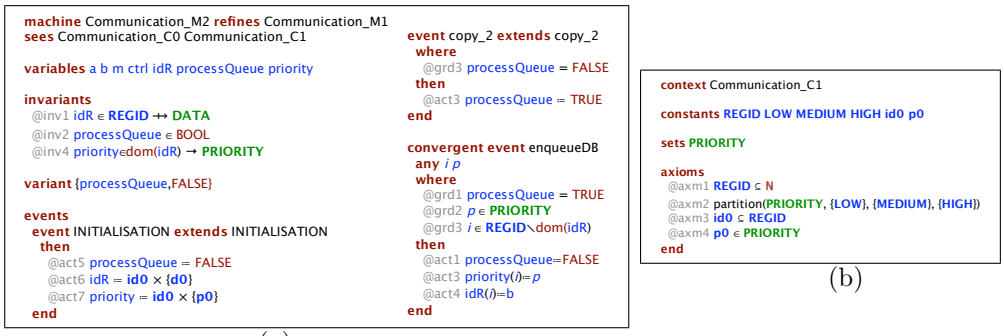


Figure 8. Excerpt of machine *Communication_M2* and context *Communication_C1* that extends *Communication_C0*

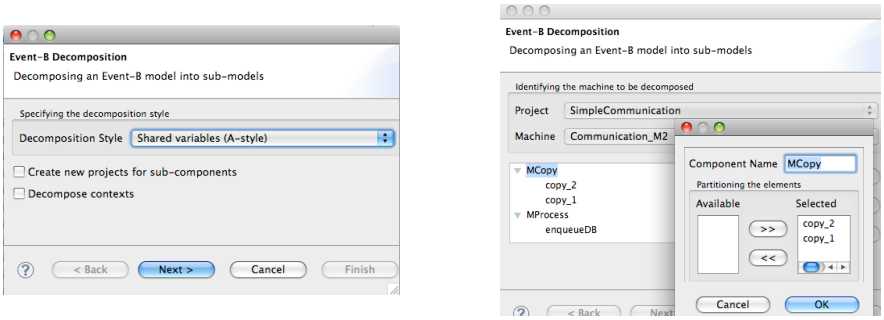


Figure 9. Shared Variable Decomposition of *Communication_M2*

Conclusion

This paper presents the decomposition of Event-B models and tool support in the Rodin platform. Decomposition can advantageously be used to decrease the complexity and increase the modularity of large systems, especially after several refinements. The main benefits are the distribution of POs over the sub-components which are expected to be easier to be discharged and the further refinement of independent sub-components in parallel introducing team development of a model which is attractive for the industry. Our goal is to develop a robust tool to model distributed systems that can be used by the industry. Application to more complex case studies and scalability issues will help improving the tool.

Several works try to exploit the decomposition benefits: [7, 6] study the formal development of MAS (Multi-Agent Systems) which are complex distributed systems to be used for critical applications using abstraction and decomposition for classical B and Event-B. [20] also studies

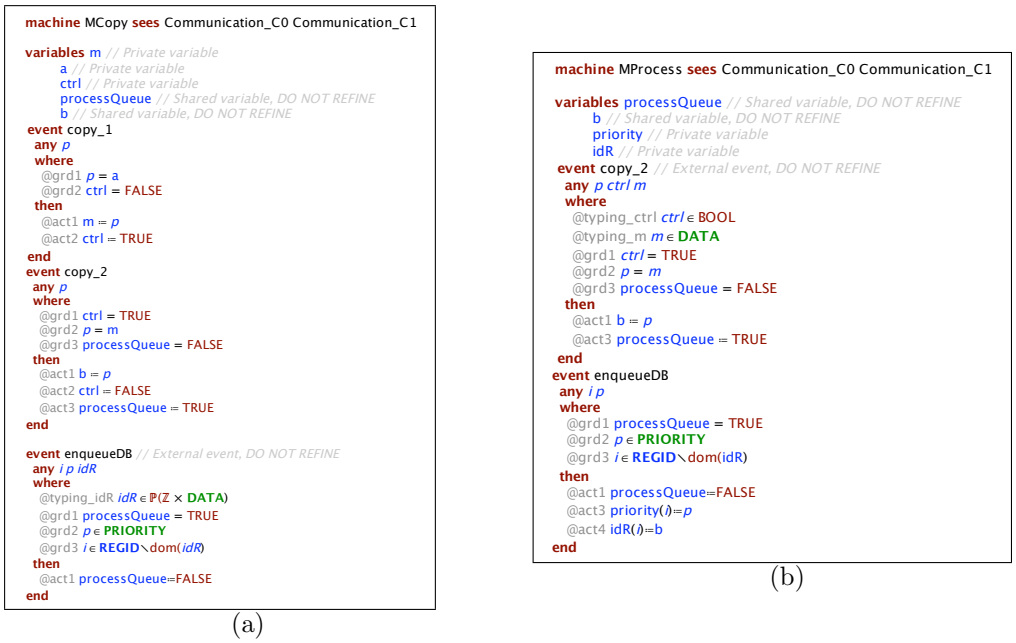


Figure 10. Excerpt of the output of shared variable decomposition of *Communication_M2*: *MCopy* and *MProcess*

MAS using shared variable decomposition to model a platoon of vehicles using Event-B. [10] uses the shared event approach in classical B to decompose a railway system into three sub-components: Train, Track and Communication. The system is modelled and reasoned as a whole in an event-based approach, both the physical system and the desired control behaviour. [17] develops a parallel program in Event-B using the tool presented in this paper. [16] proposes an automatic decomposition method using LOTOS [19]: the correctness is ensured if the combined behavior of decomposed sub-specifications is the same as the system's behavior before the decomposition. The method decomposes a process into two processes composed by the parallel operator and automatically generates an additional process that gives some information about the synchronization. The additional process corresponds to the middleware in a shared event decomposition in Event-B.

There is a need for modularisation and reuse of sub-components in order to model large systems and manage better the respective POs. Event-B lacks a sub-component mechanism so we propose to tackle that problem through the decomposition of a system by their events or variables. The shared variable (state-based) approach is suitable for designing parallel algorithms while the shared event (event-based) is suitable for message-passing distributed systems [11]. [3] suggests the shared variable decomposition where variables are shared and introduces the notion of external events. [11] suggests the shared event decomposition

where events are partition through the sub-components and the interaction occurs via shared parameters. The work developed by Butler in [8] for action system is strongly related with the same approach for shared event decomposition in Event-B [11] as both approaches are state-based formalism combined with event-based CSP [18]. The end-user chooses a decomposition style depending on specific systems and on its modelling preferences. The decomposition configuration is stored persistently for replaying/editing although further study is still required for this matter. We present by an example the different styles of decomposition of a system using the developed tool in the Rodin platform. A visualisation view for decomposition seems intuitive and we intend to explore in the future.

REFERENCES

1. Jean-Raymond Abrial. Event Model Decomposition. Technical report, ETH Zurich, 2009 (Unpublished).
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, May 2010.
3. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inf.*, 77(1-2):1–28, 2007.
4. B4free. http://www.b4free.com, September 2008.
5. R. J. R. Back and M. J. Butler. Fusion and Simultaneous Execution in the Refinement Calculus. *Acta Informatica*, 35(11):921–949, 1998.
6. Elisabeth Ball. *An Incremental Process for the Development of Multi-Agent Systems in Event-B*. PhD thesis, Southampon University, 2008.
7. Elisabeth Ball and Michael Butler. Using Decomposition to Model Multi-agent Interaction Protocols in Event-B. In *FM'06 Doctoral Symposium*. Springer, 2006.
8. Michael Butler. Refinement and Decomposition of Value-Passing Action Systems. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 217–232, London,UK, 1993.
9. Michael Butler. An Approach to the Design of Distributed Systems with B AMN. In *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212*, pages 221–241, 1997.
10. Michael Butler. A System-based Approach to the Formal Development of Embedded Controllers for a Railway. *Design Automation for Embedded Systems*, 6:355–366, 2002.
11. Michael Butler. Synchronisation-based Decomposition for Event-B. In *RODIN Deliverable D19 Intermediate report on methodology*, 2006.
12. Michael Butler. Decomposition Structures for Event-B. *Integrated Formal Methods iFM2009*, February 2009.
13. Michael Butler and Stefan Hallerstede. The Rodin Formal Modelling Tool. *BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London.*, December 2007.
14. Michael J. Butler. *A CSP Approach to Action Systems*. PhD thesis, Oxford University, 1992.
15. Eclipse. Eclipse homepage. http://www.eclipse.org, September 2008.
16. Kentaro Go and Norio Shiratori. A Decomposition of a Formal Specification: An Improved Constraint-Oriented Method. *IEEE Transactions on Software Engineering*, 25(2):258–273, 1999.
17. Thai Son Hoang and Jean-Raymond Abrial. Event-B decomposition for parallel programs. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *ASM*, volume 5977 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2010.
18. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.
19. ISO. LOTOS A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Information Systems Processing - Open Systems Interconnection, 1987.
20. Arnaud Lanoix. Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In *TASE '08: Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 297–304, Washington, DC, USA, 2008. IEEE Computer Society.
21. ProB. http://www.stups.uni-duesseldorf.de/ProB/overview.php, September 2008.
22. Rodin. RODIN project Homepage. http://rodin.cs.ncl.ac.uk, September 2008.
23. Renato Silva and Michael Butler. Parallel Composition Using Event-B. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B, July 2009.