The original publication is available at http://dx.doi.org/10.1007/978-3-642-30885-7_16
Appeared in Proceedings of the ABZ2012 Conference © Springer-Verlag

Refinement by Interface Instantiation*

Stefan Hallerstede[1] and Thai Son Hoang[2]

[1] Aarhus University, Denmark
[2] ETH Zürich, Switzerland

Abstract. Decomposition is a technique to separate the design of a complex system into smaller sub-models, which improves scalability and team development. In the shared-variable decomposition approach for Event-B sub-models share external variables and communicate through external events which cannot be easily refined.
Our first contribution hence is a proposal for a new construct called interface that encapsulates the external variables, along with a mechanism for interface instantiation. Using the new construct and mechanism, external variables can be refined consistently. Our second contribution is an approach for verifying the correctness of Event-B extensions using the supporting Rodin tool. We illustrate our approach by proving the correctness of interface instantiation.

Keywords: Event-B, Decomposition, Refinement, External variables.

1 Introduction

Decomposition of a model into sub-models allows one to continue refining the sub-models independently of each other while preserving the properties of the full model. The decomposition method for Event-B proposed by Abrial [1] splits events between the sub-models. Variables are split correspondingly into external variables shared by the sub-models and internal variables private to each model. For all external variables, external events that mimic the effect of corresponding (internal) events of other sub-models have to be added. If we want to refine external variables, we have to provide a gluing invariant that is functional, say, $v = h(w)$ where v are the abstract variables and w the concrete variables. Abrial [1] also proposes to rewrite the external events with $v := h(w)$ so that concrete and abstract events are equivalent. Internal variables and internal events are refined as usual in Event-B [2].

We call a collection of external variables with the external invariant an *interface*. Modelling interfaces by marking the corresponding variables as being external and refining them by specifying functional invariants makes it difficult to decompose and refine a model repeatedly. Fig.1 illustrates the problem where a model M is decomposed three times and the resulting sub-models are refined. We are interested in the two sub-models M_1 and M_2 at the bottom. How do we find the shared external invariant?

* This research was carried out as part of the EU FP7-ICT research project DEPLOY (Industrial deployment of advanced system engineering methods for high dependability and productivity) http://www.deploy-project.eu.

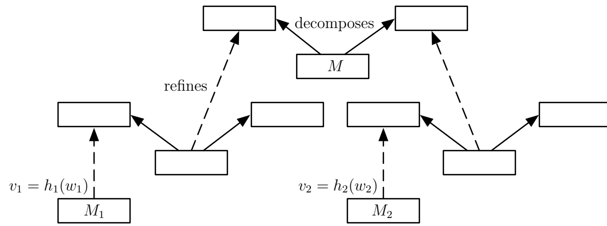

Fig. 1. Maintaining the external invariant of several sub-models

The lists of variables w_1, w_2, v_1 and v_2 are not necessarily disjoint. Let w be the list of variables occurring in w_1 or w_2 and v be the list of variables occurring in v_1 or v_2. We need to find one suitable external invariant $v = h(w)$ to be used in the sub-models M_1 and M_2. What is the shape of h? Furthermore, when refining M_2 we have to think about the necessary changes to M_1. As a consequence of the current situation, interfaces are refined to implementation level before decomposition. This complicates the use of decomposition on higher levels of abstraction. We would prefer a method where the necessary reasoning can be restricted to one place. The functional invariant h should be evident and easily maintainable also in the face of potential changes to the sub-models and the interfaces.

Using our approach of interface instantiation this can be done. Because we are treating instantiation like a special form of refinement, we can combine interface instantiation steps with refinement steps. This gives us some liberty in arranging complex refinements. We also encourage a decomposition style where a separate theory of interface instantiation is maintained. We think, that this contributes substantially to obtain models that are easier to understand and to modify. Interface instantiation supports a more incremental approach to decomposition because modifications that concern several components can be confined to only one place: the interface.

We call the very specific form of interface refinement that we use *interface instantiation*. To be useful, it should

(i) ease the proof effort compared to [1],
(ii) help to structure complex mixtures of decomposition and refinement,
(iii) work seamlessly with Event-B as it is. (It should not depend on translations.)

We argue by means of a case study that we have achieved this. The case study addresses a difficulty of relating Event-B refinement to Problem Frames elaboration [9] discussed in [5]. It has been composed from [11] and [5]. We have down-sized it in order to focus on the problem of the refinement of external variables, that is, the interfaces. We have a tool for decomposition [14] but we do not have implemented a software tool for interface instantiation. Instantiation of carrier sets has been implemented similarly internally in the ProB tool, in order to achieve better performance when model checking and constraint checking [5]. The case study as presented in [11] uses Problem Frames to achieve traceability of requirements. We have not used Problem Frames in this article because they are not required to explain interface instantiation. This also permits us to cast

the problem entirely in Event-B terminology. However, the proposed method of instantiation could be used with Problem Frames as employed in [5,11].

In the modularisation approach for Event-B presented in [8], the notion of interface has been used to capture software specifications using some interface variables and operations acting on these variables. The intention behind the use of interfaces is to separate specifications from their implementations. Our notion of interface is intended to provide efficient support for refining external variables following Abrial's decomposition method for system models. There was an earlier attempt at external variable refinement that is hinted at in the specification of the proof obligation generator for the Rodin tool [6]. This was considered too complicated and not feasible for large systems that are decomposed and refined repeatedly. We think, that our approach solves the problem. Poppleton [12] discusses external refinement based on Abrial's approach but also does not provide a practicable technique for doing so. The approach of modelling extensible records [4] also permits a form interface instantiation. A difficulty with using this approach is caused by the explicit mathematical model used for record representations and the need to specify always successor values for all fields of a record. However, extensible records could be used with our approach where it would appear useful. Behavioural interface refinement such as discussed in [13] addresses changing traces sub-models can exhibit, usually adding new events. It does not consider refinement of shared variables.

2 Event-B

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, and *theorems*, where carrier sets are similar to types [2]. Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, *theorems*, and *events*. Variables v describe the state of a machine. They are constrained by invariants $I(v)$. Theorems $L(v)$ describe consequences of the invariants, i.e., we have to prove $I(v) \Rightarrow L(v)$.

Events. Possible state changes are described by means of events. Each event is composed of a *guard* $G(t, v)$ and an *action* $A(t, v)$, where t are *parameters* the event may contain. We denote an event e by any t when $G(t, v)$ then $A(t, v)$ end in its most general form, or when $G(v)$ then $A(v)$ end if event e does not have parameters, or begin $A(v)$ end if in addition the guard equals *true*. A dedicated event of the third form is used for *initialisation*.

Assignments. The action of an event is composed of several *assignments*: $x :| \; Q(t, v, x')$, where x are some variables and $Q(t, v, x')$ a predicate. Variable x is assigned a value satisfying a predicate. Two variants of assignments are defined as follows: $x := B(t, v) \;\widehat{=}\; x :| \; x' = B(t, v)$ and $x :\in B(t, v) \;\widehat{=}\; x :| \; x' \in B(t, v)$, where $B(t, v)$ are expressions.

Refinement. A machine N can refine another machine M. We call M the *abstract* machine and N a *concrete* machine. The state of the abstract machine

is related to the state of the concrete machine by a *gluing invariant* $J(v, w)$ associated with the concrete machine N, where v are the variables of the abstract machine and w the variables of the concrete machine. Each event e of the abstract machine is *refined* by one or more concrete events f.

Decomposition. A machine M can be decomposed into several machines [1,7]. We limit the discussion here to the decomposition into two machines for the purpose of this article. Let M be a machine with variables x_1, x_3, x_5 and invariants $I(x_1, x_3, x_5)$, $I_1(x_1, x_3)$, $I_3(x_3)$ and $I_5(x_3, x_5)$. Furthermore, let e_1, e_2, e_4 and e_5 be events of M, accessing different sets of variables as follows.

$$\begin{array}{lll} \mathsf{e}_1 & \widehat{=} & \mathsf{any}\ t_1\ \mathsf{where}\ G_1(t_1, x_1)\ \mathsf{then}\ x_1 :\mid S_1(t_1, x_1, x_1')\ \mathsf{end} \\ \mathsf{e}_2 & \widehat{=} & \mathsf{any}\ t_2\ \mathsf{where}\ G_2(t_2, x_1, x_3)\ \mathsf{then}\ x_1, x_3 :\mid S_2(t_2, x_1, x_3, x_1', x_3')\ \mathsf{end} \\ \mathsf{e}_4 & \widehat{=} & \mathsf{any}\ t_4\ \mathsf{where}\ G_4(t_4, x_3, x_5)\ \mathsf{then}\ x_3, x_5 :\mid S_4(t_4, x_3, x_5, x_3', x_5')\ \mathsf{end} \\ \mathsf{e}_5 & \widehat{=} & \mathsf{any}\ t_5\ \mathsf{where}\ G_5(t_5, x_5)\ \mathsf{then}\ x_5 :\mid S(t_5, x_5, x_5')\ \mathsf{end} \end{array}$$

Machine M can be decomposed into two separate machines: M_1 with events e_1 and e_2; and M_5 with events e_4 and e_5. This is illustrated in Fig.2. As a result of the decomposition, M_1 has private variables x_1 and shared variables x_3. Invariants $I_1(x_1, x_3)$ and $I_3(x_3)$ can be distributed to M_1. The resulting sub-machine M_1 has two *internal* events e_1 and e_2 and one *external* event a_4 which abstracts[1] e_4 projected on the state containing only x_3: $\mathsf{a}_4 \ \widehat{=}\ \mathsf{any}\ t_4, x_5\ \mathsf{where}\ G_4(t_4, x_3, x_5)\ \mathsf{then}\ x_3 :\mid \exists x_5' \cdot S_4(t_4, x_3, x_5, x_3', x_5')\ \mathsf{end}$. Machine M_5 is similar to M_1, with two internal events e_4 and e_5; and an external event a_2 that abstracts e_2. It has the private variables x_5 and the shared variables x_3. Machines M_1 and M_5 can be developed independently with the constraints that the shared variables cannot be removed and the external events cannot be made more non-deterministic or less non-deterministic.

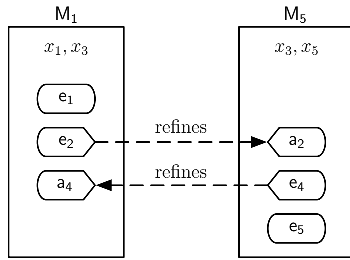

Fig. 2. Maintaining the external invariant of several sub-models

Note that invariant $I(x_1, x_3, x_5)$ are not copied to either M_1 or M_5. A possibility is to project also this invariant onto the corresponding state using existential quantifier. For example, the following can be added to M_1 as an invariant $\exists x_5 \cdot I(x_1, x_3, x_5)$.

3 Instantiation

Carrier set and constant instantiation. Contexts can be extended as usual in Event-B but we allow additionally to specify expressions to instantiate constants and carriers sets. Carriers sets must be instantiated by type expressions

[1] "a abstracts e" is the same as "e refines a".

$e(t)$ and constants can be instantiated by any expression $f(d)$.

```
context C                      context D
  sets s                         extends C with s = e(t) , c = f(d)
  constants c                    sets t
  axioms A(s, c)                 constants d
                                 axioms B(t, d)
```

The equalities specifying the instantiation are treated similarly to axioms. The abstract constants and carriers sets that are instantiated remain visible. By contrast, the instantiation proposed in [2] replaces constants and carrier sets in the instantiating context. Still, they are similar to [2]: The equations of the extends-clause are used to rewrite the abstract axioms. If this changes an axiom, that axiom must be *proved* to hold in the instantiating context. Otherwise, nothing needs to be proved. This ensures that instantiation itself does not introduce new facts. The *instantiation* proof obligation is $B(t, d) \Rightarrow A(e(t), f(d))$. In summary, conventional Event-B context extension is instantiation with identity. Only abstract axioms of C with instantiated constants need to be proved as theorems in D. The other axioms are preserved by extension.

Connecting machines to interfaces. Interfaces are declared in contexts and used in machines by connecting a machine to the interface. The machines must see the corresponding context:

```
context C                      machine M
  interface ii                   sees C
    fields m                     connects ii
    constraints P(m)
```

The constraints of an interface can refer to all constants and carrier sets of the surrounding context. In machine M the fields m are treated like variables and the constraints $P(m)$ like external invariants.

Interface instantiation. Interfaces can be instantiated by specifying equalities $m = h(n)$ for replacing fields of an abstract interface m by fields of a concrete interface n. The names on the right-hand side of the equation must not occur in the abstract interface.

```
context C                      context D
  interface ii                   extends C
    fields m                     interface jj instantiates ii with m = h(n)
    constraints P(m)               fields n
                                   constraints Q(n)
```

The expression h is often composed of pair-expressions "$\cdot \mapsto \cdot$". Interfaces are not associated with proof obligations. The constraints $P(m)$ of ii are contained in interface jj as specified by the instantiation $m = h(n)$, that is, they become $P(h(n))$. Similarly to machine variables, field names of interfaces cannot be reintroduced. Similarly to machine invariants, constraints are accumulated in by instantiation: the constraints of interface jj are $Q(n) \wedge P(h(n))$.

External event refinement. Using interface instantiation we permit refinement of external events. Consider the following external event e operating on the external variables x and its refinement f operating on the external variables y. The refinement of external variables is captured by the following relationship $x = h(y)$. Note that external events does not refer to any internal variables: it can only refer to external variables of the corresponding model.

$$\mathsf{e} \mathrel{\widehat{=}} \mathsf{any}\ t\ \mathsf{when}\ G(t, x)\ \mathsf{then}\ x :\mid S(t, x, x')\ \mathsf{end}$$
$$\mathsf{f} \mathrel{\widehat{=}} \mathsf{any}\ u\ \mathsf{when}\ H(u, y)\ \mathsf{with}\ W(t, u, x, y, y') \land x' = h(y')\ \mathsf{then}\ y :\mid R(u, y, y')\ \mathsf{end}$$

where $W(t, u, x, y, y') \land x' = h(y')$ is the witness for the refinement of e by f. It incorporates the refinement of external variables with function h.

Beside the proof obligations to prove that f is a refinement of e, we also need to prove that f is refined by e. The idea here is to prove the latter using the same given witnesses. The proof obligations are as follows (for clarity, we omit reference to possible abstract invariant $I(x)$ and other concrete invariant $J(x, y)$, which should be in the assumption of the proof obligations).

Witness feasibility:

$$x = h(y) \land G(t, x) \land S(t, x, x') \Rightarrow (\exists u, y' \cdot W(t, u, x, y, y') \land x' = h(y'))$$

In the case that h is a bijective function, the existence of y' is hence trivial, and the proof obligation can be rewritten as follows.

$$x = h(y) \land G(t, x) \land S(t, x, x') \land x' = h(y') \Rightarrow (\exists u \cdot W(t, u, x, y, y'))$$

Guard weakening:

$$x = h(y) \land G(t, x) \land S(t, x, x') \land W(t, u, x, y, y') \land x' = h(y') \Rightarrow H(u, y)$$

(Co-)Simulation:

$$x = h(y) \land G(t, x) \land S(t, x, x') \land W(t, u, x, y, y') \land x' = h(y') \Rightarrow R(u, y, y')$$

Note that *invariant preservation* for the refinement of f by e can be derived from the *invariant preservation* for the refinement of e by f and the fact that we use the same witnesses.

4 Case study: modelling of a cruise control system

We present interface instantiation by means of a model of a cruise control system. A cruise control systems permits the driver of a car to select a target speed that the vehicle should attain. The system will try to maintain a vehicle speed as close as possible to the target speed. Note, that our main interest is to discuss interface instantiation. So we will only discuss the function of the cruise control system as far as necessary for that discussion. We have modeled the system using the Rodin tool [3], emulating instantiation similarly to the approach of [5]:

interfaces are represented syntactically by a lexical convention and carrier set instantiation is modelled by suitable bijections.

We want to implement a cruise control system sy0 by the three components: the controller sy4cr, the engine sy4even and the exterior sy1evsi. Fig. 3 shows the components and their interfaces.

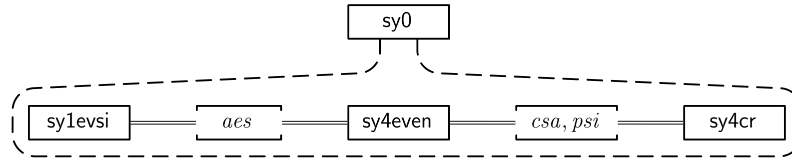

Fig. 3. Architecture of the system in terms of components and interfaces

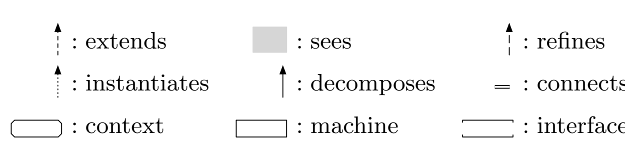

Fig. 4. Legend of used symbols

The symbols used in this figure and later figures are listed in Fig. 4. The implementations of the controller and the engine are connected by means of two interfaces: concrete speed and acceleration, *csa*, and internal pedal signals passed on from the exterior, *psi*. The interface to the exterior, *aes*, is kept abstract in the implementation.

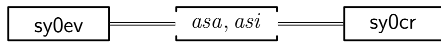

Fig. 5. Abstract components and their abstract interfaces

More abstract system models should not be forced to use the interfaces *csa* and *psi* but permit abstractions thereof (Fig. 5). The details of the interfaces should be introduced step by step, introducing the abstract interfaces *asa* and *asi* first. We prefer to refine the controller and the engine but keep the exterior abstract at first. We do not want to decide on all interfaces before decomposing system sy0: we have not decided yet on the shape of the implementation of component sy1evsi and of interface *aes*. Interface *aes* could be used to implement an interface to the exterior or it could be used for animation and visualisation [10], for instance. The problem we face is to fit the abstract components of Fig. 5 between sy0 and the implementation in Fig. 3.

4.1 The full model: refinement, decomposition and instantiation

We present an overview of the full model and discuss specific issues in subsequent sections. Fig. 6 shows the details of the development outlined in Fig. 3. We do not discuss all aspects of the development but focus on the following three:

Section 4.2: Decomposition: introducing interfaces

Section 4.3: Mixing instantiation and refinement
Section 4.4: Repeated instantiation

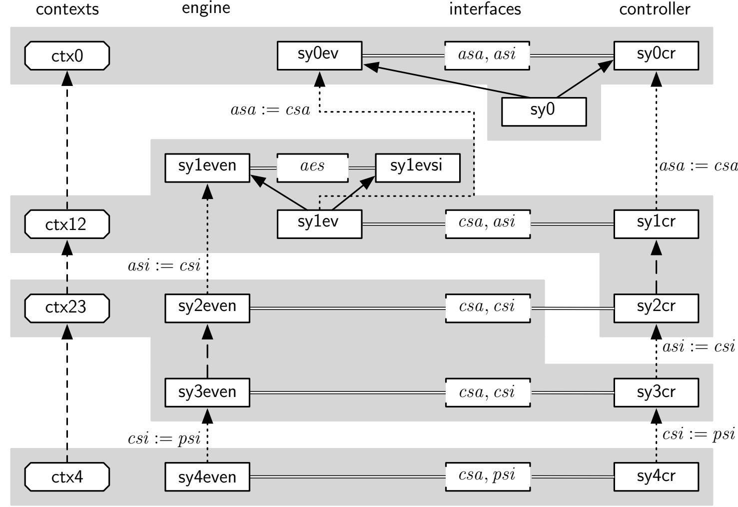

Fig. 6. Overview of system model

The separate contexts ctx0, ctx12, ctx23 and ctx4 correspond to the accompanying instantiations of carrier sets and constants.

4.2 Decomposition: introducing interfaces

Abstract model. The model sy0 from which we start the development declares variables sig, cs, vs, md, ts, acc modelling external signals, internal control signals, vehicle speed, control mode, target speed and acceleration. It does not contain any interfaces. This means we can refine this model in the usual way. Context ctx0 declares constants ES, CS, VS, VA, VRA, etc, modelling external signals, control signals, vehicle speed, vehicle acceleration, restricted vehicle acceleration. It postulates the axiom

$$VRA \subseteq VA \tag{1}$$

We have invented constant VRA to make the invariant more interesting. The constants determine the possible values of the variables by means of the invariant of sy0:

$$sig \in ES \wedge cs \in CS \wedge \ldots \wedge md \in \{C, AC, NC\} \wedge (md = C \Rightarrow acc \in VRA)$$

The constant C models "cruise control active"; AC models "change of target speed"; NC models "cruise control not active". The carrier sets, e.g., K of CS or S of VS, are not used in the machine. The reason for this is that they can

only be instantiated by type expressions. However, the more common case is that we need to instantiate by some more constrained set. See, for example, the instantiations of C, AC and NC in Section 4.3.

Events of the abstract model. We discuss three of the events of sy0: event *chm* ("change mode") models an internal state change of the controller,

$$\begin{array}{l}\mathsf{event}\ chm\ \widehat{=}\\ \quad \mathsf{begin}\ md :| \ md' \in \{C, AC, NC\} \wedge (md' = C \Rightarrow acc \in VRA)\ \mathsf{end}\,;\end{array}$$

event *chaac* ("change acceleration in mode AC") models output to the engine,

$$\mathsf{event}\ chaac\ \widehat{=}\ \mathsf{when}\ md = AC\ \mathsf{then}\ acc :\in VA\ \mathsf{end}\,;$$

event *chcs* ("change control signals") models input from the engine,

$$\mathsf{event}\ chcs\ \widehat{=}\ \mathsf{begin}\ cs := fcs(sig)\ \mathsf{end}\,.$$

It would be tempting to specify in the abstract event *chcs* the assignment $cs := sig$. However, this asserts that cs and sig have the same type. Once the system is decomposed, we would have to refine them in the same way. To avoid this, we have introduced function fcs mapping from the type of sig to the type of cs. Models always need to be prepared for decomposition. Our method of instantiation does not change this.

Decomposition of the abstract model. Decomposing sy0 into sy0ev and sy0cr we have to introduce interfaces asa and asi:

$$\begin{array}{ll}\mathsf{interface}\ asa & \mathsf{interface}\ asi\\ \quad \mathsf{fields}\ cs & \quad \mathsf{fields}\ vs, va\\ \quad \mathsf{constraints}\ cs \in CS & \quad \mathsf{constraints}\ vs \in VS \wedge va \in VA\end{array}$$

Machine sy0ev has one internal variable sig and connects to the two interfaces asa and asi. Machine sy0cr connects to the same interfaces and has two internal variables ts and md. We split the events in the usual way depending on which variables and fields the events refer to. Except for the use of interfaces the decomposition method of [1] works as before.

4.3 Mixing instantiation and refinement

Instantiation. We refine sy0cr by sy1cr by instantiating interface asa by csa while refining variable md by variable nd. The constraints and instantiation equalities of csa become part of the gluing invariant of sy1cr. The abstract constants VS, VA and VRA are instantiated by integer ranges: $VS = mS \mathrel{..} MS$, $VA = mA \mathrel{..} MA$ and $VRA = mRA \mathrel{..} MRA$ constrained by axioms

$$\ldots \wedge mA \leq mRA \wedge mRA \leq MRA \wedge MRA \leq MA \wedge \ldots \qquad (2)$$

To satisfy the instantiation proof obligation we have to verify that (2) implies (1). For clarity we introduce a new name for the interface containing the instantiated constants:

$$\mathsf{interface}\ csa\ \mathsf{instantiates}\ asa$$

Machine sy1cr and sy1ev now both need to be connected to interface csa replacing asa. The machine also need to see the extended context ctx12.

Refinement. Variable md is itself refined by instantiating the constants C, AC and NC, using the gluing invariant $nd \in md$, and constant instantiations $C = \{CRS, RES\}$, $AC = \{ACC, DEC\}$, $NC = \{OFF, ERR, REC\}$. Note, how closely constant instantiation and refinement are linked in the refinement of md. The type of the abstract variable md has been instantiated such that the gluing invariant becomes simply $nd \in md$.

4.4 Repeated instantiation

First instantiation. Continuing the development from sy1cr and sy1ev, we first instantiate interface asi by csi

interface csi instantiates asi with $cs = (ps \mapsto cis \mapsto is)$
fields ps, cis, is
constraints $ps \in PS \wedge cis \in CIS \wedge is \in IS$

where PS is a constant of context ctx32. The context also declares two constants PSE and PSS such that $PSE \subseteq PS \wedge PSS \subseteq PS \wedge PSE \cap PSS = \varnothing$. This is used for a first refinement of event chm into two events $chme$ and $chmn$:

event $chme$ refines chm $\widehat{=}$
when $ps \in PSS \cup PSE$ then $nd :\in \{ERR, REC\}$ end
event $chmn$ refines chm $\widehat{=}$
when $ps \notin PSS \cup PSE$
then $nd :| \; nd' \in \{CRS, RES\} \Rightarrow acc \in mRA \,..\, MRA$ end

Second instantiation. Subsequently we instantiate csi by psi

interface psi instantiates csi with $ps = (pbp \mapsto pbe \mapsto pcp \mapsto pce \mapsto pae)$
fields $pbp, pbe, pcp, pce, pae, cis, is$
constraints $pbp \in \mathbb{B} \wedge pbe \in \mathbb{B} \wedge pcp \in \mathbb{B} \wedge pce \in \mathbb{B} \wedge pae \in \mathbb{B}$

We instantiate the constants PSE and PSS

$$PSE = \{bp \mapsto be \mapsto cp \mapsto ce \mapsto ae | \mathbf{T} \in \{be, ce, ae\}\}$$
$$PSS = \{bp \mapsto be \mapsto cp \mapsto ce \mapsto ae | \mathbf{T} \notin \{be, ce, ae\} \wedge \mathbf{T} \in \{bp, cp\}\}$$

and prove $PSE \cap PSS = \varnothing$ as postulated above.

event $chmrb$ refines $chme$ $\widehat{=}$
when $\mathbf{T} \notin \{pbe, pce, pae\} \wedge \mathbf{T} = pbp$ then $nd := REC$ end
event $chmrc$ refines $chme$ $\widehat{=}$
when $\mathbf{T} \notin pbe, pce, pae \wedge \mathbf{T} = pcp$ then $nd := REC$ end
event $chmbe$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pbe$ then $nd := ERR$ end
event $chmce$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pce$ then $nd := ERR$ end
event $chmae$ refines $chme$ $\widehat{=}$ when $\mathbf{T} = pae$ then $nd := ERR$ end

This last instantiation is much more concise than the refinement suggested in [5]. We can avoid a lot of the overhead that is usually incurred by using refinement emulating instantiation. Not having dedicated tool support yet, the most elaborate proof of this development occurred when instantiating VA and VRA by integer intervals. With instantiation support in place this would have been trivial using the fact that $VRA = mRA \mathbin{..} MRA$. The difficulty in the proof of refinement emulating instantiation is caused by the need to use a bijection $\iota \in mRA \mathbin{..} MRA \rightarrowtail\!\!\!\!\twoheadrightarrow VRA$ so that the equation for the instantiation becomes $VRA = \iota[mRA \mathbin{..} MRA]$.

5 Correctness

We have used the Rodin tool for verifying the correctness of interface refinement. First we present a technique for verifying extensions of Event-B. We believe that it is useful beyond the use in this article for verifying the correctness of interface instantiation.

A technique for proving Event-B extensions correct. The general idea is to encode a generic model using the Rodin tool, and illustrating the extended method using the generic model. Typically, the correctness of an extension can be stated as follows: assume the consistency of some input model, then prove the consistency of some resulting model. Using the models generated by the tool, consistency of the input model are represented by the proof obligations associated with the model. To turn them into assumptions for our reasoning, we add these proof obligations as axioms to the context. Using the axioms, we prove the consistency of the resulting model.

An example of our approach is as follows. Let M be a machine with variables x, invariant $I(x)$, and an event e as follows.

$$\mathsf{e} \;\widehat{=}\; \mathsf{any}\ t\ \mathsf{where}\ G(t,x)\ \mathsf{then}\ x :\mid S(t,x,x')\ \mathsf{end} \quad .$$

First, we model the type of variables x and parameters t using some carrier sets X and T. Subsequently, I, G and S can be declared as constants with appropriate type, i.e. $I \in \mathbb{P}(X)$, $G \in \mathbb{P}(T \times X)$, and $S \in \mathbb{P}(T \times X \times X)$. The machine M is encoded accordingly using the above context, where predicates are translated using membership ($\in$) operator[2]. For example, the invariant $I(x)$ is translated as $x \in I$. Event e hence becomes

$$\mathsf{e} \;\widehat{=}\; \mathsf{any}\ t\ \mathsf{where}\ t \mapsto x \in G\ \mathsf{then}\ x :\mid t \mapsto x \mapsto x' \in S\ \mathsf{end}$$

The proof obligation stating that e maintains invariant $I(x)$, i.e., $I(x) \wedge G(t,x) \wedge S(t,x,x') \Rightarrow I(x')$ is encoded as an axiom in the context as follows $\forall t,x,x' \cdot x \in I \wedge t \mapsto x \in G \wedge t \mapsto x \mapsto x' \in S \Rightarrow x' \in I$.

Assume that N is a correct refinement of M, retaining the abstract variables x. In N, abstract event e is refined by concrete event f where parameter t is also retained.

[2] This is a well-known technique to model predicate constants or variables in Event-B using first-order logic.

$$\mathsf{f} \ \widehat{=} \ \mathsf{any}\ t\ \mathsf{where}\ H(t,x)\ \mathsf{then}\ x :| \ S(t,x,x')\ \mathsf{end}$$

The fact that f is a correct refinement of e is captured by the *guard strengthening* and *simulation* proof obligations which are encoded as the following axioms: $\forall t,x \cdot x \in I \wedge t \mapsto x \in H \Rightarrow x \in G$ and $\forall t,x,x' \cdot x \in I \wedge t \mapsto x \in G \wedge t \mapsto x \mapsto x' \in R \Rightarrow t \mapsto x \mapsto x' \in S$.

A property of refinement is the preservation of invariance properties, i.e. I should be also an invariant of the concrete model, in particular maintain by the concrete event f. This can be stated and proved as a *theorem* in the context $\forall t,x,x' \cdot x \in I \wedge t \mapsto x \in H \wedge t \mapsto x \mapsto x' \in R \Rightarrow x' \in I$.

Correctness of interface instantiation. Using the proof method, we prove the correctness of interface instantiation as follows. Our initial machine is M as described at the end of Section 2. We decompose M into M_1 and M_5, sharing the interface U. The abstract interface U encapsulates the shared variable x_3 with invariant $I_3(x)$, and subsequently instantiated by some concrete interface V containing concrete variable y_3 as follows.

interface U	interface V instantiates U with $x_3 = h_3(y_3)$
fields x_3	fields y_3
constraints $I_3(x_3)$	constraints $J_3(y_3)$

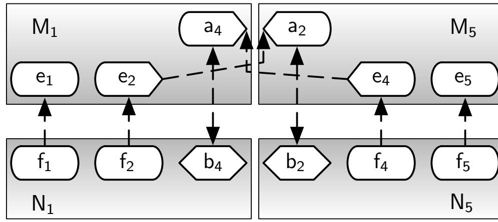

Fig. 7. Decomposition and events refinement

At the same time, M_1 and M_5 are refined into N_1 and N_5, relying on the interface instantiation. To be more precise, in N_1, internal event e_1 and e_2 are refined by f_1 and f_2, respectively. Furthermore, external event a_4 is refined *equivalently* (see Section 3) to b_4. Similarly, in N_5, f_4 and f_5 are the refinement of internal events e_4 and e_5, respectively, and b_2 is the refinement of external event a_2. The refinement relationships between the events can be depicted in Figure 7.

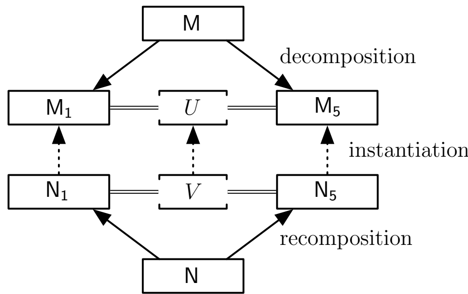

Fig. 8. Decomposition, interface instantiation, and refinement

The composition machine N comprises of internal events f_1, f_2 from N_1, and f_4 and f_5 from N_5. The summary of decomposition and interface instantiation approach is shown in Figure 8.

The correctness of our technique is guaranteed by proving that the composition N of N_1 and N_5 is indeed a refinement of the original model M, using the assumption that N_1 and N_5 refines M_1 and M_5, respectively, as described earlier. The key important aspect for the correctness of our approach is that

external events are refined equivalently. This fact guarantees that the refinement relationship between a pair of internal/external events is maintained. For example, we have that the internal event f_4 is a refinement of the corresponding external event b_4.[3]

6 Conclusion

We propose in this paper the notion of interface and interface instantiation for shared-variable decomposition in Event-B. An interface is a collection of external variables and their properties which can be shared between different sub-model after a decomposition. Interface instantiation combines instantiation of carrier sets and constants with functional refinement of external variables. The encapsulation of external variables using interface offers us some flexibility in structuring the development using complex refinement and decomposition. In particular, we provide a practical method for refining external variables which is currently quite cumbersome [1].

The novelty of our approach is in the refinement of external events: we define additional proof obligations to ensure that the external events are refined *equivalently*. By contrast, in [1] equivalence is achieved by syntactical means replacing occurrences of abstract variables v by concrete terms $h(w)$. The proof obligations of our approach are similar to the standard proof obligations, even using the same refinement witnesses for proving the equivalence. We have presented a general technique for proving correctness of Event-B extensions, and showed how this is used to demonstrate the soundness of our approach. We illustrated the method by an industrial case study modelling a cruise control system.

For future work, we want to develop a theory of interface instantiation. In particular we intend to investigate the idea of having different instantiation branches of interfaces that are joined ultimately so that all machines of a model agree on their interfaces. The idea is illustrated in Fig. 9. In the figure, an abstract interface U containing two abstract fields x_1 and x_2 is used by M_1 and M_2. Subsequently, U is instantiated by V_1 where x_1 is replaced by y_1 and x_2 is retained. At the same time, M_1 is refined into N_1 using V_1. Similarly, M_2 is refined by N_2 using V_2. Finally, V, an instantiation of both V_1 and V_2 can be use to refine N_1 and N_2 into O_1 and O_2, respectively. What this lattice of interfaces

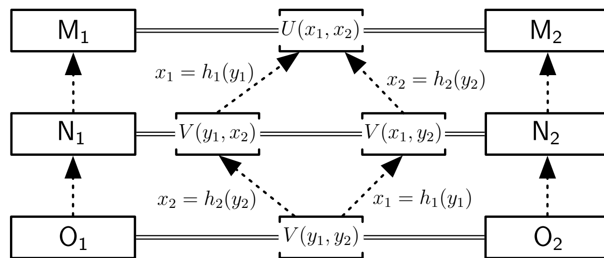

Fig. 9. A lattice of interfaces

[3] The development is available at `http://deploy-eprints.ecs.soton.ac.uk/364/`

allows us to do is to have different order of instantiating the fields of an abstract interface in an individual sub-model. In particular, we actually abandon the compositionality of the intermediate machines (here N_1 and N_2), only to re-establish it later for the final machines O_1 and O_2, by connecting them to the same interface V.

Finally, we are looking at extending the Rodin tool [3] to support the notion of interface and interface instantiation.

References

1. J.-R. Abrial. Event-B: Structure and Laws. 2005.
2. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
3. 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.
4. N. Evans and M. J. Butler. A Proposal for Records in Event-B. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *LNCS*, pages 221–235. Springer, 2006.
5. R. Gmehlich, K. Grau, S. Hallerstede, M. Leuschel, F. Lösch, and D. Plagge. On fitting a formal method into practice. In S. Qin and Z. Qiu, editors, *ICFEM*, volume 6991 of *LNCS*, pages 195–210. Springer, 2011.
6. S. Hallerstede. The Event-B Proof Obligation Generator, 2005.
7. T. S. Hoang and J-R. Abrial. Event-b decomposition for parallel programs. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *ABZ2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2010.
8. 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.
9. M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., 2001.
10. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising event-B models with B-motion studio. In M. Alpuente, B. Cook, and C. Joubert, editors, *FMICS*, volume 5825 of *LNCS*, pages 202–204. Springer, 2009.
11. F. Loesch, R. Gmehlich, K. Grau, C. B. Jones, and M. Mazzara. DEPLOY Deliverable D19: Pilot Deployment in the Automotive Sector.
12. M. Poppleton. The composition of event-B models. In E. Börger, M. J. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *LNCS*, pages 209–222. Springer, 2008.
13. S. Schneider and H. Treharne. Changing system interfaces consistently: A new refinement strategy for $CSP||B$. *Sci. Comput. Program.*, 76(10):837–860, 2011.
14. R. Silva, C. Pascal, T. S. Hoang, and M. Butler. Decomposition tool for event-B. *Softw, Pract. Exper*, 41(2):199–208, 2011.