Invariant Guided System Decomposition

Richard Banach

School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, UK
banach@cs.man.ac.uk

Abstract. We re-examine the problem of decomposing systems in Event-B. We develop a pattern for cross-cutting events and invariants that enables the core dependencies in multi-machine systems to be tracked. We give the essential verification conditions.

Keywords: System Decomposition, Cross-Cutting Invariants, Event-B

1 Introduction

In top down model based development methodologies, especially the B-Method, the issue of composition and decomposition of (sub)systems has received a lot of interest. See e.g. [1, 3, 2, 5]. For us, the main issue may be illustrated in a simple example.

Suppose there is a machine M with variables x, y. Suppose M needs to be partitioned into two machines, $M1$ and $M2$. Suppose that x needs to go into $M1$ and y needs to go into $M2$. Suppose that there is an invariant of M involving both variables, $InvM(x,y)$. If the partitioning is to go ahead, what are we to do about $InvM(x,y)$?

Sometimes it is suggested that an invariant like $InvM(x,y)$ might be replaced by $InvM1(x) \equiv (\exists\, y \bullet InvM(x,y))$ in $M1$, say. However although $InvM(x,y) \Rightarrow InvM1(x)$, the converse does not hold. Therefore, recognising that $InvM$ and $InvM1$ are inequivalent, if $InvM(x,y)$ is a critical safety invariant, then the suggested partitioning strategy would render the system incapable of discharging its most important duty. The usual approach if $InvM$ is important enough, is simply to not partition. However, such an approach does not scale.

The remainder of the paper is as follows. Section 2 introduces our approach to decomposition in generic terms. Section 3 covers verification issues, while Section 4 covers machine decomposition. Section 5 looks at refinement. Section 6 concludes.

2 Variable Sharing via INTERFACEs

We note that in typical embedded systems, connections are invariably unidirectional, often mirroring physical connections such as wires. We exploit this unidirectionality to design a methodology for handling a useful class of invariants that cut across subsystem boundaries. We first introduce a concept of INTERFACE, rooted in the work of Hallerstade and Hoang [4], which we extend, just enough to achieve what we desire.

An interface is a syntactic construct that declares some variables, and (going beyond [4]), some invariants that interrelate them, and their initialisations. Any machine that needs to access any of these variables must refer to the interface. The interface mechanism is the only permitted way for more than one machine to have access to the same set of variables. Our use of interfaces is based on the following principles.

Consider a set of variables $\mathcal{V}$, a set of invariants $\mathcal{I}$ that mention some of those variables (and no others), and a set of events $\mathcal{E}$ that read and update some of those variables (and no others). Suppose the set of variables can be partitioned into subsets $A, B, C \ldots$, such that for every invariant $Inv \in \mathcal{I}$:

[•1] either all variables mentioned in *Inv* belong to some subset, eg. A;
[•2] or the invariant *Inv* is of the form $U(u) \Rightarrow V(v)$, where there are distinct subsets of the partition A and B say, such that u and v refer to variables in A and B respectively.

We call these type **[1]** and type **[2]** invariants respectively (t1i and t2i). For a t2i, the A and B subsets are the local and remote subsets (containing the local variables u and remote variables v). We observe that unless a system already consists simply of two unconnected, completely independent subsystems, in which all properties split into a conjunctionof properties of the two subsystems, there will be, in general, an infinity of properties that couple the two subsystems nontrivially. Referring to the discussion of the Introduction, the problem of what to do about cross-cutting invariants is unavoidable. Our thesis is that, in the kind of embedded systems we spoke of, the unidirectionality of the connections between subsystems implies that t2is are adequate to capture a sufficiently rich class of inter-subsystem properties for practical use.

Henceforth we restrict to collections of variables/invariants/events conforming to these restrictions, calling them pre-systems. Note that *any* collection of variables and invariants is as a pre-system with a sufficiently coarse variable partition, e.g. a singleton partition. We can organise a pre-system into machines and interfaces as follows.

Every subset of variables of the partition can consist of variables that, exclusively:

[•3] either are declared as the variables of a single machine;
[•4] or are declared as the variables of a single interface.

Each interface:

[•5] must contain all the type **[1]** invariants that mention any of its variables;
[•6] may contain type **[2]** invariants for which the interface's variables are in the local subset; in each such case the interface must contain a READS *ReadItf* declaration for the interface *ReadItf* that contains the remote variables.
[•7] may contain REFERS *RefItf* declarations, whenever any of its variables are the remote variables of a type **[2]** invariant declared in an interface *RefItf*.

Each machine:

[•8] may declare the variables belonging to a subset of the partition as local (i.e. unshared) variables;
[•9] may contain one or more CONNECTS *Itf* declarations giving access to the variables of the interface;
[•10] may contain one or more READS *Itf* declarations giving read-only access to the variables of the interface;
[•11] must contain all the type **[1]** invariants that mention any of its local variables;

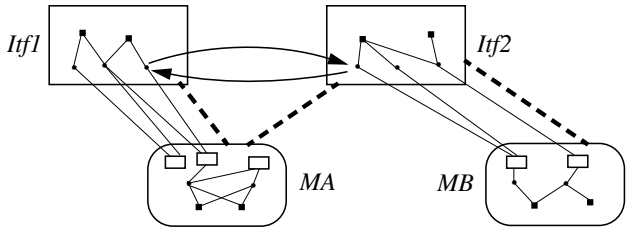

Fig. 1. An illustration of the constraints **[•1]**–**[•15]**.

Each event:

[•12] may read and update variables that are declared locally in the machine containing the event, or that are introduced via CONNECTS *Itf* declarations in the machine containing the event;

[•13] may read (in its guards or in the expressions that define update values) variables that are introduced: either via READS *ReadItf* declarations in the machine containing the event, or via READS *ReadItf* or REFERS *RefItf* declarations contained in an interface *Itf* that the machine containing the event CONNECTS (to).

[•14] must preserve all invariants that are declared in the machine that contains it, or that are declared in any CONNECTS *Itf* declarations of the machine, or that are contained in any READS *ReadItf* or REFERS *RefItf* declarations contained in an interface *Itf* that the machine containing the event CONNECTS (to).

Each invariant:

[•15] must be contained in the interface or machine which declares all its variables (if it is a type **[1]** invariant), or must be contained in the interface which declares its local variables (if it is a type **[2]** invariant).

By a system, we mean a collection of machines satisfying **[•1]**–**[•15]** above. We note that the keywords we introduced, CONNECTS, READS, REFERS, have no semantic connotations other than the ones we mentioned. We can see fairly readily that in a system, verifying that all the invariants are preserved by all event executions (provided the initial state satisfies them all), can be readily accomplished using verification conditions that depend on information that is easily located from the syntactic context of the event, namely, from the interfaces explicitly mentioned in the machine that defines the event. We examine verification conditions in more detail in the next section.

In Fig. 1 we show an illustration of the constraints **[•1]**–**[•15]**. Dots represent variables, while small squares represent type **[1]** invariants. Small rectangles represent events. Events and invariants are connected to the variables they involve by thin lines. Interfaces are large rectangles containing the variables and invariants they encapsulate — there are two in Fig. 1, *Itf*1 and *Itf*2. Machines are large rounded rectangles, containing their events and local variables — again there are two, *MA* and *MB*. The CONNECTS relationship is depicted by thick dashed lines. Finally, type **[2]** invariants are represented by arrows from the local to the remote interface.

3 Verification of Type [2] Invariants

In this section we focus on the verification of nontrivial t2i invariants, assuming that t1i invariants can be handled unproblematically by reference to the relevant interfaces during verification. (The same applies to an event that must maintain a t2i if it can access and update variables in *both* relevant interfaces (simultaneously).)

Consider a t2i $(*) \equiv U(u) \Rightarrow V(v)$, where u and v belong to different interfaces. We prime after-state expressions generically, thus: $(*') \equiv U'(u') \Rightarrow V'(v')$. We write the events of interest as $EvXYZ$ where $X, Y, Z \in \{U, V\}$. This means that the guard g_{UVV} of $EvUVV$ mentions the variables u, v and the before-after relation BA_{UVV} of $EvUVV$ updates variable v. The shorter notation $EvUV$ means that the guard mentions only u and the update is to variable v alone.

We assume that for events $EvUU$ and $EvVV$, verification would be restricted to variables u and v of invariant $(*)$ respectively, while for $EvUVU$ and $EvUVV$, both parts of $(*)$ could participate in verification, since both sets of variables are read via the relevant interfaces. Read access to additional variables is obviously harmless and is not considered further.

Theorem 1. *Assuming that initial states are invariant, and that all events preserve all type **[1]** invariants declared locally and in CONNECTS Itf declarations on reachable states, the following proof obligations (POs) are sufficient to preserve reachable invariance for type **[2]** invariants.*

$$EvUVU : g_{UVU}(u,v) \wedge \neg V(v) \Rightarrow g_{UVU}(u,v) \wedge \neg U(u) \Rightarrow BA_{UVU}(u,u') \Rightarrow \neg U(u')$$
(obvious analogue for EvVU) (1)

$$EvUU : g_{UU}(u) \wedge \neg U(u) \Rightarrow BA_{UVU}(u,u') \Rightarrow \neg U(u') \quad (2)$$

$$EvUVV : g_{UVV}(u,v) \wedge U(u) \Rightarrow g_{UVV}(u,v) \wedge V(v) \Rightarrow BA_{UVV}(v,v') \Rightarrow V(v')$$
(obvious analogue for EvUV) (3)

$$EvVV : g_{VV}(v) \wedge V(v) \Rightarrow BA_{VV}(v,v') \Rightarrow V(v') \quad (4)$$

The above gives a selection of POs which can be used for verifying the preservation of cross-cutting invariants of the t2i kind that we have considered, based on the occurrences of the relevant variables in the events that access those variables.

4 Machine Decomposition

The account so far permits us to assemble a large system by composing a number of machines together via a collection of interfaces that obey **[•1]**–**[•15]**. Equally interesting though for the B-Method in general, is the problem of the *decomposition* of a machine into a collection of smaller (sub)machines $M_1 \ldots M_k$, the development of which can subsequently be pursued (at least relatively) independently. We examine this issue now.

We approach the decomposition problem by positing that decomposition should be a syntactic manipulation whose correctness ought to be demonstrable generically. In this light, the principle constraining decompositions of a machine can be described as follows:

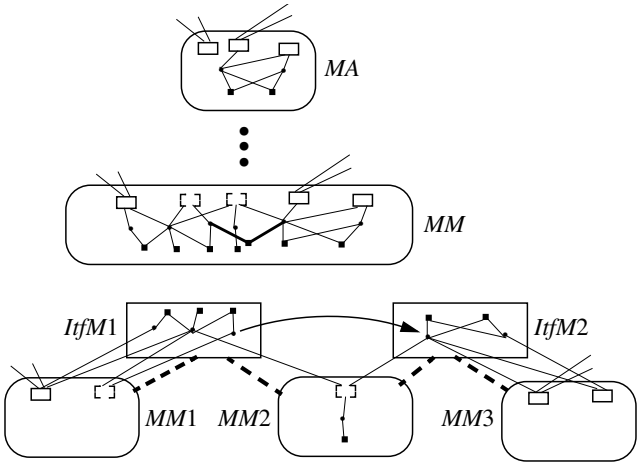

Fig. 2. An illustration of the decomposition mechanism. *MA*, refined to a larger machine *MM*, is decomposed into smaller machines and interlinking interfaces.

[•16] Regarding the variables and invariants (and events) declared in a machine M as a pre-system (but not including the variables or invariants of any interface accessed by M), any decomposition of M into submachines and interfaces is considered valid provided: firstly, it conforms to restrictions **[•1]**–**[•15]**; secondly, any submachine M_j that includes an event of M that uses a variable accessed (directly or indirectly) via an interface of M, must access the same interface appropriately.

It is clear that adhering to **[•16]** refines the partition of variables when M is part of a larger system already adhering to **[•1]**–**[•15]**, without spoiling **[•1]**–**[•15]** overall.

Fig. 2 shows the decomposition mechanism at work. Machine *MA* from Fig. 1 is first refined to a larger machine *MM*, containing more local variables and invariants, as well as some new events shown using broken small rectangles. One new invariant is connected to its variables using slightly thicker lines. Machine *MM* is now decomposed into a collection of smaller machines and interfaces, $MM1$, $MM2$, $MM3$, and $ItfM1$, $ItfM2$. The connections from *MA* events to previously existing interfaces are retained, while the decomposition of the new ingredients conforms to constraints **[•1]**–**[•15]**. The invariant connected using slightly thicker lines becomes a type **[2]** invariant with $ItfM1$ and $ItfM2$ as its local and remote interfaces respectively (on the presumption that it was of the correct syntactic shape at the outset).

5 Refinement

We turn to the crucial issue of refinement. As for decomposition, there is a key guiding principle behind the way that refinement is handled in our scheme.

[•17] The variables of an interface *Itf* must be refined to the variables of its refining interface *ItfR* via a retrieve relation that mentions only the variables of *Itf* and *ItfR*.

[•18] The variables of a machine M must be refined to the variables of its refining machine MR via a retrieve relation that mentions only the variables of M and MR.

The independence of refinement of machines and interfaces prevents the inadvertent falsifying of refinement relations in situations such as the following.

Suppose each of M_1 and M_2 CONNECTS Itf; these constructs being refined to M_1R, M_2R and $ItfR$ respectively. Suppose the joint invariant of the M_2 to M_2R refinement involves the variables of Itf and $ItfR$ too. Then when concrete machine M_1R executes an event, faithful to some abstract event of M_1, there is no guarantee that the new state in M_1 and M_1R and Itf and $ItfR$ still satisfies the joint invariants of M_2 and M_2R via the coupled joint invariants linking the state in M_2 and M_2R to the state in Itf and $ItfR$.

Adhering to **[•17]**–**[•18]** though, it is easy to see that the problem described cannot arise. The decoupling of variables of M_2 and M_2R on the one hand, from those of Itf and $ItfR$ on the other, means that when the variables of Itf and $ItfR$ change at the behest of M_1 and M_1R, the invariants linking the M_2 and M_2R variables remain true.

6 Conclusions

In this paper we have proposed, rather tersely, an Event-B decomposition scheme inspired by the INTERFACE idea of [4]. This was broadly in the shared variables tradition, but was driven primarily by the structure of a system's invariants. Although ostensibly a shared variable approach, there are strong influences from the shared events approach too, since a key feature of both ours and the shared events approach is the desire to communicate values between machines. In this brief treatment, we just gave a minimal description of the technical details of our approach, of which a kind of pattern for cross-cutting events and invariants was the key element, and we outlined the requisite verification machinery. In a more extended treatment, we will be able to describe the mechanisms more fully, we will be able to formulate the statements as theorems, and, crucially, we will be able to illustrate the technique using examples and case studies.

References

1. Abrial, J.R.: Event-B: Structure and Laws. In: Rodin Project Deliverable D7: Event-B Language. http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf
2. Butler, M.: Decomposition Strategies for Event-B. In: Leuschel, Wehrheim (ed.) Proc. IFM-09. vol. 5423, pp. 20–38. Springer, LNCS (2009)
3. Hallerstede, S., Abrial, J.R.: Event-B Decomposition for Parallel Programs. In: Frappier and Gl asser and Khurshid and Laleau and Reeves (ed.) Proc. ABZ-10. vol. 5977, pp. 319–333. Springer, LNCS (2010)
4. Hallerstede, S., Hoang, T.: Refinement by Interface Instantiation. In: Derrick, Fitzgerald, Gnesi, Khurshid, Leuschel, Reeves, Riccobene (ed.) Proc. ABZ-12. vol. 7316, pp. 223–237. Springer, LNCS (2012)
5. Silva, R., Pascal, C., Hoang, T., Butler, M.: Decomposition Tool for Event-B. Software Practice and Experience 41, 199–208 (2011)