Problem Decomposition and Sub-Model Reconciliation of Control Systems in Event-B

Sanaz Yeganefard
Electronics and Computer Science
University of Southampton, UK
sy2g08@ecs.soton.ac.uk

Michael Butler
Electronics and Computer Science
University of Southampton, UK
mjb@ecs.soton.ac.uk

Abstract

To break the complexity of the formalisation process, we propose to model a functional requirement document of a control system as composeable monitored, controlled, mode and commanded sub-models. Influenced by the problem frame approach and the decomposition of the four-variable model, we suggest decomposing requirements of a control system into monitored, controlled, mode and commanded sub-problems. Each sub-problem can be formalised in a step-wise manner as a separate sub-model.

To introduce the phenomena shared amongst the sub-problems, the sub-models are reconciled. We propose a reconciliation process in the Event-B formal language based on the shared-variable and the shared-event styles which were originally developed for a model decomposition. The advantages and disadvantages of shared-variable and the shared-event reconciliation steps are also discussed. The requirements of an automotive cruise control system are decomposed and formalised as sub-models. These sub-models are also reconciled to introduce shared phenomena.

1. Introduction

Control systems are complex, as they usually consist of a number of hardware and software components which interact with the evolving environment and can receive commands from operators. Such systems are usually used in life critical situations, and as they control environmental quantities, such as temperature, pressure or speed, their failures can be costly and potentially life threatening.

To improve the functionality of control systems, their software applications have become more complex. For instance, software has allowed the automotive industry to produce intelligent cars. This means software errors can play an essential role in the system failure. As is well-known, the cost of identifying and correcting a software error is usually higher in late stages of a development life cycle compared with early stages. Also, one cause of software errors is 'faulty requirements definition'. Thus, approaches such as formal methods, which are mathematical based techniques, can be used to reveal ambiguity, incompleteness and inconsistency of requirements and verify system properties in early development stages.

However, the transition from informal requirements to a formal model can be time consuming and complicated, since the requirements of a control system are usually large in number and complex. One approach for breaking system complexity is *decomposition*. To help with understanding a complex problem, decomposition of a problem into smaller *sub-problems* each with a degree of unity has been suggested [8]. One contribution of this paper is to propose an approach for breaking the complexity of the formalisation process of a functional requirement document (RD) of a control system.

The first step of the approach is to decompose an RD of a control system into *monitored*, *controlled*, *mode* and *commanded sub-problems*. The sub-problems are defined based on the requirement structuring approach for control systems [11]. The second step is to formalise each sub-problem as a *separate sub-model* which is *composeable* with other sub-models. So, while each sub-model captures a set of requirements, the composition of the sub-models represents the overall specification. Furthermore, to simplify the formalisation process *refinement* techniques are used within each sub-model. As it is often not clear how to construct a refinement chain, in this paper we suggest some guidelines on refinement steps. Our chosen formal language is Event-B [3], although this approach is applicable in other state-based formal languages with composition mechanisms.

Another contribution of this paper is to propose *reconciliation* of the phenomena shared amongst the sub-models formalised in the Event-B language. Shared phenomena are reconciled as *shared variables* and *shared events* based on the *shared-variable* [4] and the *shared-event* [6] decomposition styles in Event-B. During reconciliation the cross-cutting invariants and the consistency of sub-models are proven. Note that the shared-event and the shared-variable styles were originally developed for model decomposition. In this paper, they are adapted and developed further for the purpose of the shared phenomena reconciliation. Also, an evaluation of these two reconciliation steps is provided. The proposed approach is applied to a case study of a cruise control system (CCS).

2. Decomposing RD into Sub-Problems

To structure an RD of a control system which consists of plant, controllers and operators, the identification of *monitored*, *controlled*, *mode* and *commanded* variable phenomena are suggested [11]. Influenced by the four-variable model [9], the *monitored* and *controlled* phenomena are environment quantities whose values are determined by the plant and the controller respectively. The *mode* phenomenon represents the controller modes. *Commanded phenomena* are quantities whose values are determined by the operator and that influence controlled and mode phenomena. The RD of a control system can be structured according to these phenomena by following the steps below [11].

1. List the system's monitored, controlled and commanded phenomena.

2. List the values of the mode phenomenon. One simple way is to use a state machine diagram.

3. Organise RD into four sections: monitored (MNR), controlled (CNT), mode (MOD) and commanded (CMN). Requirements are placed in the section which represents the type of their *main phenomenon*. This is the phenomenon that the requirement describes how or when it will be modified.

4. Add unique ID labels. Every ID starts with the section the requirement belongs to (i.e. MNR, CMN, CNT or MOD), followed by a unique number.

The above requirement structuring steps will result in the RD being divided into monitored, controlled, mode and commanded sections where each section describes the modification of its corresponding phenomena. Thus, each section can be treated as a sub-problems which can be formalised independently. The independent sub-models are then reconciled to accommodate shared phenomena.

3. Formalising Sub-Problems as Sub-Models

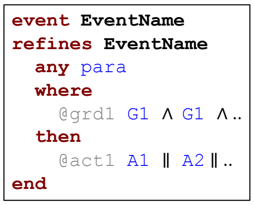
An overview of Event-B is given in Section 3.1. Section 3.2 briefly explains the formalisation of sub-problems.

3.1. Event-B Formal Language

Event-B is extended from B Method [1] and has evolved from action systems [5]. In particular, the notation of guarded actions and some refinement notations are based on action systems. An example is that new variables and events can be defined in the refinement levels by refining a dummy event called *skip* which does nothing in the more abstract level. This refinement style allows behaviour to be added to a model gradually thus elaborating the model with requirements that were not included in abstract levels.

The Event-B language is chosen because of the simplicity of its notation, its support for incremental refinement-based development and the availability of a tool. This tool, called Rodin, provides automatic proof and a wide range of plug-ins, such as shared-event composition [10] which is used in this work.

An Event-B model consists of `contexts` and `machines`. Contexts specify the static part of a model and provide axiomatic properties. In this paper we focus on the dynamic part which is captured in `machines`. Machines contain `variables`, `invariants` and `events`. An event, Figure 1, can refine a more abstract event. It is possible to replace the `refines` keyword with `extends` which means the guards and actions of the abstract event remain unchanged, while further guards or actions can be introduced in the refinement. Proof obligations are generated to ensure that a refined (extended) event is correct with respect to its abstraction. An event can consist of `parameters`, `guards` and `actions`. Guards are predicates which describe the conditions necessary for the occurrence of an event. Actions determine how specific state variables change as a result of the event occurrence. Actions of an event take place simultaneously to preserve its atomic nature.



```
event EventName
refines EventName
  any para
  where
    @grd1 G1 ∧ G1 ∧..
  then
    @act1 A1 ∥ A2 ∥..
end
```

Figure 1. An event in Event-B.

3.2. Formalisation of Sub-Problems

To decompose the complexity of the modelling process, we formalise each monitored (MON), controlled (CNT), mode (MOD) and commanded (CMN) sub-problem as an independent sub-model which is reconciled to deal with phenomena it shares with other sub-models. To provide the overall system specification, the most concrete levels of all the sub-models can be composed.

To formalise the sub-problems, we propose to identify the *monitor*, *control*, *mode* and *command* event phenomena which update or modify variable phenomena, e.g. a mode event such as *switch on* updates the *mode* variable. The variable and event phenomena are then represented formally in their corresponding sub-model, e.g. the *switch on* event and the *mode* variable are formalised in the MOD sub-model.

We also suggest using refinement to formalise the requirements of a sub-problem gradually. In the most abstract level we start with the phenomenon that represents *the main behaviour of the sub-problem*. It is the modeller's judgment to identify the main behaviour. Human judgment is necessary in the development of a formal model, for instance in constructing the refinement chain and validating the model against requirements. Human judgment can be contrasted with a mathematical judgment which provides the means for proofs and verification.

After modelling the main behaviour, the phenomena remaining in a sub-problem can be formalised in refinement steps. The main rule we follow is to model as few phenomena as possible. We suggest formalising a small number of phenomenon at every refinement level. Sometimes phenomena are interrelated which means they need to be modelled simultaneously in one level.

4. Reconciliation of Shared Phenomena

Sub-models can share phenomena because requirements of a sub-problem may refer to phenomena in other sub-problems. To ensure shared phenomena are modelled correctly and cross-cutting invariants are preserved, the sub-models are reconciled. In Section 4.1, the shared-variable decomposition is adapted for reconciliation of *variable phenomena* shared amongst sub-models. In Section 4.2, the shared-event composition style is used for reconciliation of *shared event phenomena*. Furthermore, in Section 4.3 we developed an approach based on the shared-event composition that allows the reconciliation of *shared variable* as well as *shared event phenomena*.

4.1. Shared-Variable Reconciliation

The shared-variable reconciliation is based on the model decomposition style of [4]. Here, this decomposition is adapted to a composition style based on which sub-model reconciliation is developed. The reconciliation is explained using the composition example of Figure 2. In this figure M is a composition of the machines M1 and M2. M1 consists of the events E1 and E2 whose dependencies to the variables *v1* and *v2* are shown using the links. A dependency link shows that an event can modify or access a variable. This in Event-B means the variable may appear in actions or guards of the event. Similarly, E3 and E4 in M2 depend on *v2* and *v3*. So, while *v1* and *v3* are *local* (internal) to M1 and M2 respectively, *v2* is *shared* among them.

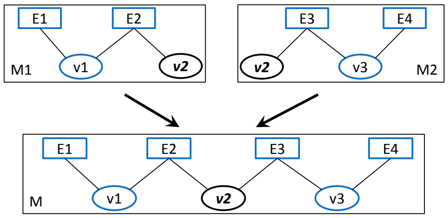


Figure 2. Shared-variable composition style.

We treat M1 and M2 as composeable sub-models that share the variable phenomenon *v2*. To ensure that the behaviours on *v2* in M1 and M2 are consistent and also the cross-cutting invariants are preserved the sub-models are reconciled. To do this, *external events* [4] which simulate the behaviour on the *shared variable phenomena* are added to the sub-models. Introducing external events gives rise to proof obligations (POs), such as the invariant preservation PO, which ensure the correctness of sub-models. Thus, after adding external events the sub-models which were independent of each other, may require modifications to preserve the invariants.

Figure 3 shows the external events *E2'* and *E3'*. *E3'* (*E2'*) of M1 (M2) simulates the way in which its environment, i.e. M2 (M1), may modify the shared variable *v2*.

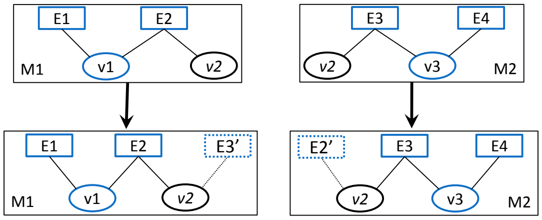


Figure 3. Shared-variable style reconciliation.

Assume *E3* has the following form:

$$E3 \triangleq \; Any \;\; p \;\; Where \; G(v2,p) \wedge H(v3,p)$$
$$Then \; S(v2,p) \parallel T(v3,p)$$

E3' can be constructed from *E3* by projecting out the non-shared variable *v3* resulting in the following:

$E3' \triangleq \; Any \;\; p \;\; Where \; G(v2,p) \;\; Then \; S(v2,p)$

Notice that external events cannot be refined, as refining an external event can break its consistency with other sub-models.

The nature of this style allows the reconciliation of *shared variable phenomena*. As will be discussed later, shared event phenomena cannot be reconciled in this style.

4.2. Shared-Event Reconciliation

The shared-event style "corresponds to the synchronous parallel composition of processes as found in process algebra such as CSP" [6]. The nature of this style allows us to reconcile *shared event phenomena*. This means events of different sub-models can contribute to a specification, since they can be composed. As shown in Figure 4, M1 and M2 share the event *E2*. The event *E2a* (*E2b*) in M1 (M2) represents parts of *E2* which refer to the variable *v1* (*v2*). The variable *v1* and the event *E1* are *local* (internal) to M1. Similarly, *v2*, *v3* and *E3* are local to M2. So, M1 and M2 can be treated as sub-models which share an event.

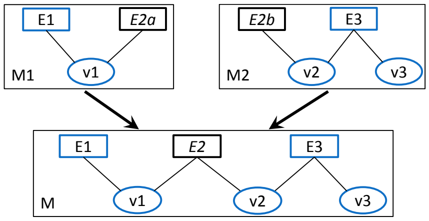


Figure 4. Shared-event composition style.

As shown in (1) below, two events are composed by conjoining their guards and combining their actions so they happen simultaneously. It is possible for guards and actions with disjoint variables to have common parameters. In Definition 1, the events *E2a* and *E2b* share the parameter *p*.

$$
\begin{aligned}
E2a &\triangleq \; Any \;\; p \;\; Where \; G(v1, p) \;\; Then \; S(v1, p)\\
E2b &\triangleq \; Any \;\; p \;\; Where \; H(v2, p) \;\; Then \; T(v2, p)\\
E2 &\triangleq \; Any \;\; p \;\; Where \; G(v1, p) \wedge H(v2, p)\\
&\qquad Then \; S(v1, p) \parallel T(v2, p)
\end{aligned}
\tag{1}
$$

4.3. Introducing Shared Variables in Shared-Event Reconciliation

It is possible for sub-models to share variable phenomena. As an example consider a requirement such as "*v1* shall be increased when *v3* is greater than 0" for Figure 4 where *v1* and *v3* reside in two different sub-problems (M1 and M2 respectively). This requirement belongs to the M1 sub-problem, since it defines a condition under which *v1* can be modified. This means *v3* is a phenomena shared amongst M1 and M2. Note that our aim is to model every requirement in a single sub-model. In other words, we avoid modelling the part referring to *v1* in M1 and the rest in M2.

To model such requirements using shared-event reconciliation, variable sharing in addition to event sharing is required. This is shown in Figure 5. Here, the event *E3'* is defined in M1 to simulate the behaviour on *v3* in M2. To some extent this event is similar to an *external event* in the shared-variable style. However here, events on shared phenomena synchronise, which means they are *shared* amongst the sub-models. For instance, *E3* and *E3'* are shared events.

However, shared variables can be modified only in one sub-model. In Figure 5, *v3* can be modified by events in M2, whereas E1 can only read the value of *v3*. In other words, a single sub-model has read and write access to a shared variable, while other sub-models have a read-only access. This is shown by the dependency link between *E1* and *v3* which is named *rd*. So, in this style, sub-models can have *shared event phenomena* as well as *read-only shared variable phenomena*.

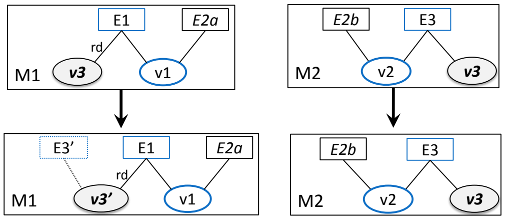


Figure 5. Shared-event style reconciliation.

Figure 5 also shows that a copy of *v3*, called *v3'*, is defined in M1. This is because the current semantics of the shared-event style does not allow the introduction of variables with the same name in sub-models. This semantic approach emphasises that variables are *local* to a sub-model. So, a variable which is shared amongst several sub-models is named differently in every sub-model. To prove the correctness of a sub-model with regards to all possible values for a shared phenomenon and to prove the consistency between the sub-models, invariants which equalise shared variables are defined in the composed model, e.g. the invariant *v3* = *v3'*. Such invariants give rise to consistency and correctness related POs. Note that to discharge this invariant all possible modifications of *v3* in M2 should be simulated in M1.

5. Case Study: Cruise Control System

A CCS receives the *actual speed* of the car from the environment and the *target speed* from the driver and its role is to minimise the difference between these two. This is done by setting the *acceleration* of the car [2]. Some of the requirements of the CCS are structured in Table 1. The CCS can be modelled as monitored (MNR), controlled (CNT), mode (MOD) and commanded (CMN) sub-models which share some phenomena.

The requirement CNT1 (CCS operates when it is `active`) denotes that the CNT and MOD sub-problems share the *mode* variable phenomenon. This is because the *mode* phenomenon belongs to the MOD sub-problem, however, CNT1 refers to *mode*. Also, the requirement CMN1 (once CCS is `on`, the driver can set the target speed) shows that *mode* is shared amongst the CMN and MOD sub-problems. Thus, the sub-models of the CCS need to be reconciled to deal with the shared variable *mode*. Similarly, the four sub-models are reconciled to model other shared phenomena, such as *brake pedal* which is shared between MNR and MOD sub-problems. However, in this paper only reconciliation of the *mode* phenomenon is explained.

Phen.	**ID**	**Requirement Description**
Actual speed	MNR1	CCS shall monitor the vehicle's actual speed.
Brake pedal	MNR2	CCS shall monitor any pressure applied to the brake pedal.
Acceleration	CNT1	When CCS is `active`, it will maintain the difference between actual speed and target speed as close to 0 as possible by correcting the acceleration.
Mode	MOD1	CCS can be switched `on` or `off` by the driver.
	MOD2	When CCS is `on`, it will be `activated` as soon as the driver sets the target speed.
	MOD3	Once CCS is `active`, if the driver uses the brake pedal, it will be `suspended`.
Target speed	CMN1	Once CCS is switched `on`, the driver can set the target speed to the actual speed.
	CMN2	The target speed is always within a specific range.

Table 1. Structured RD of a CCS.

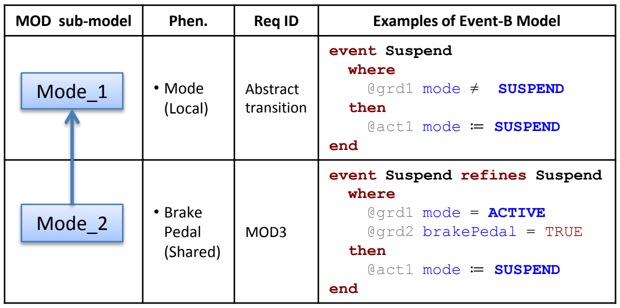
6. Formalising CCS as Sub-Models

In Section 6.1, the formalisation of the CNT, MOD and CMN sub-problems as sub-models are explained. The MNR sub-model is not represented in this paper, although the approach for modelling MNR is similar to others. The sub-models are reconciled in Section 6.2 and 6.3 based on the shared-variable and the shared-event styles respectively.

6.1. CNT, MOD and CMN Sub-Models in CCS

The CNT, MOD and CMN sub-models of the CCS which formalise their corresponding sub-problems are discussed in this section. As mentioned these sub-problems share the *mode* variable phenomenon.

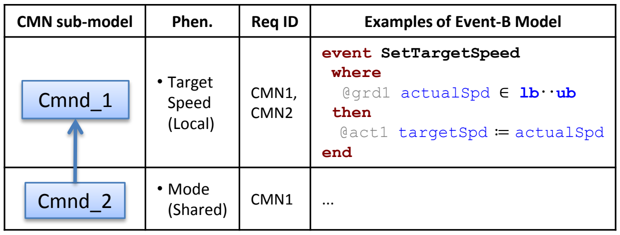
Figure 6 shows parts of the refinement in the MOD sub-model. It also shows the requirement ID numbers which are modelled at each step. In the MOD sub-model, the transitions between values of the *mode* variable are modelled in the most abstract level, *Mode_1*. For instance, the *Suspend* event at this level simply shows that the CCS can be suspended from any other state. After this, the conditions under which the mode change happens are added in the refinement steps. For instance, the CCS should be suspended when brake pedal is pressed. This is shown in the guard $brakePedal = TRUE$ in *Mode_2*. Note that the reconciliation of introducing the monitored variable *brake pedal* in the MOD sub-model is not discussed in this paper. However, it is similar to reconciling the shared *mode* variable.



MOD sub-model	Phen.	Req ID	Examples of Event-B Model
Mode_1	• Mode (Local)	Abstract transition	event Suspend where @grd1 mode ≠ SUSPEND then @act1 mode ≔ SUSPEND end
Mode_2	• Brake Pedal (Shared)	MOD3	event Suspend refines Suspend where @grd1 mode = ACTIVE @grd2 brakePedal = TRUE then @act1 mode ≔ SUSPEND end

Figure 6. MOD sub-model in CCS.

Figure 7 represents the CMN sub-model where the commanded variable *target speed* and its corresponding events are formalised. As shown in *Cmnd_1*, to model the requirement CMN1 the formal variable *targetSpd* is set to *actualSpd*. The introduction of the shared variable *mode* in *Cmnd_2* is discussed later on.



CMN sub-model	Phen.	Req ID	Examples of Event-B Model
Cmnd_1	• Target Speed (Local)	CMN1, CMN2	event SetTargetSpeed where @grd1 actualSpd ∈ lb‥ub then @act1 targetSpd ≔ actualSpd end
Cmnd_2	• Mode (Shared)	CMN1	...

Figure 7. CMN sub-model in CCS.

The *acceleration* phenomenon is modelled in the CNT sub-model which is shown in Figure 8. As *Cntrl_1* shows a function, called *accFun*, determines the value of *acceleration* based on the *actual speed* and the *target speed* phenomena. The *mode* phenomenon, which is shared amongst MOD and CNT sub-models is modelled in *Cntrl_2* (Figure 8) and its reconciliation is explained later on.

CNT sub-model	Phen.	Req ID	Examples of Event-B Model

Cntrl_1	• Acceleration (Local)	CNT1	`event UpdateAcceleration`
			` then`
			`  @act1 acceleration ≔`
			`    accFun(targetSpd↦actualSpd)`
			`end`

Cntrl_2	• Mode (Shared)	CNT1	…

Figure 8. CNT sub-model in CCS.

6.2. Reconciling CCS using Shared-Variable Style

As mentioned in the shared-variable reconciliation *external events* are defined to simulate the behaviour on a shared variable. So, to reconcile the variable *mode*, external events which capture the behaviour on *mode* in the MOD sub-model are added to the CNT and CMN sub-models. Figure 9 illustrates the state machine which represents the transitions updating *mode* in the MOD sub-model. An abstraction of these events is defined as an external event in CNT and CMN, i.e., in refinement levels $Cntrl\_2$ and $Cmnd\_2$.

Mode Events in MOD Sub-Model	Abstraction of Mode Events in CNT and CMN Sub-Models

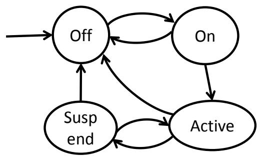
	`event UpdateMode`
	`  any st`
	`  where`
	`    @grd1 st ∈ STATUS`
	`  then`
	`    @act1 mode ≔ st`
	`end`

Figure 9. External mode event.

When the shared variable *mode* and its external events are introduced in the CNT and CMN sub-models, the requirements CNT1 and CMN1 can be modelled by refining the events *UpdateAcceleration* and *SetTargetSpeed* in $Cntrl\_2$ and $Cmnd\_2$ respectively. The refinement of these events are shown in Figure 10.

CNT Sub-Model	CMN Sub-Model

`event UpdateAcceleration`
`extends UpdateAcceleration`
`   when`
`      @grd1 mode = ACTIVE`
` end`
	`event SetTargetSpeed`
	`extends SetTargetSpeed`
	`   where`
	`      @grd2 mode = ON`
	`end`

Figure 10. CNT and CMN sub-models after shared-variable reconciliation.

6.3. Reconciling CCS using Shared-Event Style

As discussed in Section 4.3, we developed the shared-event reconciliation in a manner that sub-models are able to share *variable* (read-only access) or *event* phenomena.

So, to model the requirement CNT1 (CCS operates when it is `active`) in $Cntrl\_2$, the shared variable *mode* is added to the CNT sub-model. To do this a copy of *mode*, which is called *cnt_mode* is introduced in the CNT sub-model. The consistency of these two variables and any cross-cutting invariant can be proved by adding the invariant $mode = cnt\_mode$ to the composed model.

An abstract event of the mode transitions is added to the CNT sub-model in order to update *cnt_mode*. This event is represented in Figure 11 as *UpdateMode*. The *UpdateMode* event synchronises with the mode events in MOD. For instance, Figure 11 illustrates that the events *Suspend* in MOD and *UpdateMode* in CNT synchronise through the parameter *st*. The requirement CNT1 is modelled by refining the event *UpdateAcceleration*.

MOD Sub-Model	CNT Sub-Model

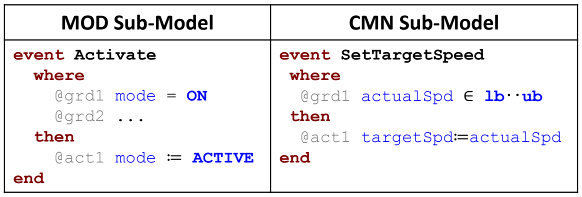
`event Suspend`
`refines Suspend`
`  any st`
`  where`
`    @grd1 mode = ACTIVE`
`    @grd2 brakePedal = TRUE`
`    @grd3 st  = SUSPEND`
`  then`
`    @act1 mode ≔ SUSPEND`
`end`
	`event UpdateMode`
	`  any st`
	`  where`
	`    @grd1 st ∈ STATUS`
	`  then`
	`    @act1 cnt_mode ≔ st`
	`end`
	
	`event UpdateAcceleration`
	`extends UpdateAcceleration`
	`  when`
	`   @grd1 cnt_mode = ACTIVE`
	`end`

Figure 11. Shared-event reconciliation for the CNT sub-model.

In this reconciliation style, sub-models can share event phenomena. Therefore, the requirement CMN1 (Once the CCS is switched `on`, the driver can set the target speed to the actual speed) and MOD2 (When CCS is on, it will be `activated` as soon as the driver sets the target speed) can be modelled as independent events which belong to the CMN and MOD sub-models respectively, but are also shared amongst these sub-models. Figure 12 shows that the former requirement is modelled by the *SetTargetSpeed* event, while the latter is represented by the *Activate* event. The composition of these two events will provide an event which updates the variables *target speed* and *mode* simultaneously. Thus, the requirements CMN1 and MOD2 are modelled as a single atomic event.

7. Evaluation and Discussion on Reconciliation

Section 7.1 and 7.2 evaluate the shared-variable and the shared-event reconciliation styles respectively.



MOD Sub-Model

```
event Activate
  where
    @grd1 mode = ON
    @grd2 ...
  then
    @act1 mode ≔ ACTIVE
end
```

CMN Sub-Model

```
event SetTargetSpeed
 where
  @grd1 actualSpd ∈ lb··ub
 then
  @act1 targetSpd≔actualSpd
end
```

Figure 12. *Activate* **and** *SetTargetSpeed* **are reconciled as shared events.**

7.1. Reconciliation based on Shared-Variable Style

The main disadvantage of the shared-variable reconciliation is that sub-models cannot share events. In other words, events of different sub-models cannot contribute to the specification of requirements. As an example, consider a commanded requirement "x should be set to 0 as soon as the *mode* becomes `off`", and a mode requirement which states "*mode* becomes `off`" under certain conditions.

To formalise these two requirements in a single model, two design decisions can be made. Firstly, these requirements can be modelled *concurrently* using an event with two actions which update x and *mode* simultaneously. Secondly, these requirements can be formalised *sequentially* by modelling each one as a separate event; a mode event which has the action $mode := off$ and a command event which updates x when the guard $mode = off$ holds.

However, when modelling the considered requirements in sub-models (e.g. one in CMN and another in MOD), the shared-variable reconciliation will not allow event synchronisation. Therefore, the only way to model these requirements is the sequential design decision. This was the case in the example of the CCS, Section 6.2, where an external mode event was defined in the CMN sub-model and the guard $mode = ON$ was introduced to the event *SetTargetSpeed*.

Therefore, in the shared-variable reconciliation, requirements of different sub-problems cannot be modelled concurrently, because this style does not allow synchronisation of events, i.e., *shared event phenomena* cannot be defined. This imposes an early design decision in the modelling process. To avoid this restriction sub-models should be allowed to contribute to the specification of a single event. This is similar to the shared-event style.

7.2. Reconciliation based on Shared-Event Style

The main advantage of the shared-event reconciliation we proposed in Section 4.3 is that sub-models can share *variable* and *event phenomena*. Sharing events means several sub-models can contribute to the specification of an event. Therefore, this style allows freedom in design decision. However, the formalisation of shared variables in this reconciliation is more restrictive than the shared-variable reconciliation, since here shared variables can be modified only in one sub-model, while other sub-models have read-only access to the shared variables.

One disadvantage of the shared-event reconciliation is the extra events which should be defined to update shared variable phenomena. As Figure 11 showed, the event *UpdateMode* was defined in CNT to simulate the behaviour on *mode* in the MOD sub-model. Defining such simulating events is also a disadvantage of the shared-variable reconciliation which requires *external events*. However, to prove the correctness of cross-cutting invariants and consistencies of shared variables, the shared-event reconciliation requires composing the sub-models to express synchronisations between simulating events and their concrete simulated events. Whereas in the shared-variable reconciliation, ensuring that external events are simulated correctly is sufficient for proving cross-cutting invariants.

Another disadvantage of the shared-event reconciliation is that because of restrictions on its semantics, shared variable phenomena should be renamed in different sub-models. In addition, invariants which equalise the shared variables (such as $mode = cnt\_mode$ in the CCS example) should be defined to prove consistencies between sub-models. Renaming variables, defining extra events and composing the sub-models to prove consistencies can impose a great overhead to the modelling process.

8. Related Work

The proposed monitored, controlled, mode and commanded phenomena are influenced by the four-variable model [9]. In particular, the concept of monitored and controlled phenomena are taken from the four-variable model. The other two categories of the four-variable model are *input* and *output*. We suggested that these are not used in the system modelling; instead we introduce them in later refinement steps where design details, such as sensors and actuators are modelled [12].

A related work is problem frames (PF) [8] which is a semi-formal approach used in requirement engineering of a range of systems. PF introduces patterns (frames) according to which sub-problems can be identified. Also, it tackles the issue of sub-problem consistencies (requirement conflicts) in a semi-formal manner using Composition Frames. However, this paper proposed an approach for categorising requirements into sub-problems and then formalising them. Also, we overcame the issue of consistencies between the formalised sub-problems (sub-models) using formal proofs that result from the reconciliation step.

The work of [7] proposes an approach for decomposing requirements into sub-problems based on the frames and patterns of PF. Similarly to our work, in [7] sub-problems are formalised individually and connected to one another. While the approach of [7] requires a good knowledge of problem frames, we based the decomposition of requirements on simple concepts which involve understanding requirements. Also, we focused on proposing and evaluating approaches for formalising sub-problems and reconciling sub-models in Event-B.

Another related work, is the approach of [13] for composing partial specifications which are written in different formal languages by defining a general semantic domain. Similarly to this approach we proposed a systematic way of composing sub-models. However, we focus on the Event-B formal language and the composition of sub-models in this language. Also, the approach of [13] does not consider decomposition of an RD into sub-problems and their mappings to a formal model.

9. Conclusion and Future Work

In this paper we discussed an approach for decomposing a control system RD into *monitored*, *controlled*, *mode* and *commanded sub-problems* which can be formalised as four composeable sub-models. We believe this approach is applicable in other state-based formal languages with composition mechanisms, although in this paper we considered the Event-B language only.

Dividing an RD into sub-problems means the required effort and the complexity of modelling each sub-problem will be less than the overall RD. In addition, in the proposed approach sub-models are modularised, since shared phenomena can be modified (read-write access) in one sub-model, while other sub-models can only read the shared phenomena (read-only access). Thus, it is sufficient to model the concrete behaviour of phenomena in a sub-model, while others can have an abstract representation of shared phenomena.

This is a consequence of decomposing an RD into sub-problems according to the structuring steps of Section 2, as requirements with write access to a phenomenon always reside in a single sub-problem.

In addition, we adapted the *shared-variable* model decomposition style to provide a reconciliation step for modelling variables shared amongst sub-problems. We showed that this reconciliation style is restrictive, as it imposes an early design decision. We also developed a *shared-event* reconciliation step based on the shared-event model decomposition style. This reconciliation step can be used to model shared variables and shared events. However, the main disadvantage of the shared-event reconciliation is the overhead of defining extra events and variables with different names.

In our future work the approach of formalising control systems as monitored, controlled, mode and commanded sub-models will be applied to other formal languages. Also, the semantics of the shared-event reconciliation in Event-B will be extended, so that it will include the advantage of event synchronisation, but eliminate the overheads. In addition, in our future work we will refine each sub-model independently to introduce design details such as sensors, actuators and the user interface. Data refinement of shared variables should also be considered, as currently this cannot be done easily in either of the styles.

References

[1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. Cruise control requirement document. Technical report, Internal report of the Deploy project, 2009.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.

[5] R. J. R. Back. Refinement calculus, part II: parallel and reactive programs. In *Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, REX workshop, pages 67–93. Springer-Verlag New York, Inc., 1990.

[6] M. Butler. Decomposition structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[7] D. Jackson and M. Jackson. Problem decomposition for reuse. *Software Engineering Journal*, 11(1):19–30, 1996.

[8] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co.,, Boston, USA, 2001.

[9] D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, 1995.

[10] R. Silva and M. Butler. Shared event composition plug-in. http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B, 2011. cited 2012 April.

[11] S. Yeganefard and M. Butler. Control systems: phenomena and structuring functional requirement documents. In *17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2012).*, April 2012.

[12] S. Yeganefard, M. Butler, and A. Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 182–191. NASA, April 2010.

[13] P. Zave and M. Jackson. Conjunction as composition. *ACM Trans. Softw. Eng. Methodol.*, 2(4):379–411, Oct. 1993.