Feature Refinement

Steve Reeves and David Streader
Department of Computer Science, University of Waikato, Hamilton, New Zealand
{stever,dstr}@cs.waikato.ac.nz

Abstract

Development by formal stepwise refinement offers a guarantee that an implementation satisfies a specification. But refinement is frequently defined in such a restrictive way as to disallow some useful development steps. Here we define feature refinement to overcome some limitations of refinement and show its usefulness by applying it to examples taken from the literature.

Using partial relations as a canonical state-based semantics and labelled transition systems as a canonical event-based semantics, we define functions formally linking the state- and event-based operational semantics. We can then use this link to move notions of refinement between the event- and state-based worlds.

An advantage of this abstract approach is that it is not restricted to a specific syntax or even a specific interpretation of the operational semantics

Keywords: state-based refinement, event-based refinement, feature refinement, stuttering refinement, Z, B, Event B

1 Introduction

In this paper we introduce *feature refinement* (Definition 11). We apply it to examples used in the literature, including one where the point is to show that retrenchment is more flexible than normal refinement.

In general a refinement step is the development of a concrete description C from an abstract description A, written $\mathsf{A} \sqsubseteq \mathsf{C}$. The idea, as commonly described in the literature, is that $\mathsf{A} \sqsubseteq \mathsf{C}$ iff:

> any user of A will not be able to observe if they had actually been given C in its place

There are *many* distinct definitions of refinement, including those that we consider here, that can can be said to formalise this. To illustrate this we consider a simple example illustrating the difference between normal refinement [12, 9, 1, 23] and feature refinement, both of which can be viewed as particular formalisations of this common description.

1.1 An example needing feature refinement?

Can we refine a one place buffer $\mathsf{B_A}$ with two operations in and out, into $\mathsf{B_C}$, a one place buffer with the additional feature of a third operation del that terminates the buffer (this example is considered in more detail in [18])? This is not allowed as a refinement by many definitions in the literature but it can be argued that it should be, as any successful use of $\mathsf{B_A}$ will also succeed with $\mathsf{B_C}$.

Clearly if a user–program were to call the del operation then they would be able to tell if they were given $\mathsf{B_A}$ or $\mathsf{B_C}$ and hence it would not be a refinement. But by restricting the set of programs to those that call only the operations of $\mathsf{B_A}$, then none can tell if they were given $\mathsf{B_C}$ and $\mathsf{B_A} \sqsubseteq \mathsf{B_C}$ would be a valid refinement.

Using the normal state-based definitions of refinement found in the literature $\mathsf{B_A}$ cannot be refined into $\mathsf{B_C}$, an exception being [2] where, as we shall see, it depends crucially upon the details of the definitions.

A reader might, quite reasonably, take the view that the step from $\mathsf{B_A}$ to $\mathsf{B_C}$ is not a refinement step but some other development step for example a "versioning" step. From this view point the work presented here can be seen as giving a formal definition of the version relation and a formal guarantee of what behaviour is preserved between different versions.

In fact, in order to satisfy not only people who, like us for the reasons above, view this buffer development as a valid refinement, but also people who want it to be *disallowed*, we allow specifications of either view.

1.2 The bigger picture

Development by formal stepwise refinement offers a guarantee that an implementation satisfies a specification. But refinement is frequently defined in such a restrictive way as to not allow some useful development steps. Retrenchment [6] offers a more flexible development but the

Unrecognized Copyright Information
DOI 10.1109/SEFM.2007.44

IEEE computer society

guarantee that an implementation satisfies a specification is lost.

In order to combine the advantages of event-based and state-based approaches we are going to build semantic mappings (both ways) between partial relations, a canonical state-based operational semantics, and labelled transition system (LTS), a canonical event-based operational semantics. That this is possible is well-known, of course, but should be kept in mind in the sequel as it motivates much that we do. Its details are straightforward, as we will see, and it is used formally in the proofs of most of the lemmas in this paper.

Event-based refinements in [11] permit the addition of new operations and are given an abstract formalisation in [18] thus, using the mappings mentioned above, allowing the results to be applied to a variety of event-based *and* state-based formalisms. Here we will elaborate the work in [18] using a more state-based language and add a parameterised definition of simulation. This has enabled us to show that a recent definition of Event B refinement [2] is less general than an old event-based refinement in [11].

It is important to recall that event-based operational semantics, commonly defined as LTS , have been given many different interpretations, e.g. abstract data types with singleton failure semantics [7], handshake processes with failure and trace semantics [13] and with broadcast semantics [17], etc. Similarly state-based operational semantics, commonly defined as partial relations, have been given many different interpretations. In Z and B [23, 21, 1] partial relations are interpreted as undefined outside of precondition and totally correct, while in [7] they are interpreted as guarded outside of precondition and totally correct, but in [9, Chapter 1-7] they are interpreted as partially correct. It is common with both state- and event-based operational semantics to give different interpretations by using different definitions of refinement [7, 13, 17, 23, 21, 1, 9] to, so to speak, 'complete' the semantics.

Our bridge between the state- and event-based approaches can be used prior to giving the operational semantics any specific interpretation, that is prior to completing the semantics.

This contrasts with much of the work combining state- and event-based approaches [7, 11, 10] where at the very start specific interpretations of the state- and event-based semantics are chosen.

1.3 Plan of the paper

The operational semantics of B, Event B, Z and so on are concrete versions of the general, more abstract, semantics we develop in Section 2.

It can be argued that one of the most important contributions of the B methodology is the definition of first-order logical conditions sufficient to imply forward simulation and hence refinement [12, 9]. Thus satisfying B's proof obligations implies refinement as found in B. We wish to keep to this admirable and practical philosophy, hence, after defining feature refinement ($\sqsubseteq^D_{FR}$) this paper: one, defines a forward simulation that implies feature refinement; two, establishes in what way C satisfies A when A is a feature refinement of C; and three, illustrates the usefulness of feature refinement in modelling design steps that previously have only been formalised as retrenchments.

In Section 3 we define the event-based labelled transition system semantics and in Section 3.1 we relate it to the previously defined state-based semantics. In Section 4 we give a state-based definition of stuttering steps and using this we define the parameterised feature refinement in Section 4.1 and parameterised simulation in Section 4.2.

In Section 5 we give the event-based definition of hiding and restriction and using the simple relation (Section 3.1) between the state- and event-based semantics show that hiding is related to stuttering and feature refinement includes both hiding and restriction.

In Section 6 we specialise our abstract definition of refinement and apply it to an example formalised in B, using both undefined and guarded (magic) parts of an operation's domain.

2 State-based relational semantics

This section considers operations on state that are given a partial relation semantics. Special cases of this include, but are not limited to, how Z and B work. It should be clear that the machines of B relate directly to the relational semantics defined below, as do collections of Z operation schemas and the state over which they are defined.

A relational semantics is based around a set of named partial relations.

In both this section and the next we have Act, the universal set of operation/event names.

Definition 1 *Relational semantics. Let* Σ_{M} *be a finite set of states, the state space of* M. M *is a relational semantics where:*

$$\mathsf{M} \triangleq (\Sigma_{\mathsf{M}}, \mathsf{init}_{\mathsf{M}}, Npr_{\mathsf{M}}, Alp_{\mathsf{M}}),$$

where, given a single global state $\bullet \notin \Sigma_{\mathsf{M}}$,

$$\mathsf{init}_{\mathsf{M}} \subseteq \bullet \times \Sigma_{\mathsf{M}}$$

is an initialising operation,

$$Npr_{\mathsf{M}} \subseteq \{(\mathsf{o}, R_{\mathsf{o}}) | \mathsf{o} \in Alp_{\mathsf{M}} \wedge R_{\mathsf{o}} \subseteq \Sigma_{\mathsf{M}} \times \Sigma_{\mathsf{M}}\}$$

is a set of named relations, and

$$Alp_{\mathsf{M}} \subseteq Act$$

and we call Alp_M the alphabet of M. *Let*

$$rel_{\mathsf{M}}(\mathsf{o}) \triangleq R \text{ iff } (\mathsf{o}, R) \in Npr_{\mathsf{M}}$$

$\square$

Note that M gives no meaning to elements of $Act \setminus Alp_{\mathsf{M}}$, i.e. operations not in the alphabet of M.

The standard formalisation of refinement uses programs built from a sequence of parts, first init, the initialisation of a semantics, followed by a sequence of operations.

Definition 2 *Programs. Let programs be defined by:*

$$Prog \triangleq \{\mathsf{init};\mathsf{s}|\mathsf{s} \in Act^*\}$$

and programs using only operations from Alp_{M}, the alphabet of semantics M, *be defined by:*

$$Prog_{\mathsf{M}} \triangleq \{\mathsf{init};\mathsf{s}|\mathsf{s} \in Alp_{\mathsf{M}}^*\}$$

The relational semantics of program $\mathsf{p} = \mathsf{init}; \mathsf{o}^1; \mathsf{o}^2 \ldots$ is defined to be the sequential composition of the relational semantics of its constituent operations ([23, 1]):

$$rel_M(\mathsf{p}) \triangleq \mathsf{init}_{\mathsf{M}}; rel_M(\mathsf{o}^1); rel_M(\mathsf{o}^2) \ldots$$

$\square$

Note that if some o^{i} is given no meaning by M then p is given no meaning either.

In the literature [12, 9, 1, 23] refinement $\sqsubseteq_L$ is defined, or by a slight rearrangement can be defined, by:

Definition 3 *Let* A *and* C *be relational semantics. Let $r \subseteq \Sigma_{\mathsf{A}} \times \Sigma_{\mathsf{C}}$ be the* retrieve *relation between the state of (abstract) semantics* A *and the state of (concrete) semantics* C. $\mathsf{A} \sqsubseteq_L \mathsf{C}$ *iff*

1. $Alp_{\mathsf{A}} = Alp_{\mathsf{C}}$*; and*
2. $\forall \mathsf{p} \in Prog_{\mathsf{A}}.rel_{\mathsf{C}}(\mathsf{p}) \subseteq rel_{\mathsf{A}}(\mathsf{p}); r.$ $\square$

It should be noted that (as in B) we do not consider a finalisation operation. Because of this in the second clause we have included $; r$ to relate the final states of the two programs.

3 Event-based LTS semantics

In this section we will define a fairly standard event-based operational semantics and in Section 3.1 define its relationship with the fairly standard state-based relational semantics of (collections of) operations from the previous section. This provides a route for the transfer of ideas from one world view to the other.

Definition 4 *LTS—labelled transition systems. Let N_{A} be a finite set of nodes. Let τ be a special unobservable operation.* A *is a LTS where:*

$$\mathsf{A} \triangleq (N_{\mathsf{A}}, S_{\mathsf{A}}, T_{\mathsf{A}}, Alp_{\mathsf{A}})$$

where

$$S_{\mathsf{A}} \subseteq N_{\mathsf{A}},$$

and we call S_{A} the start nodes of A,

$$T_{\mathsf{A}} \subseteq \{(n, \mathsf{a}, m)|n, m \in N_{\mathsf{A}} \wedge \mathsf{a} \in Alp_{\mathsf{A}} \cup \{\tau\}\},$$

and we call T_{A} the transitions of A, *and*

$$Alp_{\mathsf{A}} \subseteq Act,$$

and we call Alp_{A} the alphabet of A $\square$

In what follows we assume the existence of at least one start node $s_{\mathsf{A}} \in S_{\mathsf{A}}$.

Definition 5 *Paths. A* path *is a sequence of alternating nodes and operations. The set of paths generated by the LTS* A *is:*

$$Path_{\mathsf{A}} \triangleq$$
$$\{s_{\mathsf{A}}, \rho_1, n_2, \rho_2, \ldots |(s_{\mathsf{A}}, \rho_1, n_2), (n_2, \rho_2, n_3), \ldots \in T_{\mathsf{A}}\}$$

We write $|\rho|$ for the number of operations in a path and ρ^α for the sequence of operations in path ρ, so if $\rho = s_{\mathsf{A}}, \rho_1, n_2, \rho_2 \ldots$ then $\rho^\alpha \triangleq \rho_1, \rho_2 \ldots$.

For finite paths $\rho = s_{\mathsf{A}}, \rho_1, n_2, \rho_2, \ldots n_i$ define $last(\rho) \triangleq n_i$.

We will write ϵ for the empty sequence of operations. $\square$

Where A is obvious from context we write:

- $x \xrightarrow{\mathsf{a}} y$ for $(x, \mathsf{a}, y) \in T_{\mathsf{A}}$;
- $n \xrightarrow{\mathsf{a}}$ for $\exists m.(n, \mathsf{a}, m) \in T_{\mathsf{A}}$;
- $s_{\mathsf{A}} \xrightarrow{\rho^\alpha}$ when $\rho \in Path_{\mathsf{A}}$; and finally
- $s_{\mathsf{A}} \xrightarrow{\rho^\alpha} n$ when $\rho \in Path_{\mathsf{A}} \wedge last(\rho) = n$.

Definition 4 takes no account of τ operations being unobservable, and we call $\rightarrow$ a *strong semantics* and define the traces of A in:

Definition 6 *Traces of LTS* A.

$$Tr(\mathsf{A}) \triangleq \{\rho^\alpha | s_{\mathsf{A}} \xrightarrow{\rho^\alpha}\},$$

Trace refinement:

$$\mathsf{A} \sqsubseteq_{Tr} \mathsf{C} \triangleq Tr(\mathsf{C}) \subseteq Tr(\mathsf{A}).$$

$\square$

3.1 Relating state- and event-based semantics

We define a function lts that maps the state-based relational semantics A (Definition 1) into an event-based LTS semantics (Definition 3) and its inverse npr. Both mappings are little more than a reorganisation of definitions.

Definition 7 *For relational semantics*

$$\mathsf{M} = (\Sigma_{\mathsf{M}}, \mathsf{init}_{\mathsf{M}}, Npr_{\mathsf{M}}, Alp_{\mathsf{M}}),$$

we have

$$lts(\mathsf{M}) \triangleq (N_{\mathsf{M}}, S_{\mathsf{M}}, T_{\mathsf{M}}, Alp_{\mathsf{M}})$$

where

$$N_{\mathsf{M}} \triangleq \Sigma_{\mathsf{M}},$$

$$S_{\mathsf{M}} \triangleq range(\mathsf{init}_{\mathsf{M}}),$$

and

$$T_{\mathsf{M}} \triangleq \{(x, n, y) | (n, R) \in Npr_{\mathsf{M}} \wedge (x, y) \in R\}.$$

For LTS

$$\mathsf{A} = (N_{\mathsf{A}}, S_{\mathsf{A}}, T_{\mathsf{A}}, Alp_{\mathsf{A}})$$

we have

$$npr(\mathsf{A}) \triangleq (\Sigma_{\mathsf{A}}, \mathsf{init}_{\mathsf{A}}, Npr_{\mathsf{A}}, Alp_{\mathsf{A}})$$

where

$$\Sigma_{\mathsf{A}} \triangleq N_{\mathsf{A}},$$

$$\mathsf{init}_{\mathsf{A}} \triangleq \{(\bullet, n) | n \in S_{\mathsf{A}}\}$$

and

$$Npr_{\mathsf{A}} \triangleq \{(n, R) | n \in Alp_{\mathsf{A}} \wedge (x, y) \in R \Leftrightarrow x \xrightarrow{n} y\}$$

□

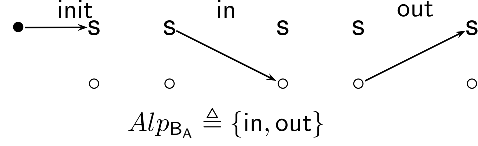
As we previously stated both operational semantics are open to many different interpretations so we view them as giving just part of the semantic story. By defining the translation between state-based systems and event-based systems on the operational semantics we have not restricted ourselves to a particular interpretation of the operational semantics.

Where not confusing we will give the same name to a machine (in the B sense, or a collection of operation schemas and state schemas in the Z sense, for example) and its relational semantics.

Lemma 1 *For relational semantics (machines)* A *and* C*, if* $Alp_{\mathsf{A}} = Alp_{\mathsf{C}}$ *then*

$$\mathsf{A} \sqsubseteq_L \mathsf{C} \Leftrightarrow lts(\mathsf{A}) \sqsubseteq_{Tr} lts(\mathsf{C})$$

□



$Alp_{\mathsf{B_A}} \triangleq \{\mathsf{in}, \mathsf{out}\}$

Figure 1. Relational semantics - $\mathsf{B_A}$

The very simple relation between the state- and event-based semantics shown in Definition 7 makes a number of event-based results from the literature available to state-based models and vice versa. But we must be careful about any informal interpretation we bring to any formal model.

The relational semantics in Figure 1 gives a state-based partial relation semantics of the buffer $\mathsf{B_A}$ (from Section 1) that we can transform, using lts, into the event-based LTS semantics in Figure 2 (left-hand LTS).

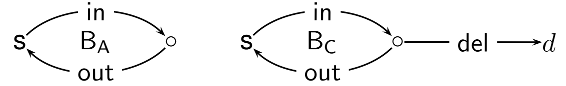


Figure 2. LTS Buffers

From the state-based perspective we know that these partial relations can be interpreted either as *guarded* outside of precondition or as *undefined* outside of precondition, similarly they can be interpreted either as a *totally-correct* model or as a *partially-correct* model, whereas in process algebra the LTS in Figure 2 would always be interpreted as *guarded* outside of precondition and to distinguish between totally- or partially-correct interpretations different refinements would be used.

4 Stuttering

The state-based world formalises an operation that that is both unobservable and has no effect as a $skip$ operation. Stuttering refinement has been defined so as to ignore these $skip$ operation. In the event-based world there are two operations that can not be observed, τ that can be performed but not observed and δ that does nothing, see [4] for details. The *unobservable* τ can be removed by *hiding* or *abstraction* where as the δ can be removed by *restriction* [4]. The two event-based unobservable operations (Definition 13) will be discussed in Section 5.

Data refinement $\sqsubseteq_L$ has been weakened [3] by replacing clause 1 with the following clause that permits stuttering steps to be added to the concrete machine. A concrete operation o is a stuttering step if it refines an abstract $skip \triangleq id_{\mathsf{A}}$ operation, where id_{A} is the identity relation over the state space of A.

Definition 8 *Stuttering refinement. Relational semantics (machine)* A *refines relational semantics (machine)* C *with stuttering,* $\mathsf{A} \sqsubseteq_S \mathsf{C}$*, where* A *and* C *have retrieve relation* r *between them, iff*

1. $Alp_{\mathsf{A}} \subseteq Alp_{\mathsf{C}}$ *and* $\forall\ \mathsf{o} \in Alp_{\mathsf{C}} \backslash Alp_{\mathsf{A}}$ *we have* $r; rel_{\mathsf{C}}(\mathsf{o}) \subseteq id_{\mathsf{A}}; r$ *and*
2. $\forall \mathsf{p} \in Prog_{\mathsf{C}}.rel_{\mathsf{C}}(\mathsf{p}) \subseteq rel_{\mathsf{A}}(\mathsf{p}); r$

□

Before we liberalise the definition of refinement we rephrase it so as to drop any reference to the alphabet of the machines.

To do this we assume that when a program calls an operation that is not in the alphabet of a machine then the program executes a $skip$ operation.

Definition 9 *We write* $[\![_]\!]_{\mathsf{M}}$ *for the semantic mapping that gives the relational semantics of both an operation* o *and a program* p*.*

$$[\![\mathsf{o}]\!]_{\mathsf{M}} \triangleq\ if\ \mathsf{o} \in Alp_{\mathsf{M}}\ then\ rel_{\mathsf{M}}(\mathsf{o})\ else\ id_{\mathsf{M}}$$

$$[\![\mathsf{p}]\!]_{\mathsf{M}} \triangleq \mathsf{init}_{\mathsf{M}}; [\![\mathsf{o}^1]\!]_{\mathsf{M}}; [\![\mathsf{o}^2]\!]_{\mathsf{M}} \ldots$$

□

Using the relations in Definition 9 we can drop the first clause in Definition 8.

Definition 10 *Machine refinement. Relational semantics (machine)* A *(machine) refines relational semantics (machine)* C*,* $\mathsf{A} \sqsubseteq_B \mathsf{C}$*, where* A *and* C *have retrieve relation* r *between them, iff*

$$\forall \mathsf{p} \in Prog.[\![\mathsf{p}]\!]_{\mathsf{C}} \subseteq [\![\mathsf{p}]\!]_{\mathsf{A}}; r$$

□

It is easy to see that this is no more that a rephrasing of refinement that permits stuttering steps.

Lemma 2 *For machines* A *and* C

$$\mathsf{A} \sqsubseteq_S \mathsf{C} \Leftrightarrow \mathsf{A} \sqsubseteq_B \mathsf{C}$$

□

4.1 Feature Refinement

In this section we formally introduce feature refinement as a liberalisation of $\sqsubseteq_B$. This parameterised definition of refinement is a specialisation of a more general definition of refinement found in [18]. It is a significant liberalisation of normal refinement and has been designed via its relation to simulation, as we will see in this section.

Whereas $\sqsubseteq_B$ considers all programs $Prog$ we simply allow the alphabet of the programs considered to be restricted.

Definition 11 *Feature Refinement. Let* $D \subseteq Act$ *and* A *and* C *be machines, and let* r *be a retrieve relation between them.*

Let

$$P \triangleq \{\mathsf{init}; \mathsf{s} | \mathsf{s} \in (Act \backslash D)^*\}$$

then

$$\mathsf{A} \sqsubseteq^D_{FR} \mathsf{C} \triangleq \forall \mathsf{p} \in P.[\![\mathsf{p}]\!]_{\mathsf{C}} \subseteq [\![\mathsf{p}]\!]_{\mathsf{A}}; r$$

□

It is important to note that all we are doing that is new is letting the set of programs be restricted to P and hence by definition $\sqsubseteq_B = \sqsubseteq^{\emptyset}_{FR}$. Again straight from the definition we can see that:

Lemma 3 $\mathsf{A} \sqsubseteq_B \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq^D_{FR} \mathsf{C}$ □

Now that we have refinement parameterised over the set D we explain its parameterised guarantee:

> feature refinement $\mathsf{A} \sqsubseteq^D_{FR} \mathsf{C}$ guarantees that any behaviour of machine C could be a behaviour of machine A when C is used by a program that calls operations from $Alp_{\mathsf{C}} \backslash D$.

4.2 Simulation

In order for our new definition to be of much use in practice we need to define sound simulation rules.

Definition 12 *Let* A *and* C *be machines and* r *a retrieve relation between them. Forward simulation* $\mathsf{A} \sqsubseteq^X_{FS} \mathsf{C}$ *holds iff:*

1. $[\![\mathsf{init}]\!]_{\mathsf{C}} \subseteq [\![\mathsf{init}]\!]_{\mathsf{A}}; r$ *and*
2. $\forall \mathsf{o} \in X.r; [\![\mathsf{o}]\!]_{\mathsf{C}} \subseteq [\![\mathsf{o}]\!]_{\mathsf{A}}; r$

□

This can be visualised as in Figure 3.

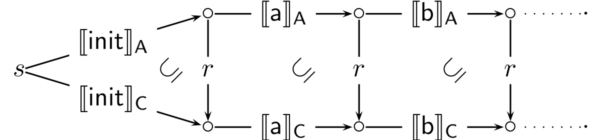


Figure 3. Forward simulation

The classic result that forward simulation is sound with respect to refinement still applies if we restrict both the alphabet used in refinement and the alphabet used in the definition of simulation.

Lemma 4 *Let* A *and* C *be machines,* $D \subseteq Act$ *and* $X \triangleq Act \backslash D$*. Then,*

$$\mathsf{A} \sqsubseteq^X_{FS} \mathsf{C} \Rightarrow \mathsf{A} \sqsubseteq^D_{FR} \mathsf{C}$$

□

5 Hiding and Restriction

The event-based literature supports a bottom-up approach to development by defining two operators, hiding and restriction, that build an abstract model from a more concrete model by treating unobservable events in two ways. We use them in our subsequent definitions of refinement relations in the more usual (for state-based developments) top-down style to build concrete models from more abstract models.

Definition 13 *Restriction and hiding on LTS* $\mathsf{A} = (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}}, Alp_{\mathsf{A}})$.

$$\mathsf{A}\delta_D \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}\delta_D}, Alp_{\mathsf{A}\delta_D})$$

where

$$D \subseteq Act,$$

and D *is the set of operations to restrict (i.e. make unobservable and blocked),*

$$T_{\mathsf{A}\delta_D} \triangleq \{n \xrightarrow{\mathsf{a}}_{\mathsf{A}\delta_D} l \mid n \xrightarrow{\mathsf{a}}_{\mathsf{A}} l \wedge \mathsf{a} \notin D\}$$

and

$$Alp_{\mathsf{A}\delta_D} \triangleq Alp_{\mathsf{A}} \backslash D.$$

$$\mathsf{A}\tau_T \triangleq (N_{\mathsf{A}}, s_{\mathsf{A}}, T_{\mathsf{A}\tau_T} Alp_{\mathsf{A}\tau_T})$$

where

$$T \subseteq Act,$$

and T *is the set of operations to hide (i.e. make unobservable and unblockable),*

$$T_{\mathsf{A}\tau_T} \triangleq$$

$$\{n \xrightarrow{\mathsf{a}}_{\mathsf{A}\tau_T} l \mid n \xrightarrow{\mathsf{a}}_{\mathsf{A}} l \wedge \mathsf{a} \notin T\} \cup \{n \xrightarrow{\tau}_{\mathsf{A}\tau_T} l \mid n \xrightarrow{\mathsf{a}}_{\mathsf{A}} l \wedge \mathsf{a} \in T\}$$

and

$$Alp_{\mathsf{A}\tau_T} \triangleq Alp_{\mathsf{A}} \backslash T.$$

□

To model τ operations as unobservable we define how to abstract (remove) them from a LTS:

Definition 14 *Observational semantics* $\Longrightarrow$. *For LTS* A,

$$n \overset{\mathsf{a}}{\Longrightarrow} m \triangleq n \overset{\tau}{\Longrightarrow} n', n' \xrightarrow{\mathsf{a}} m', m' \overset{\tau}{\Longrightarrow} m \wedge \mathsf{a} \in Act$$

where

$$s \overset{\tau}{\Longrightarrow} t \triangleq s \xrightarrow{\tau} s_1, s_1 \xrightarrow{\tau} s_2, \ldots s_{n-1} \xrightarrow{\tau} t \vee s = t$$

Also,

$$Abs(\mathsf{A}) \triangleq (N_{\mathsf{A}}, S_{\mathsf{A}}, \{(n, x, m) | n \overset{x}{\Longrightarrow} m\}, Alp_{\mathsf{A}}).$$

and hiding is given by:

$$\mathsf{C} \backslash \mathsf{T} \triangleq Abs(\mathsf{C}\tau_{\mathsf{T}})$$

□

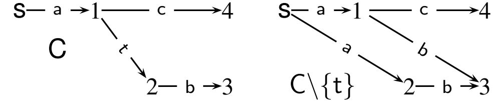


Figure 4. Hiding the event t, i.e. making it unobservable and unblockable and then abstracting

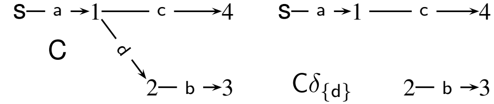


Figure 5. Restricting the event d, i.e. making it unobservable and blocked

Hiding and restricting are illustrated in Figure 4 and Figure 5. Having explained event-based restriction ($_\delta_D$) and hiding ($_\backslash\mathsf{T}$ or $Abs(_\tau_T)$) that allow us to move from the concrete to the abstract we reverse them to give two refinement steps from the abstract to the concrete.

Definition 15 *Tau- and delta-refinement. Let* A *and* C *be machines and* $New \subseteq Alp_{\mathsf{C}} \backslash Alp_{\mathsf{A}}$ *be a set of new observable operations. Then:*

$$\mathsf{A} \sqsubseteq_{Tr\delta New} \mathsf{C} \triangleq \mathsf{A} \sqsubseteq_{Tr} \mathsf{C}\delta_{New}$$

$$\mathsf{A} \sqsubseteq_{Tr\backslash New} \mathsf{C} \triangleq \mathsf{A} \sqsubseteq_{Tr} \mathsf{C} \backslash New$$

□

The event-based delta-refinement appears in [11] as a behavioural subtype. By applying the definition to the operational semantics we are able via Definition 7 to "port" it to state-based models.

Lemma 5 *Let* A *and* C *be machines and* $N = Alp_{\mathsf{C}} \backslash Alp_{\mathsf{A}}$. *Then:* $\mathsf{A} \sqsubseteq_B \mathsf{C} \Leftrightarrow lts(\mathsf{A}) \sqsubseteq_{Tr\backslash N} lts(\mathsf{C})$ □

Viewed from the state-based perspective the event-based action abstraction mirrors stuttering, as is well known. Thus t, in Figure 6, is a stuttering step. From the state-based perspective $\mathsf{C}\backslash\{\mathsf{t}\}$ is the abstract machine A and the effect of the concrete program init; a; t; b in Figure 6 is the same as the effect of the abstract program init; a; b.

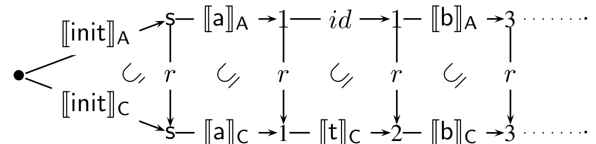


Figure 6. $\mathsf{t} \in Alp_{\mathsf{C}} \backslash Alp_{\mathsf{A}}$

The removal of τ events has been widely studied in the event-based literature (for details see [15, 8, 22, 19]) and because of lack of space we do not wish to repeat the discussion here. But we do wish to remind the reader that there are many different, valid, interpretations given to τ-loops.

In the CSP failure/divergence semantics τ-loops are interpreted as *diverging*, that is having *chaotic* behaviour "whether it is true or not" [20, p95]. In CCS [15] and the fair failure semantics of [8, 16] τ-loops can simply be ignored and for an interpretation neither based on fairness nor chaos we have CFFD and NDFD in [22] and for an operational model neither based on fairness nor chaos see [19].

Some state-based models introduce rules, notably WED_EF in Event B [14], that prevent the introduction of τ-loops. In Section 6.1 we take an example from the literature where the introduction of τ-loops was usefully given a fair interpretation. Consequently we will adopt the fairness assumption.

Lemma 6 *Let* A *and* C *be machines,* $T \subseteq Act$*,* $D \subseteq Act$*,* $T \cap D = \emptyset$ *and* $T = Act \backslash (Alp_{\mathsf{A}} \cup D)$*. Then:*

$$\mathsf{A} \sqsubseteq^{D}_{FR} \mathsf{C} \Leftrightarrow lts(\mathsf{A}) \sqsubseteq_{Tr\backslash T} lts(\mathsf{C})\delta_D$$

□

5.1 Old ideas in new places and put to new uses!

We have combined, in novel ways, old ideas from a variety of places. While hiding is likely to be familiar restriction is, from a state-based perspective, less so. Therefore, to clarify things we review the event-based notion of restriction from a state-based perspective by returning to the buffer example from Section 1.

It is clear from the LTS in Figure 2 that $\mathsf{B_A} \sqsubseteq_{Tr\delta\{\mathsf{del}\}} \mathsf{B_C}$, but how are we to interpret this? It should be noted that $\sqsubseteq_{Tr\delta\{\mathsf{del}\}}$ formalises a design decision, i.e. tells us something about the relation between the abstract $\mathsf{B_A}$ and the concrete $\mathsf{B_C}$. $\mathsf{B_A}$ specifies the correct behaviour of a buffer; it tells us nothing about error events, such as del. They are assumed to not occur (i.e. are blocked) in the abstract $\mathsf{B_A}$.

So, when $\mathsf{B_A}$ was originally formulated only the behaviour of operations in and out were considered. Subsequent design decisions to model errors introduced the event del that was previously neither observable nor executable (blocked), but on programs where del does not appear, $\mathsf{B_A}$ and $\mathsf{B_C}$ have the same meaning, so $\mathsf{B_C}$ is as acceptable as $\mathsf{B_A}$. Hence the delta-refinement.

As we said in Section 1 we wish to be able to define the buffer in such a way as to prevent *or* allow the introduction of certain events. We can easily prevent the introduction of del in future development of $\mathsf{B_A}$ by expanding the alphabet while keeping the picture (Figure 2) the same. Thus if we define $\mathsf{B_{Ad}}$ to have the same transitions and initial state as $\mathsf{B_A}$ but arrange for its alphabet to include del, i.e. $Alp_{\mathsf{B_{Ad}}} \triangleq Alp_{\mathsf{B_A}} \cup \{\mathsf{del}\}$, then clearly del cannot be added again and $\mathsf{B_{Ad}} \not\sqsubseteq_{Tr\delta\{\mathsf{del}\}} \mathsf{B_C}$.

We have not found, in the state-based literature, a definition that mirrors this abstract definition of delta-refinement, though Event B has a definition of refinement permitting new events to be introduced where "the only constraint on these events is that they maintain the local invariant" [2].

Delta-refinement is a liberalisation of refinement in [2]. It should be noted that the refinement in [2] can be applied to none of our examples. This is so for three reasons: one, the operations can be *undefined* on some of the domain and hence have a different semantics to Event B; two, in none of our examples is the local invariant preserved by the new events; and three, the refinement in [2] excludes the introduction of τ-loops.

6 Feature refinement in B

In this section we show how the syntax of B machines can be extended to allow the liberalisation of refinement to be incorporated into B-style development.

To distinguish tau-refinement from delta-refinement we introduce two clauses to the B syntax: DELTAOPS D and TAUOPS T, where D and T are sets of operations.

When DELTAOPS D appears in a MACHINE M the D operations must not be defined in the OPERATIONS clause (because they have to be blocked) and this clause alters the semantics of the machine only by adding the D operations to the alphabet of the machine. Consequently we define Alp_{M} to be the union of the operations in the OPERATIONS and DELTAOPS clauses.

In a REFINEMENT C let the operations in the OPERATIONS clause be O_{C} and New be the operations not in the MACHINE A it refines, $New = O_{\mathsf{C}} \backslash Alp_{\mathsf{A}}$.

When both DELTAOPS D and TAUOPS T clauses appear in a REFINEMENT C we require that $T \cap D = \emptyset$, because obviously an operation cannot be both blocked and unblocked, and $T = New$ since all the TAUOPS have to be (newly) defined. These clauses indicate which type of abstract operation has been refined by the introduction of the new concrete operation. The TAUOPS clause is not strictly needed but is added both for clarity and the above consistency checking.

Whenever DELTAOPS D appears in a REFINEMENT the refinement is taken to be $\sqsubseteq^{D}_{FR}$ and if no DELTAOPS clause appears then D is taken to be empty, so the refinement is $\sqsubseteq^{\emptyset}_{FR}$, i.e. $\sqsubseteq_S$ or, equivalently, $\sqsubseteq_B$.

As C is guaranteed to behave like A only for programs that do not call operations in D it is clear that the D operations require *no proof obligation* as no guarantee needs to be satisfied (recall refinement in [2]). We illustrate this in the next section.

6.1 Example - Mobile Radio

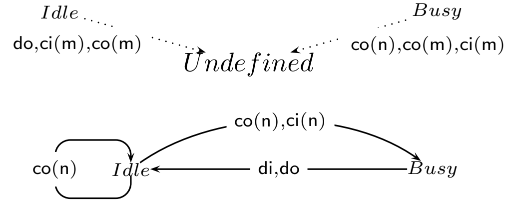
For our second example we use the specification of a Mobile Radio from [6] where the difficulties refining the high-level mobile radio $HLMR$ (Figure 7) are used to motivate the use of retrenchment.

MACHINE $HLMR$
SETS $CALLS = \{Idle, Busy\}$
VARIABLES $callState, currChan$
INVARIANT $callState \in CALLS \wedge$
$currChan \in CHAN$
INITIALISATION $callState := Idle \parallel$
$currChan :\in CHAN$
OPERATIONS
do $\triangleq$ PRE *callState* = *Busy* THEN
callState := *Idle* END;
di $\triangleq$ SELECT *callState* = *Busy* THEN
callState := *Idle* END;
co(x) $\triangleq$ PRE $callState = Idle \wedge x \in CHAN$ THEN
CHOICE $callState := Busy \parallel currChan := x$
OR $skip$ END
END;
ci(x) $\triangleq$ PRE $x \in CHAN$ THEN
SELECT $callState = Idle$
THEN*callState* := *Busy* $\parallel$ *currChan*:= x
ELSE $skip$ END
END;
END

Figure 7. HLMR

The radio is either in an $Idle$ or a $Busy$ state and the channel number is always set to an element of $CHAN$. The radio is initially in $Idle$. The call incoming action ci(x) has a channel number as parameter . The set of values that this parameter ranges over can be partitioned into two sets and we will assume n $\in CHAN$ and m $\notin CHAN$ in what follows.

The LTS semantics for these machines can be messy and dominated by the transitions used to represent the undefined parts of the operations. Purely as sugar we split the LTS semantics into two LTS (see Figure 8[1]). In the bottom LTS of Figure 8 we have included only the active and guarded parts of the events and have omitted the undefined parts. The undefined parts appear in the top LTS. We can build a LTS for HLMR by adding the undefined parts to the bottom LTS. This can be achieved by adding $\{n \xrightarrow{\mathsf{a}} x | x \ a \ state \ of$ HLMR$\}$ for each $n \xrightarrow{\mathsf{a}} Undefined$ in the top LTS.



Guaranteed actions of HLMR

Figure 8. High Level Mobile Radio

The specification makes no guarantee as to its behaviour if one of the undefined operations is called.

The high-level Mobile Radio has been partially specified. How it behaves with operations ci(m) and co(m) is completely undefined, as are operation do from state $Idle$ and operation co(n) from state $Busy$, and note that it is blocked from the operation di in state $Idle$.

The error features we wish to add require the addition of new operations fade, reset, sel. In $HLMR$ we have not considered the ability of the radio to fade. That is to say we have made the simplifying assumption that radio will not fade.

The less abstract, lower-level view of the mobile radio $LLMR$ takes into account new features, in particular three new operations that do not appear anywhere in the high-level specification $HLMR$: when the radio is *Busy* it may fade and when a fade occurs the radio is *Jammed*; when the radio is *Jammed* it must be reset to the *Idle* state. This specification is very weak, it assumes that reset will only be called when $callState = Jam$; before the radio will work the user must select a suitable wave band.

Let us assume that the $LLMR$ description is a more accurate depiction of the actual radio. We can view the refinement from $HLMR$ to $LLMR$ as adding these new features, while still providing the service guaranteed in $HLMR$.

This difference in the interpretation of the new operations is reflected in how their non-appearance in the high-level view $HLMR$ is "explained" within $LLMR$, i.e. that they are either TAUOPS or DELTAOPS.

The $LLMR$ machine (Figure 9) is taken from [6] but with some small amendments. Firstly the TAUOPS and DELTAOPS clauses state how to abstract the new operations and hence which proof obligations should be applied. The TAUOPS clause means that the usual B proof obligations for such operations are used for the new operations, and the DELTAOPS clause needs *no* proof obligations because no guarantee needs to be made since the operations

[1] It is well-known from the literature that total correctness semantics is easier to model using total relations than using partial relations. Standard ways to lift and totalise partial relations so as to give a guarded outside of precondition interpretation can be found in the literature. Here, as is commonly done, we represent the semantics of operations using the underlying partial relation, the only difference being that we transform the semantics into LTS via our linking function lts.

REFINEMENT	$LLMR$
REFINES	$HLMR$
TAUOPS	sel,reset
DELTAOPS	fade
SETS	$JCALLS = CALLS \cup \{Jam\}$
VARIABLES	$jcallState, jcurrChan, bandSelected$
INVARIANT	$jcallState \in JCALLS \wedge$

$bandSelected \in Bool \wedge jcurrChan \in CHAN \wedge$
$(callState, jcallState) \in$
$\{(Idle, Idle), (Idle, Jam), (Busy, Busy)\} \wedge$
$currChan = jcurrChan \wedge$
$(bandSelected = FALSE \Rightarrow jcallState = Idle)$
INITIALISATION $jcallState = Idle \parallel$
$bandSelected = FALSE \parallel jcurrChan :\in CHAN$
OPERATIONS
sel $\triangleq$ SELECT $band_selected = FALSE$
THEN $band_selected := TRUE$ END;
reset $\triangleq$ SELECT $callState = Jam$
THEN $callState := Idle$ END;
fade $\triangleq$ SELECT $callState = Busy \wedge$
$band_selected = TRUE$ THEN
$callState := Jam$ END;
do $\triangleq$ PRE $jcallState = Busy \wedge$
$bandSelected = TRUE$
THEN *jcallState* := *Idle* END;
di $\triangleq$ SELECT $jcallState = Busy \wedge$
$bandSelected = TRUE$
THEN *jcallState* := *Idle* END;
co(x) $\triangleq$ PRE $jcallState = Idle \wedge x \in CHAN \wedge$
$bandSelected = TRUE$
THEN
CHOICE $jcallState := Jam$ OR $skip$ OR
$jcallState := Busy \parallel jcurrChan := x$ END
END;
ci(x) $\triangleq$ PRE $x \in CHAN$ THEN
SELECT $jcallState = Idle \wedge$
$bandSelected = TRUE$
THEN
$jcallState := Busy \parallel jcurrChan := x$
ELSE $skip$ END END;
END

Figure 9. LLMR

are blocked. Secondly the INVARIANT is changed to define the reachable set of nodes. Without this the proof obligations for the refinement of operation co(x) from $HLMR$ to $LLMR$ could not be satisfied.

In the LTS for *LLMR* (Figure 10) states where $band_selected = FALSE$ are, to save space in the diagram, represented with $(bsf, _)$, while those where $band_selected = TRUE$ are left unamended. (Note that the unreachable states (bsf, Jam) and $(bsf, Busy)$ may be "deleted" by subsequent refinement).

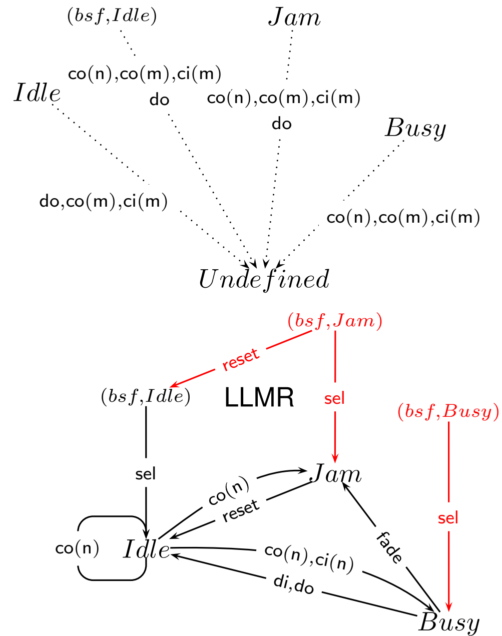


Figure 10. Low Level Mobile Radio

Note the difference between the TAUOPS operations reset and sel and the DELTAOPS operation fade: while fade can be simply deleted from $LLMR$ without breaking the specification given by $HLMR$, reset cannot be simply deleted but has to become a $skip$ in order that in $HLMR$ co(n) loops back to $Idle$; also sel cannot simply be deleted either since otherwise the initial state $(bsf, Idle)$ would have no transitions leading from it!

The "DELTAOPS {fade}" clause in the REFINEMENT (Figure 9) is not giving a meaning to the fade operation in the concrete machine *LLMR*. It is defining what type of refinement is being performed and hence can be viewed as giving meaning to the relation between *HLMR* and *LLMR*. More specifically it tells us that in the *HLMR* fade operations were considered to not occur (they were blocked).

If we were to add "DELTAOPS {fade}" to *HLMR* then because the fade operation would now be in the alphabet we know that it has been *considered* but because it is new, i.e. always blocked, the specification is saying that it *cannot be performed* from any state. Consequently we would no longer be able to use feature refinement to build *LLMR*. Recall the effect of adding of the DELTAOPS clause is simply to add the fade operation to Alp_{HLMR}. Thus the set of programs considered in definition of $\sqsubseteq^{\emptyset}_{FR}$ is enlarged to include fade operations.

7 Conclusion

Refinement as usually understood in the state-based world (e.g. Z, B and so on) is frequently found to be too restrictive or conservative [6, 5] in that it does not recognise as a refinement some useful and uncontroversial development steps.

By using an obvious and well known relation between state- and event-based operational semantics we have been able to combine known results from state- and event-based models to create feature refinement, $\sqsubseteq_{FR}^{D}$, a new (to the state-based world) and flexible definition of refinement. Interestingly we are able to see in what contexts C behaves like A when $\mathsf{A} \sqsubseteq_{FR}^{D} \mathsf{C}$: they turn out to be simply the programs over $Alp_{\mathsf{C}} \setminus D$.

The usefulness of this new refinement has been shown by applying it to an example (Section 6.1) taken from the literature.

The closest state-based definition of refinement to feature refinement that we can find in the literature is Event B refinement. But feature refinement and Event B refinement differ in the following three ways: one, feature refinement makes no restriction on the machine INVARIANT; two, τ-loops are treated fairly; and three, feature refinement is equally applicable whatever (different) interpretations are given to the operational semantics.

Acknowledgements

We thank various referees for their comments, and the Foundation for Science, Research and Technology (FRST) of New Zealand for funding this research.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial, D. Cansell, and D. Méry. Refinement and reachability in Event B. In H. Treharne, S. King, M. C. Henson, and S. Schneider, editors, *ZB05: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer, 2005.

[3] R.-J. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3):324–346, 1996.

[4] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, 1990.

[5] R. Banach and J. Derrick. Filtering retrenchments into refinements. In *SEFM*, pages 60–69. IEEE, 2006.

[6] R. Banach and M. Poppleton. Retrenchment, refinement and simulation. In *Proc. ZB-00*, volume 1878 of *LNCS*, pages 304–323, 2000.

[7] C. Bolton and J. Davies. A singleton failures semantics for Communicating Sequential Processes. *Formal Aspects of Computing*, 18:181–210, 2006.

[8] E. Brinksma, A. Rensink, and W. Vogler. Fair testing. *LNCS 962*, pages 313–327, 1995. Springer-Verlag.

[9] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model oriented proof methods and their comparison*. Cambridge Tracts in theoretical computer science 47, 1998.

[10] J. Derrick and E. Boiten. Relational concurrent refinement. *Formal Aspects of Computing*, 15(2):182–214, November 2003.

[11] C. Fischer and H. Wehrheim. Behavioural subtyping relations for object-oriented formalisms. *LNCS*, 1816:469–483, 2000.

[12] J. He, C. Hoare, and J. Sanders. Data refinement refined. *ESOP 86 LNCS*, 213:187–196, 1986.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[14] C. Metayer, J.-R. Abrial, and L. Voisin. Event-B language. RODIN Project Deliverable D7, May 2005.

[15] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[16] V. Natarajan and R. Cleaveland. Divergence and fair testing. In Z. Fülöp and F. Gécseg, editors, *ICALP*, volume 944 of *LNCS*, pages 648–659. Springer, 1995.

[17] K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programing*, 25((2-3)):285–327, 1995.

[18] S. Reeves and D. Streader. Comparison of Data and Process Refinement. In J. S. Dong and J. C. P. Woodcock, editors, *ICFEM 2003*, LNCS 2885, pages 266–285. Springer-Verlag, 2003.

[19] S. Reeves and D. Streader. Atomic Components. In *ICTAC 2004*, LNCS 3407, pages 128–139. Springer-Verlag, 2004.

[20] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science, 1997.

[21] J. M. Spivey. *The Z notation: A reference manual*. Prentice Hall, 1989.

[22] A. Valmari and M. Tienari. Compositional Failure-based Semantics Models for Basic LOTOS. *Formal Aspects of Computing*, 7(4):440–468, 1995.

[23] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.