Electronic Communications of the EASST
Volume 46 (2011)

Proceedings of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS 2011)

A Survey on Event-B Decomposition

Thai Son Hoang and Alexei Iliasov and Renato A Silva and Wei Wei

15 pages

Guest Editors: Jens Bendisposto, Cliff Jones, Michael Leuschel, Alexander Romanovsky
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A Survey on Event-B Decomposition

Thai Son Hoang[1] and Alexei Iliasov[2] and Renato A Silva[3] and Wei Wei[4]

[1] Department of Computer Science
ETH-Zürich
Email: htson@inf.ethz.ch
and
[2] School of Computer Science
Newcastle University, UK
Email: Alexei.Iliasov@ncl.ac.uk
and
[3] School of Electronics and Computer Science
University of Southampton, UK
Email: ras07r@ecs.soton.ac.uk
and
[4] SAP Research Darmstadt
SAP AG
Email: wei01.wei@sap.com

Abstract: Model decomposition is a powerful tool to scale the design of large and complex systems. It enables developers to separate components development from the concerns of their integration and orchestration. Event-B is a refinement-based formal method, equipped with three decomposition styles that come with solid semantic foundations and strong tool support. This paper intends to give some useful insights and modelling guidelines for using these decomposition styles, illustrated by an actual development of a master data updating system.

Keywords: Decomposition, Event-B, Modeling, Guidelines, Formal Methods

1 Introduction

Modern software systems are becoming more complex everyday. This trend is going to continue as industry witnesses unprecedented explosive demands for social and business connectivity. This poses a huge challenge to system modelling for rigorous quality assurance, while scalability becomes a great issue. Hence model structuring is paramount in mastering complexity of large systems. Although there has never been a lack of theoretical research in this area, few modelling platforms provide a sufficiently good level of tool support to evaluate structuring techniques and their impact on industrial application of formal modelling. Moreover, there are no clear guidelines for model designers which often impedes the application of formal verification to large-scale designs.

Event-B [3] has established itself as a popular formal system development with broad industrial adoptions. Event-B was designed as a minimalistic formalism to form the core of the Rodin platform [4]. A great range of features have been provided as extensions to the modelling platform, covering almost all aspects of model-driven engineering such as requirement, simulation, visualization, testing, formal verification, and code generation. The extension architecture

allows a modeller to enable only those features that are relevant for the domain of modelled problem. This keeps the core features of the platform as simple as possible while not sacrificing any functionalities.

The basic Event-B language supports model structuring by nothing more than the use of events and refinement inside one model. However, it is highly impractical to construct a large scale formal design as one monolithic model, which results in numerous problems with legibility, maintainability, team work, reuse, and so on. Notably, it also affects proof structuring: autonomous provers are suffocated by a large number of hypotheses and thus anything that makes the context of a proof smaller is extremely beneficial.

Model decomposition comes as a great potential to solve the above problem. Three different decomposition styles exist for Event-B, all providing a guarantee of refinement monotonicity: the model after decomposition is a correct refinement of the one before decomposition, provided all obligated proofs are given. This allows a decomposed part of the model to be treated as an independent artefact so that the modeller can concentrate on this part and does not have to worry about the other parts. The tool supports for these techniques, realized as Rodin plugins, not only provide assistance on how to decompose a model, but also generate explicit constraints and relations between the decomposed model and the resulting sub-components.

Technically, decomposition is a special form of refinement by which a single abstract model is refined by several concrete models and their aggregation. Like refinement, a properly planned decomposition step results in very low proof cost. It requires, however, a good degree of foresight. First, a model is difficult to decompose if the model does not possess a structure that can be easily divided and mapped to independent components to be produced by decomposition. Second, any careless design decision before decomposition may be difficult to correct afterward, which often leads to complete rework on the development of each individual local component. Therefore, a modeller can serve a better job when some guidelines are available so that the modeller knows which decomposition style to choose from, and which design decisions need to be taken care of at each stage.

In this paper we provide some insights and propose modelling guidelines for applying model decomposition, drawn from our personal experiences and illustrated by a case study by an industrial user of Event-B. These guidelines are not supposed to give a one-for-all solution for all decomposition attempts. We introduce stages in decomposition and point out which design problems (that can be easily overlooked) need to be addressed in each step.

Outline. Sec. 2 briefly introduces Event-B and decomposition in general. Sec. 3 gives our modelling guidelines. Different decomposition styles are detailed in Sec. 4 – Sec. 6 by showing their applications on the modelling of a master data updating system. We compare the different approaches in Sec. 7. Finally, we draw some conclusions in Sec. 8.

2 Background

2.1 Event-B

Event-B is a formal modelling method for developing *correct-by-construction* hardware and software systems. An Event-B model is a state transition system where the state corresponds to a set of *variables* v and transitions are represented by a collection of *events* evt. The most general

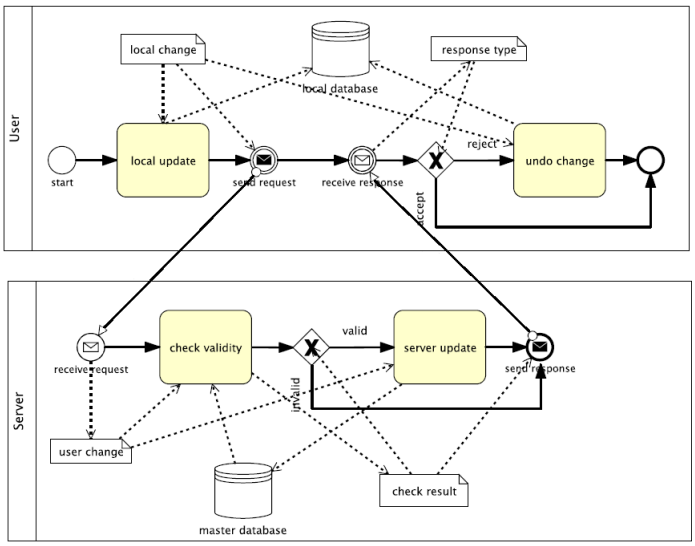

Figure 1: The Process Model of Update Master Data

form of an event is: evt $\widehat{=}$ **any** t **where** $G(t,v)$ **then** $A(t,v)$ **end** , where t is a set of parameters, $G(t,v)$ is the enabling condition (called guard) and $A(t,v)$ is an action changing the value of v. An action comprises several assignments executing in parallel. Each assignment can have one of the following forms: $x := E(t,v)$, $x :\in S(t,v)$ or $x :| P(t,v,x')$, where x are some variables in v. The first form assigns value of expression $E(t,v)$ to x. The second assignment form non-deterministically assigns to x some element of set $S(t,v)$. The third assignment form non-deterministically assign to x some after value x' satisfying the *before-after predicate* $P(t,v,x')$. In the first and last assignment forms, x can be a vector of variables. The last assignment form is also the most general one: other assignment forms can be equivalently represented using before-after predicates. Essential to Event-B is the formulation of *invariants* $I(v)$: safety conditions to be preserved at all times.

To facilitate the construction of large-scale models, Event-B advocates the use of *refinement*: the process of gradually adding details to a model. An Event-B development is a sequence of models linked by refinement relations. It is said that a concrete model refines an abstract one. Abstract variables v are linked to concrete variables w by a *gluing invariant* $J(v,w)$. Any behaviour of the concrete model must be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v,w)$.

Rodin [4] is an industrial-strength toolset supporting Event-B. Rodin provides an integrated modelling environment with a range of editors, modelling assistants, automatic generator of verification conditions and a set of automated provers tasked to discharge verification conditions.

Example – A Master Data Updating System Fig. 1 shows a master data updating system that we use as the case study of this paper. The system consists of a User process and a Server process keeping some master data in sync. When User proposes a data change, it first updates its local copy, and then sends a request message to Server and waits for the answer. Upon receiving

the request, Server checks the validity of the proposed change, and updates the master copy only when the change is deemed valid. Then, Server sends to User a response containing either an approval or a rejection. User has to roll back the change if a rejection is received. We are interested in the global property that the *local and master databases are always identical before and after each data update procedure*.

The Event-B model in Fig. 2 serves as the most abstract view of the above system. The initial model is designed to have as few variables and events as sufficient to express the above mentioned property (see invariant **inv0_3**). The model contains variables *udb* and *sdb* denoting User and Server's database respectively. The Boolean variable *is* denotes if the global system is *in synch*. There are three events, namely u_update, s_update and u_final. When the system is in synch, u_update changes the local database, which invalidates the insynch status. While the system is out of synch, s_update may occur to update the server database (either to be the same as *udb* or unchanged). Finally, u_final occurs to put the system back in synch by making the local database to be identical as the server database. Note that s_update may be skipped when the system is out of synch. Although this cannot happen in the real system, we permit it in the abstract model to simplify proofs, which does not affect the satisfaction of the global property in consideration. This spurious behavior will be removed by refinement and decomposition in the further developments.

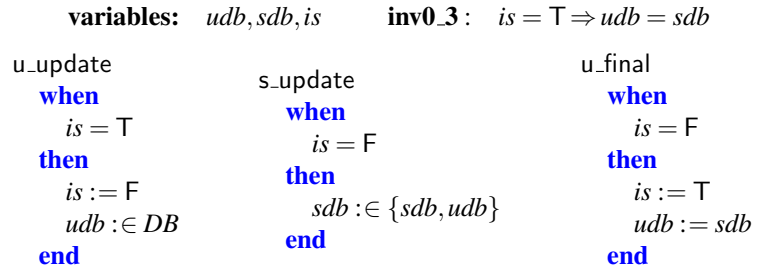

Figure 2: The top abstract model of Update Master Data

2.2 Decomposition

The *top-down* style of development used in Event-B 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. *Decomposition* addresses such difficulty by providing a mechanism for splitting a large model into several sub-models (that can be further developed independently). Several decomposition techniques have been proposed by extending the existing Event-B notation. This paper is concerned with three existing approaches: *shared-variable* [2], *shared-event* [7] and *modularisation* [11], all of which are supported by Rodin plug-ins [13, 1]. These decomposition techniques differ in that different model elements are shared among sub-components. For shared-variable decomposition, a part of state information (*variables*) is shared among sub-components. Further refinements then concentrate on how each component processes shared state information. For

shared-event decomposition, a set of *events* are synchronised and shared by sub-components. Hence, it is important to take care of the inputs/outputs of these synchronised events. Modularisation defines a set of *interfaces* that are shared and accessed by different components. Interfaces provide callable operations and promises that these operations can deliver. The implementation of an operation should guarantee that the promises are fulfilled for any given circumstance.

Shared-variable decomposition is similar to rely/guarantee approach from Jones [12]: internal/external events is essentially an encoding of rely/guarantee conditions. It also corresponds to *concurrent action systems* [5] where a solution for the interleaving semantics is proposed.

Shared-event decomposition allows separation of aspects by using synchronisation and communication based on Butler's work [6] combining Action System and CSP [10]. CSP value passing channels correspond to events that communicate via shared parameters.

The separation of procedure declaration from implementation have a long history both in computer programming and modelling. Modularisation closely relates to the treatment of procedures in Hoare logic [9, 8]: procedure calls are used as a metaphor to benefit from refinement monotonicity. Consequently independent model aspects can be considered separately although this should not be confused with modelling a procedure call as a construct of a programming language.

3 Guidelines

The primary challenge of applying decomposition is to ensure that the structure of the original model fits the requirements of the chosen decomposition style, leading to helpful sub-models that can be developed separately with a tangible advantage in terms of proof efforts and overall model scale. As with any *top-down* approach for system development using refinement, the more abstract models are initially, the more useful the decomposition step will be. Here we do not focus on directly justifying the use of a particular decomposition style. Instead we focus on *how to proceed* when decomposing using one of the suggested decomposition styles.

We define a general top-down guideline for the three decomposition techniques based on the following common template.

Stage 1 To model the system abstractly, expressing all the relevant global system properties.

Stage 2 To refine the abstract model to fit the structure expected by a given decomposition technique.

Stage 3 To apply decomposition.

Stage 4 To develop the resulting sub-models independently.

Following this guideline, global properties are captured early in the model and guaranteed to hold in the final models by combining refinement and decomposition. The development of each decomposed part is done independently of the others. Consequently, we can have different implementations for a decomposed model that is guaranteed to work with any implementation of other decomposed models.

In the subsequent sections, we elaborate on the application of different decomposition techniques using our proposed modelling guideline.

4 Shared-Variable Decomposition

Consider Fig. 3 where machine M has four events, e1 to e4, and three variables, *v1* to *v3*. The solid lines connect variables used by events. In Fig. 3, M is *shared-variable* decomposed and events are partitioned into sub-components: e1 and e2 are allocated to machine M1; e3 and e4 are allocated to machine M2. Consequently, *v1* belongs to M1 and *v3* belongs to M2 (*private variables*). Variable *v2* is *shared* between M1 and M2. Furthermore, additional *external events* are required to simulate how shared variables are handled in the other sub-component (e3_e is added to M1 and e2_e to M2). Assuming that e2 has the general form

$$\mathsf{e2} \;\widehat{=}\; \textbf{any}\ t\ \textbf{where}\ G(t,v1,v2)\ \textbf{then}\ v1,v2\ :|\ P(t,v1,v2,v1',v2')\ \textbf{end}\ ,$$

the corresponding external event e2_e can be generated as follows.

$$\mathsf{e2\_e} \;\widehat{=}\; \textbf{any}\ t,v1\ \textbf{where}\ G(t,v1,v2)\ \textbf{then}\ v2\ :|\ \exists v1'\cdot P(t,v1,v2,v1',v2')\ \textbf{end}\ .$$

Intuitively, e2_e is a projection of e2 onto the state without variable $v1$.

There exist certain constraints about shared-variable decomposition during the development of the resulting sub-components: they can be refined independently but shared variables and external events *must be present and cannot be refined*. More information on shared-variable decomposition in Event-B can be found in [2].

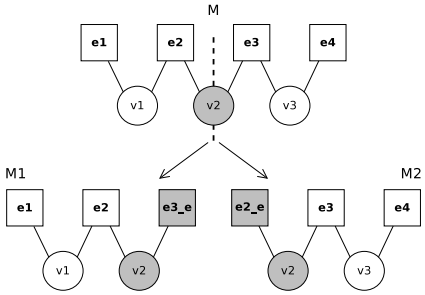

Figure 3: Shared-variable decomposition

4.1 Master Data Updating System

We describe in detail how to develop the example of update master data using shared-variable decomposition. The model described in Section 2 represents our abstract model of **Stage 1**. We continue with the subsequent stages of our modelling guideline.

Stage 2. Shared Channels Between Components. In this preparation stage, we introduce the channels (the shared elements) acting in between User and Server. The channels are modelled by two variables *creq* and *cres*, corresponding to the set of messages going through the request and response channels respectively.

Memo SV1 In this preparation stage, variables going to be shared are introduced.

Moreover, since in **Stage 4** the shared variables and external events can be *neither removed nor refined*, the shared elements introduced in this preparation stage must be *concrete*.

> **Memo SV2** The shared variables must be concrete.

Furthermore, we are going to split the variables and events into two groups, corresponding to each site, preparing for the later decomposition step. An important design constraint here is that User's events can only reference the variables belonging to User and the shared channels, but not the variables of Server. The same for Server's events.

> **Memo SV3** Events belonging to each sub-component only reference its own variables and shared variables.

As a result, variable *is* must be refined away. We replace *is* by *uis*, the local in-synch flag, with a gluing invariant $uis = is$, i.e. the global in synch is consistent with the User's view. A separated in-synch flag *sis* is introduced for Server. Moreover, in order to separate User and Server completely, we introduce new variables for keeping some information belong to each site. For User, variable *udb_old* is added in order to keep the old value of User's database for undoing later if necessary. For Server, variable *sc* keeps the user's change to the database on the server site for updating Server's database if the change is valid.

Despite of the details that we have to introduce in order to clearly separate the future sub-components, we aim to keep the model at this stage fairly abstract. It should contain only necessary information for maintaining the global properties and specifying the shared elements between future sub-components. Other information, e.g. control/data flows within each sub-component can/should be abstracted away.

> **Memo SV4** Unnecessary details irrelevant to decomposition should be abstracted from the model in **Stage 2**.

For example, we assume for the moment that the update of the database and sending the request message from User happens simultaneously. This is represented by event u_update_and_req, a refinement of the abstract event u_update.

u_update_and_req **refines** u_update
  any ch **where**
    $uis = \mathsf{T} \wedge ch \in CH$
  then
    $uis, udb, udb\_old, creq := \mathsf{F}, upd(udb \mapsto ch), udb, \{ch\}$
  end

In u_update_and_req, the local database *udb* is updated to be $upd(udb \mapsto ch)$, the new value obtained by applying changes *ch*; the old local database is saved in *udb_old*; and the actual change is send as a request to the server via channel *creq*. Note that u_update_and_req satisfies our **Memo SV3**, i.e. reference only variables belonging to User and the shared channel *creq*.

Other events in this model include: s_receive_req for Server to receive some request; s_accept_res for Server to update its database and send a positive response; s_reject_res for Server to send a negative response without updating its database; u_receive_res_acc and u_receive_res_rej for User to receive some (positive/negative) response and act accordingly.

An important advantage during the model design in this stage is the use of the abstract model from **Stage 1**. Consistency enforced by refinement guides our design in **Stage 2**, i.e. constraints on the shared variables will be *derived* from the need to maintain the global properties introduced in **Stage 1** (typically in terms of invariants).

In our example, the following invariants are *discovered* during the process of discharging proof obligations such as guard strengthening and invariant preservation of the model. They relate the content of the channels and the internal status of User and Server. Invariants **inv1_7** and **inv1_8** relate Server's database *sdb* with the User's database (current *udb* or old *udb_old*) depending on the content of the channel *cres*. Invariants **inv1_9** and **inv1_10** state that User is out of synch if there are some request or response messages.

$$\begin{array}{ll} \textbf{inv1\_7}: & cres = \{\mathsf{T}\} \Rightarrow sdb = udb \\ \textbf{inv1\_8}: & cres = \{\mathsf{F}\} \Rightarrow sdb = udb\_old \\ \textbf{inv1\_9}: & creq \neq \varnothing \Rightarrow uis = \mathsf{F} \\ \textbf{inv1\_10}: & cres \neq \varnothing \Rightarrow uis = \mathsf{F} \end{array}$$

Stage 3. Decomposition Summary. This stage is semi-automatic: we provide the tool with input on how the events are partitioned into different future sub-models. Intuitively, we separate our events into two groups, corresponding to User and Server accordingly. The variables distribution amongst these model are calculated according to the information about events distribution. The summary of our decomposed models is as follows.

	User	Server
Internal events	u_update_and_req u_receive_res_acc u_receive_res_rej	s_receive_req s_accept_res s_reject_res
Private variables	udb, udb_old, uis	sdb, sc, sis
Shared variables	$creq, cres$	$creq, cres$

Stage 4. Developments of Sub-models. We present a summary of the additional refinement steps for each model. Most invariants in the sub-models are technical and related to the sequentialisation of the actions, reflecting the process flows in Figure 1.

User The control flow is introduced via means of a program counter *upc* to capture the actual sequential steps inside the User process. Other internal variables of User are introduced accordingly, i.e. User's change *uch* and User's stored response type *ures*.

Server Similarly, the control flow of Server is introduced via means of a program counter *spc*. Internal variables of Server, such as the check result *scr*, are introduced.

5 Shared-Event Decomposition

In Fig. 4, M is *shared-event* decomposed into two parts: M1 and M2. Variables are partitioned into the sub-components: *v1* is placed in M1 and *v2*, *v3* are placed in M2. Unlike the shared variable approach, no variable sharing is allowed. Events using variables allocated to different sub-components (e2 shares *v1* and *v2*) must be *split*.

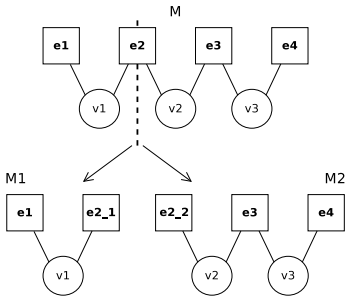

Figure 4: Shared-event decomposition

Assuming that event e2 has the following form

$$
\begin{array}{l}
\mathsf{e2} \\
\quad \textbf{any} \quad t \quad \textbf{where} \\
\qquad G_1(t, v1) \\
\qquad G_2(t, v2) \\
\quad \textbf{then} \\
\qquad v1 \ :| \ P_1(t, v1, v1') \\
\qquad v2 \ :| \ P_2(t, v2, v2') \\
\quad \textbf{end}
\end{array} \quad ,
$$

event e2_1 is defined as a partial version of e2 only referring to variable *v1*, i.e.

$$\mathsf{e2\_1} \ \widehat{=} \ \textbf{any}\ t, v1\ \textbf{where}\ G(t, v1)\ \textbf{then}\ v1 \ :| \ P_1(t, v1, v1')\ \textbf{end}\ .$$

Event e2_2 is defined similarly but only refers to *v2*.

> **Memo SE1** The abstract model of shared-event decomposition is such that each event updating non-private variables may be syntactically split into two events.

5.1 Master Data Updating System

Using the same initial model in Fig. 2, we describe the following stages in the application of a shared event decomposition. The system is designed to be decomposed into components User and Server *synchronously* communicating by value passing messages.

Stage 2. The Value Passing Protocol. The goal of this stage is to have a model where the state variables are *partitioned* amongst the future sub-models. Typically, this stage involves refinement of events to introduce the shared elements in the form of *events' parameters*. Similar to the shared-variable style, a good abstract model is pursued where only necessary information related to the global properties and the shared elements are specified.

> **Memo SE2** Irrelevant details should be abstracted away from the model before decomposition.

In this refinement, we prepare the decomposition by introducing synchronous channels and respective value passing protocol. The content of the protocol is represented by shared parameters (in the resulting sub-events). At this stage, the communication is abstract and occurs in a single event.

Memo SE3 Shared elements are introduced by means of event parameters.

The global flag *is* is replaced by *uis* and *sis* for User and Server sync respectively. The gluing invariants between *uis*, *sis* and *is* are given by *inv*1_1, *inv*1_2 and *inv*1_3: *uis* always matches *is*; while a request is being processed, *sis* matches *is*; otherwise, the server is synchronised ($sis = \mathsf{T}$).

variables:	$udb, sdb, uis,$	**inv1_1**	$uis = is$
	u_ch, u_rq	**inv1_2**	$u\_rq = PRC \Rightarrow sis = is$
	sis, s_st, s_ch	**inv1_3**	$u\_rq \neq PRC \Rightarrow sis = \mathsf{T}$

Some control variables are added: *u_ch* corresponds to the User change; *u_rq* holds the request state on the User side where *PRC* corresponds to the processing state; *s_st* corresponds to the server state and *s_ch* holds the change from the Server's viewpoint. Refined event u_update models a modification that is stored in *u_ch* before being sent to the server by the new event rq. Event rq simultaneously sends the request from User and receives it in the server. Then the request is stored in *s_ch* and User ($u\_rq := PRC$) and Server ($s\_st := VAL\_RQ$) states are updated . The server is considered out of sync once receives a request ($sis := \mathsf{F}$).

u_update **refines** u_update
  any *ch* **where**
    $ch \in CH$
    $uis = \mathsf{T}$
    $u\_rq = IDLE$
  then
    $uis, u\_ch := \mathsf{F}, ch$
  end

rq
  any *msg* **where**
    $uis = \mathsf{F} \wedge msg = u\_ch \wedge u\_rq = IDLE$
    $s\_st = S\_IDLE$
  then
    $u\_rq := PRC$
    $s\_ch, s\_st, sis := msg, VAL\_RQ, \mathsf{F}$
  end

Note that event rq has been designed so that it can be syntactically split into parts concerning only with variables of the User or Server (**Memo SE1**). The request validation can be deferred until the decomposition because it is irrelevant to the considered global property. The server is updated in the refined event s_update for a valid request. Even when the request is deemed invalid, a response is sent back by the new event rsp. This event syncs in the Server and updates the User's request. If the request is valid, *u_ch* is applied locally; if the request is invalid, *udb* remains the same. In either case, *udb* is back in sync with the server.

Several gluing invariants are discovered as a result of the generated proof obligations. **inv1_4** state that while *u_rq* is processed, User/Server changes match; if *u_rq* is deemed invalid, *sdb*/*udb* are identical (**inv1_5**); a valid request results in *sdb* matching with *udb* updated with *u_ch* (**inv1_6**).

inv1_4 :	$uis = \mathsf{F} \wedge u\_rq = PRC \Rightarrow s\_ch = u\_ch$
inv1_5 :	$u\_rq = INVLD \Rightarrow udb = sdb$
inv1_6 :	$u\_rq = VLD \Rightarrow upd(udb \mapsto u\_ch) = sdb$

Our model is synchronous since the messages exchanged by User and Server are sent and received simultaneously. Alternatively, we could also model asynchronous communication by introducing a *buffer* between *udb* and *sdb* suggesting a three way decomposition.

Stage 3. Decomposition Summary. Sub-models User and Server result from the allocation of the original variables according to their use. The decomposition is summarised in the following table:

	User	Server
Variables	udb, u_ch, uis, u_rq	sdb, s_st, sis, s_ch
Events	u_update, u_final, rq, rsp	s_update, rq, rsp

Stage 4. Developments of Sub-models. The decomposition allows the separation of sending/receiving a request by defining the request as parameter *msg* shared by User and Server (similarly applied to the server's response). The resulting sub-models can be refined independently:

User A program counter is added defining the local states (update *udb*, send request, receive response, commit/discard change). Two events refined the two possible outcomes: *l_commit* for valid modifications updating *udb* and *l_discard* to discard the modification. An additional refinement could add a request queue removing the waiting between the server reply and the next modification.

Server The server is refined by modelling the request validation with a new event s_val.

6 Modularisation

In modularisation, *interfaces* are defined for sub-components such as the interface I in Fig. 5, which contains interface variable *iv* and operations o1 and o2. Operations are specified by pairs of pre/post-conditions. An interface is separated from its implementation IM that provides concrete behavior for each of the interface operations. An abstract machine of the integrated system is modeled in M, which is refined by M1 where sub-component behavior is replaced with respective operation calls.

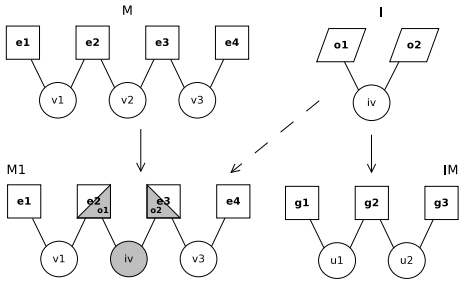

Figure 5: Decomposition via modularisation

In M, logical subunits must be identified so that they can be easily mapped to operation calls. Respective variables in M are mapped to interface variables, and actions can be replaced by interface operations. An interface must be carefully designed such that a minimal level of details of the sub-component behavior is exposed.

Memo M1 A module interface should be as general (weak) as possible because of re-usability.

Following the above principle, we should expect that sometimes an interface is too weak in that the post-conditions of its operations are not strong enough to establish the soundness of the caller machine.

Memo M2 In case of a too weak interface, we should strengthen operation post-conditions using the undischarged proof obligations as guidance.

Machine M1 can be further refined. However, operation calls to o1 and o2 must stay intact.

Memo M3 Operation calls must be preserved in the chain of refinements except for parameter refinement.

Unlike the two other decomposition approaches, modularisation offers a greater degree of flexibility on how to construct a decomposed specification because of lower coherence among sub-components. One consequence is that the modularisation approach applies to both *top-down* and *bottom-up* designs, and even blurs the boundary. For top-down development, the guideline in Sec. 3 can be used in modularisation approaches as well. In bottom-up development, sub-component interface may already exist before a global integration scenario is designed. This is useful for many industrial use cases in which service integrations and customizations are formally analyzed.

6.1 Master Data Updating System

We take a different approach here that does not follow the guideline in Sec. 3. Even though we will explain the approach in a top-down fashion, it resembles certain aspects of a bottom-up development in that the design of sub-component interfaces is relatively independent of the global integration, because the interfaces are standard communication interfaces that produce and consume messages.

We start with an abstract machine that only specifies message flows and does not state the global property. Interfaces are defined for the two processes and implemented by separate machines by adding local control and data flow information. A sufficient amount of implementation details, such as local variables and properties, is carefully chosen and exposed in their interfaces to enable the verification of the global property. Finally, the top abstract machine is refined by adding details of operation calls and message buffers. The global property is then verified on a final refinement.

Stage 1. Specifying Abstract Message Flows. The top abstract machine uses flags to indicate whether a certain message has been sent or received. For example, if the flag *req_snt* is $\top$ then a request message has been sent. Several invariants describing the order of message events are added for property verification later. As an example, **inv6** specifies that a response message has to be sent before it can be received. Message events are abstract at this level and simply set the flags accordingly.

variables: $req\_snt, req\_rcv, res\_snt, res\_rcv$
inv6 : $res\_snt = \mathsf{F} \Rightarrow res\_rcv = \mathsf{F}$
send_req $\widehat{=}$ **when** $req\_snt = \mathsf{F} \wedge res\_rcv = \mathsf{F}$ **then** $req\_snt = \mathsf{T}$ **end**

Stage 2. Process Interfaces. The interface of each process defines a set of message operations. These operations do not consider how messages are transported, but merely specify the types of messages that they provide or expect. For example, interface User provides messages to be sent (get_request), and take incoming messages fed to them for local consumption (put_response). In particular, get_request produces a message equivalent to the local change (ch) proposed by the user.

get_request
pre
$req\_snt = \mathsf{F} \wedge res\_rcv = \mathsf{F}$
return
msg
post
$msg' = ch$
$req\_snt' = \mathsf{T}$
end

put_response
any msg
pre
$msg \in \mathrm{BOOL} \wedge req\_snt = \mathsf{T}$
$res\_rcv = \mathsf{F}$
post
$ures' = msg$
$res\_rcv' = \mathsf{T}$
end

Stage 3. Process Implementations. Concrete details of sub-components are added in implementations, such as events that describe how control states and local variables are updated. Message-related flags are no longer present, thus we provide a link between those flags and local control states (u_cs) as gluing invariants like the one below among others.

inv18 $req\_snt = \mathsf{F} \Leftrightarrow u\_cs \in \{start, upd, req\}$

An interface implementation is essentially an Event-B refinement step. We need to prove that the postcondition of any interface operation must be fulfilled by the corresponding events that implement the operation. We also need to prove relative deadlock freedom that, whenever an interface operation is enabled, some of its implementing events must be executable.

Stage 4. Final Global Machine. The top-level machine is refined at this level to contain operation calls and actual message exchanging behavior. Each process has a buffer to store incoming messages. When a message needs to be sent, the corresponding interface operation of the sender process is called to retrieve the outgoing message, which is then added to the corresponding buffer. When a message is to be received, the message is taken out of the buffer and passed to the receiver process by calling the respective interface operation. In the following code, the prefix `user_` is used in interface variables and operations of the `user` module.

send_req $\widehat{=}$ **when** $user\_req\_snt = \mathsf{F} \wedge user\_res\_rcv = \mathsf{F}$ **then** $buf\_s = buf\_s \cup \{\text{user\_get\_request}\}$ **end**

Stage 5. Property Verification. Unlike the other approaches, the global property is encoded and proved at the final global machine. The proof is based on the symbolic values of the local and master databases delivered as operation post-conditions in the process interfaces.

7 Discussions

All approaches decompose global machines into two components, one per process[1]. Shared-variable and shared-event approaches start with similar abstractions, specifying a minimal set of events reflecting how the local and remote databases are updated while preserving the global property of interest. A series of refinements are introduced with appropriated chosen gluing invariants and proofs of deadlock freedom and convergence. The two approaches are different in that the shared-event version implements a synchronous message passing model. The choice is mostly motivated by the fact that synchronised message passing events can be shared by two processes. However, the system could be modelled in an asynchronous communication by introducing a *buffer* sub-component. These two approaches ensure that the global properties are preserved before decomposition. Afterwards each individual sub-component focus on their specific properties. A system involving a shared object is favoured by a shared variable decomposition where the shared object can be accessed by all the sub-components. On the other hand, communicating system parts can be shared event decomposed possibly introducing a middleware to allow an asynchronous approach.

In contrast to the other approaches, the modularisation version formulates and proves the global property at the final stage, after the module interfaces are designed. This is possible because the global machines and modules are loosely coupled, only linked by interfaces. An advantage that immediately comes to mind is flexibility: changes to existing components and inclusion of new components do not necessarily affect unchanged modules nor their proofs. However, the largest challenge of modularisation is the design of appropriate interfaces as they play a crucial role in linking multiple worlds while preserving the global property. During the design of our model, we go through iterations of "trial and error" to find the appropriate amount of information to be exposed in interfaces. As we learned from our experiences, a good practice is to start with an initial interface containing a minimal amount of information and weak possible post-conditions for operations. When these are insufficient to prove the global property, we can gradually bring more information to the interface and strengthen post-conditions.

8 Conclusion

We have presented modelling guidelines for three decomposition techniques of Event-B, illustrated by a case study. Due to limited space, it was impossible to tell the complete story of the case study development. We emphasise on the several design decisions that were taken during the development of the examples, in order to successfully apply decomposition, resulting in helpful sub-models for further independent elaboration. Together, the guidelines and these design decisions act as a document of our experience in applying decomposition techniques.

Decomposition is a powerful technique to cope with the complexity of system development but applying decomposition comes at a price: the cost of planning a development strategy that fits a certain decomposition style. It turns out that mastering the technicalities related to establishing decomposition correctness and even an extensive tool support are still not enough to systematically apply decomposition. Decomposition is rarely successful without prior planning

[1] The models are available online at http://eprints.ecs.soton.ac.uk/22164/

and such development planning is not yet supported by an established methodology.

One fascinating topic for future research is to see how different decomposition techniques complement each other in the same development. The diversity in decomposition approaches may be exploited to make decomposition more flexible and simpler for an end user. We are also planning to formalise the criteria of model decomposability and, if successful, mechanise them in a tool. Such a tool, in principle, could give an immediate answer on which decomposition technique, if any, would succeed for a given model.

Acknowledgment. This work has been supported by the EC FP7 Integrated Project Deploy, the EPSRC grant TrAmS (EP/E035329/1) and Fundação Ciência e Tecnologia (FCT-Portugal).

Bibliography

[1] Modularisation plug-in for Event-B. http://wiki.event-b.org/index.php/Modularisation_Plug-in.

[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 Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[5] R-J Back. Refinement Calculus II: Parallel and reactive programs. In *Proc. of REX 89*, volume 430 of *LNCS*, pages 67–93. Springer, 1989.

[6] M. Butler. Stepwise refinement of communicating systems. *Sci. of Comp. Prog.*, 27(2):139–173, September 1996.

[7] M. Butler. Decomposition structures for Event-B. In *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.

[8] D. Gries. *The Science of Programming*. Springer, 1987.

[9] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *LNM*, 188, pages 102–116. Springer, 1971.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting reuse in Event B development: Modularisation approach. In *ASM*, volume 5977 of *LNCS*, pages 174–188. Springer, 2010.

[12] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[13] R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, February 2011.