Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 207 (2008) 187–202
www.elsevier.com/locate/entcs

Generic Tools via General Refinement

Steve Reeves David Streader

Department of Computer Science, University of Waikato, Hamilton, New Zealand

Abstract

Tools have become essential in the formal model-driven development of software but are very time consuming to build and often restricted to a particular semantic interpretation of a particular syntax. This is regrettable since there is large amount in common between tools, even if they do "implement" different syntaxes and different semantics.
We propose splitting tools into front- and back-ends where an operational semantics acts as the link between the two. We will not have much to say about the front-end and the link in this paper since it is theoretically straightforward. Instead, we concentrate on the second part and provide a well-motivated, general, mathematical framework to form the underlying theory that gives great flexibility to the back-end of a tool which is concerned with developing software via stepwise refinement.
From a general model of refinement between two entities, where the refinement is parameterised on contexts and observations, we build logical theories which have refinement as implication. Further, we consider what can be expected of a guarantee concerning the behaviour of an implementation relative to a specification. Then by fixing the contexts and observations in suitable ways, and so getting particular, special models of refinement, we give a formal interpretation of a guarantee. To this we add theory morphisms between special models, where a theory morphism can change the contexts and observations we can make in controlled and useful ways, mainly by preserving a refinement relation between entities even as we change them.
We show how the generality brought about by the parameterisation allows an example from the literature, which seems formally not to be a refinement, to be captured as a refinement, in accordance with our intuitions about the example.
In this way we show the flexibility of our theory for a refinement tool back-end. From this it follows that the effort put into building a tool based on our theory will be well-spent—a single tool should be parameterised (just as our theory is) to deal with the many different notions of refinement found in the literature. Thus we make a contribution to the problem of ensuring correctness and dependability of software using formal methods and tools of modelling, design, verification and validation.

Keywords: basis for tools, refinement, theory morphism, flexible refinement

1 Introduction

Refinement is used to model the design decisions made in the stepwise development of an abstract specification to an implementation or more concrete specification. In practice refinement requires a large number of small and uninteresting details to be checked with great accuracy. Undertaking such work by hand is very error prone and tool support is vital for refinement to be widely applied to anything other than toy examples.

Refinement is formalised to provide a guaranteed relation between the abstract specification and the implementation. Knowing how refinement is formalised allows

1571-0661 © 2008 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2008.03.093

the person writing a specification to know in what way the implementation will *satisfy* it.

We note that refinement is defined in different ways in different formalisms *e.g.* CSP, IOA, B, Event B, Z *etc.*, and these differences can be subtle and make it hard to relate the different definitions. This may be important if, for example, we wish to specify some parts of a system in one language and other parts in another language. For example event-based specification may be appropriate for the interactive parts of a system and state-based specification may be appropriate for the purely transformational parts.

It turns out that if we keep sight of both the contexts we place systems in and the observations a user can make, then a single general definition of refinement can be given. This can be specialised to particular refinement relations with appropriate (depending on the problem domain) properties by fixing (instantiating) the contexts and observations to give a more specialised, concrete definition of refinement.

We have shown [1,2] that contexts and observations can be selected in such a way that some of the well-known refinements found in the literature can be expressed as specialisations of our general model. Further we have shown [3] that using our general model of refinement we are able to *transfer* a definition of refinement from one (state-based) formalism to a different (event-based) formalism in which setting it appears to be new. The bridge between state- and event-based formalisms used in [3] is the well-known isomorphism between a state-based operational semantics (sets of named partial relations, Npr) and an event-based operational semantics (labelled transition systems, LTS).

Taking a very high-level view for a moment we can say there is a wide variety of different formal models, each with different syntax. CSP, CCS, IOA, CBS, Z, B, Event B *etc.* all have an operational semantics based on LTS or Npr. Thus we can use the operational semantics as a *common intermediate language*.

We are not going to discuss the translation into this common (up to isomorphism) operational semantics since it is quite straightforward (if intricate). Rather we are going to focus our attention on a general theory that permits:

(i) different interpretations of the operational semantics;

(ii) formalises changes in the interpretation as a refinement step.

The formal model we will discuss is of interest from the tool-building perspective for two important reasons:

(i) definitions and results can be given at a general level and consequently are applicable for a variety of interpretations of the operational semantics;

(ii) it allows a great deal of flexibility in what can be considered as refinement, *i.e.* what development steps are permitted.

An illustration of this first point can be seen in [3] where an existing event-based definition of refinement is re-expressed and then re-established at a more general level, and subsequently applied with different state-based interpretations of the operations being made. The second point above has, to some degree, been

illustrated in [1,3], but here we will give a further and, we believe, more convincing illustration.

To summarise: we believe that there are advantages to developing tools in two parts—the front-end that translates languages of various types into an operational semantics that takes the role of a common intermediate language, and the back-end that manipulates the operational semantics. The front-end is important, of course, but technically straightforward. Consequently, we restrict ourselves here to the definition of a general formal model from which a flexible back-end might be built.

1.1 Flexible development

The very reason for building a specification before an implementation is that the specification can be more abstract than the implementation, the idea being that certain "important" features can be considered without the clutter of unnecessary detail. We need to remember the top-level specification needs to be both written and understood by a person. By allowing as much flexibility as is practical when deciding what features to focus on and what to abstract we aid both the writing and comprehension of a specification.

For example programmers frequently consider data structures (lists, trees, sets *etc.*) without fixing the maximum size they can grow to. Nonetheless, at some point in the design, maybe near the end of the process, the maximum size must be fixed (even if only by accepting some system defaults). Traditionally the fixing of the size would not be considered a formal refinement but some other informal design step. What we do here is relax the formal definition of refinement so that this step can be viewed as a refinement, with a guaranteed relation between the abstract specification and the implementation.

We take as a running example an abstract specification of a data structure representing a set of undetermined size. Then we show how to formally refine this specification into a more concrete specification of a set of a given size.

In order to model this design step we have had to take a relaxed view of both operation refinement and data refinement. Rather than define two new definitions of refinement, one for operation refinement and the other for data refinement, we have defined one general notion and then shown how to specialise this to both operation and data refinement. So, thinking back to this being a theory for the basis of a tool, we see that by implementing flexible refinement we can build other, more specialised, definitions of refinement (like operation and data refinement shown here) simply by defining the values to two parameters for each specialised refinement.

Our running example is not of our making but has been taken from the retrenchment literature as a natural and practical design step, yet it cannot be formalised (to the best of our knowledge) by any known refinement in the literature. Indeed for a class of problems retrenchment [4] can be replaced by flexible refinement thus maintaining a guaranteed relation between the specification and implementation.

In Section 2 we give an informal definition of refinement and a motivating example. The technical detail of the semantics on which to base the flexible back-end of a tool is given in Section 3. Then in Section 4.1 and Section 5 we show how to apply

our definitions to single operations and machines (collections of named operations and private state) respectively. Using this in Section 6 we formalise as refinement the development step of our example, and in Section 7 we conclude.

2 Conceptualisation

Our starting point is the following natural notion of refinement that appears in many places in the literature [5,6,7,8,9,10] and can be applied to operations, processes, machines *etc.*, all of which we refer to as *entities*:

> The concrete entity C is a refinement of an abstract entity A when no user of A could observe if they were given C in place of A.

Thus the details of refinement both define and are defined by the interpretation of the *guarantee* that C satisfies (behaves like) the specification A.

2.1 Motivating Example

This paper is about the refinement of entities represented by some operational semantics. It does not matter to us what syntax is used to define the entities. We take our example, formalised in Z, from [4], but wish to stress our paper is not about Z. We are simply using Z to express an entity because the published example did.

All the reader needs to know about Z is that state spaces and operations over them are defined by schemas: named boxes with declarations above the dividing line and predicates giving properties below the line. Operations are then to be understood as relations between "before and after" states, or pre- and post-states, using the useful convention that pre-state observation names are unprimed, *e.g.* s, and post-state observation names are primed, *e.g.* s'. This priming convention is also applied to state schemas, so $State'_{\mathsf{A}}$ has an observation named s'. [1]

The abstract definition $\mathsf{Set_A}$ of a (data structure for a) set containing natural numbers with two operations $\mathsf{Put_A}$, to add numbers to the set, and $\mathsf{Get_A}$, to remove them, can be found in Fig 1. [2]

It is plainly not possible to implement such a definition. Computers have a finite amount of storage and hence a program that is repeatedly executing $\mathsf{Put_A}$ will, at some point, simply run out of space. A more concrete, and now implementable, definition, $\mathsf{Set_C}$, with the size of the set bounded by three, can be found in Fig 1 too (for the moment ignore mention of the set $\{X\}$).

If $\mathsf{Set_A}$ is interpreted as a specification guaranteeing that an infinite number of $\mathsf{Put_A}$ operations, each with distinct inputs, can be successfully called then $\mathsf{Set_C}$ certainly does not meet this guarantee. Consequently if refinement is meant to

[1] Seasoned Z readers will note conventions that we might have followed to make our Z more standard—we have omitted these since, as we said, this paper is not about Z, does not depend on it and no knowledge of Z is needed to read it.

[2] We note that, in fact, since Z is strongly typed and so $\cup$ must join two things of the same type, the type $\mathbb{P}\,\mathbb{N} \cup \{X\}$ in this example as it stands cannot properly be said to be Z. However, with more work, we could make this proper Z, but it would complicate, somewhat, what is written for the type concerned and would therefore distract us from the point of this paper. Finally, we have used exactly the example from [4].

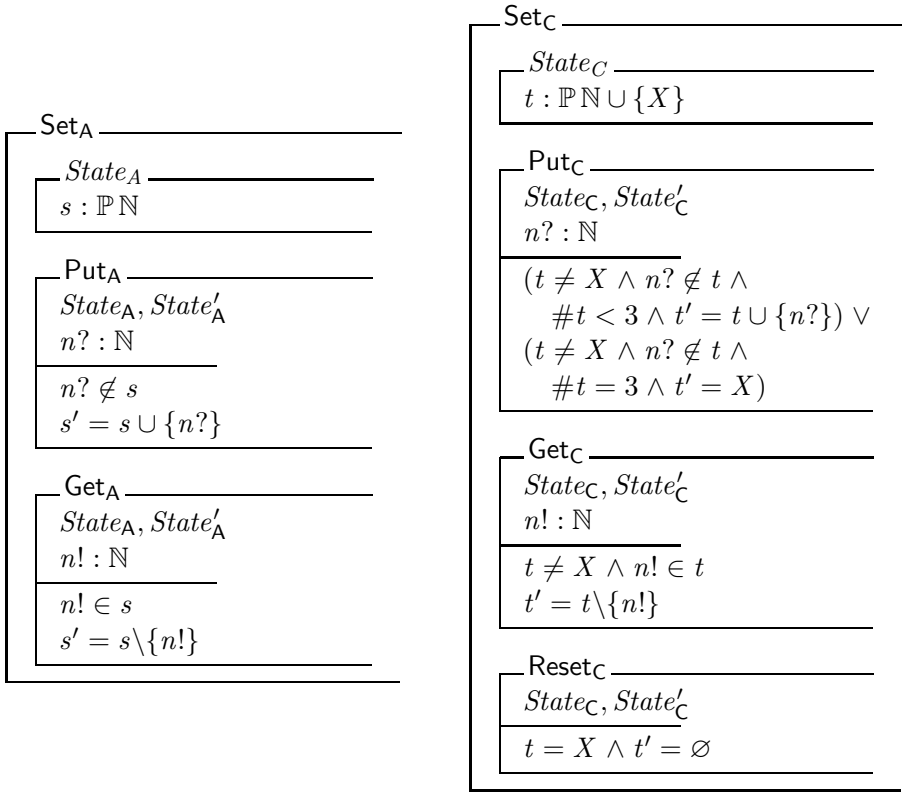

Fig. 1. Infinite $\mathsf{Set_A}$ and bounded $\mathsf{Set_C}$

capture this guarantee then $\mathsf{Set_C}$ is not a refinement of $\mathsf{Set_A}$. Nonetheless, if we accept that in some "practical situations" any reasonable person might wish to insist on viewing $\mathsf{Set_C}$ as a refinement of $\mathsf{Set_A}$ then reasonable people cannot be interpreting $\mathsf{Set_A}$ as guaranteeing so much. So, the notion of what a guarantee is needs to be considered.

In an early step in the development of a system we might specify $\mathsf{Set_A}$ because a set will be needed for the *correct behaviour* of the system. In some subsequent step, when considering the *error behaviour* we specify the maximum size of the set. We would then wish to refine the abstract $\mathsf{Set_A}$ into the more concrete $\mathsf{Set_C}$, as proposed in [4].

Given that we would like $\mathsf{Set_C}$ to be a refinement of $\mathsf{Set_A}$ we can, informally speaking, ask for a guarantee that $\mathsf{Set_C}$ behaves just like $\mathsf{Set_A}$ in contexts satisfying the following assumptions:

(i) the set is not used to store more than three different numbers; and

(ii) only the "put" and "get" operations are called.

This guarantee is certainly weaker than the (unreasonable because unimplementable) guarantee we started with, but it seems to be the strongest guarantee we can expect, and, crucially, it is useful and, probably, all we were expecting all along

(being reasonable people).

We will show how to formally model the development of $\mathsf{Set_A}$ into $\mathsf{Set_C}$ as a refinement step and show that its formal guarantee corresponds to the above informal guarantee. Clearly the example is very small and we could easily have given the concrete specification in the first place. The point we make, though, is that if one were given a *large* complex specification on which a lot of time has been spent, then one would be reluctant to throw away all this effort and start again.

3 Formalisation

Note that the informal notion of refinement in Section 2 talks about not only the entities involved in the refinement, but also the observations a user can make of them. Also, since the user, in order to make observations, must presumably use the entities they must have been placed in some contexts (*e.g.* programs which call the operations the entities provide). We should be careful when formalising refinement not to lose track of, or throw away, these contexts and observations. They were important enough to be employed in the informal notion of refinement, so they might also turn out to be useful in the formalised version too.

We will give a formal general definition of refinement with explicit parameters representing both Ξ, the contexts in which A and C will be placed, and O, a function from entities to sets of traces $\wp(\mathbb{O})$ (*e.g.* of event names or states). Where each trace $tr \in \mathbb{O}$ is a potential observation. Our definition will have the following useful features:

One we can construct a guarantee that C *satisfies* A that is parameterised on both Ξ and O;

Two we can construct a simple logical theory, based on $\Xi \times \mathbb{O}$ relations, where $\mathbb{O}$ is the set of traces. In this theory refinement is modelled by implication;

Three the well-known Galois connections can be used to define a new interpretation of entities, contexts and observations in terms of existing ones, consequently giving a new interpretation to both refinement and what refinement guarantees.

This general model can be made more concrete by instantiating its parameters Ξ and O to give what we call a *special theory*. It has been shown ([11]) that some of the classic theories of both abstract data types (ADT) and processes that appear in the literature are special theories of the general model given here.

In Section 3.1 we develop a general model of flexible refinement. In Section 4.1 we apply it to individual operations, *e.g.* $\mathsf{Put_A}$ (Fig 1). Subsequently in Section 5 we apply it to whole entities, *e.g.* $\mathsf{Set_A}$ (Fig 1). In Section 6 we apply the refinement we have developed to give a formal development of $\mathsf{Set_C}$ from $\mathsf{Set_A}$ as needed in our motivating example.

3.1 General model of flexible refinement

In this section we give a general definition of a standard natural notion of refinement. We use three distinct systems: E, the entity being refined; X, the context which interacts privately with E; and U, a user that observes X. All interaction occurs at the interface between two systems. Our user U takes on the role of a tester, so it passively observes any event in the interface between X and U. This is pictured in Fig 2.

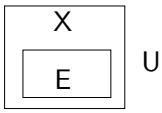

Fig. 2. An entity, a context and a user

In order to formalise this notion we must decide what the user can observe, so we make some assumptions. In practice we are interested in reasoning about and refining small modules of a larger entity. Thus we model the entity (module) E as existing in some context X (rest of larger whole) interacting on the set of events *Act* where $Act \subseteq Names$ (where *Names* is a set consisting of all possible event names). All E's events interact with X at the E—X interface (see Fig 2). So, the events in the set $Names \setminus Act$ are those which cannot appear in E and which, therefore, X and U communicate with, without interfering with communication between E and X. We model the observer as a passive user U that is a third entity that observes or interacts with X, but cannot block the X events. The user U is formalised by O, an observation function that returns sets of traces of observed events.

We use the notation $[\_]_{\mathsf{X}}$ to denote putting-in-a-context since this is suggestive of a gap into which an entity can be placed in order to interact with some context X, so placing entity E in context X will be written as $[\mathsf{E}]_{\mathsf{X}}$, which is another (composite) entity.

Definition 3.1 General refinement. Let Ξ be a set of contexts each of which the entities A and C can communicate privately with, and O be a function which returns a set of traces, each trace being what a user observes of an execution. Then:

$$\mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{C} \triangleq \forall\, x \in \Xi . O([\mathsf{C}]_x) \subseteq O([\mathsf{A}]_x)$$

This general definition of refinement is one of the central parts of this paper and later it will be specialised (made more concrete) by:

one defining that we represent our entities as partial relations;

two defining the sets of contexts Ξ; and

three defining the observation function O from entities to sets of traces.

We also define equality between representations:

Definition 3.2 Entity equality

$$\mathsf{A} =_{\Xi,O} \mathsf{B} \triangleq \mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{B} \wedge \mathsf{B} \sqsubseteq_{\Xi,O} \mathsf{A}$$

3.2 Theories and relations

It is easy to see that we can give entities in our general model a relational semantics. We are not the first to use relations as a semantics for a diverse range of models: indeed the Unifying Theories of Programming (UTP, [12]) do just this.

Definition 3.3 The relational semantics of an entity A is a subset of $\Xi \times \mathbb{O}$:

$$[\![\mathsf{A}]\!]_{\Xi,O} \triangleq \{(x, o) \mid x \in \Xi, o \in O([\mathsf{A}]_x)\}$$

It should be noted that we use quite different relations to those in UTP, but like UTP we have "refinement as subset of the relations":

$$\mathsf{A} \sqsubseteq_{\Xi,O} \mathsf{C} \Leftrightarrow [\![\mathsf{C}]\!]_{\Xi,O} \subseteq [\![\mathsf{A}]\!]_{\Xi,O}.$$

This means that refinement is implication between the predicates that define the relations. Thus we can view each set of relations as defining a logical theory where refinement is implication.

It is important to note that we have not fixed what the underlying operational or denotations semantics of the entities are. Indeed, the entities in a theory can have an operational semantics of ADTs, processes of various kinds and even individual operations, and different refinement relations (differentiated by the contexts and observations we choose) can give different denotations (meanings) to the same operational semantics.

Definition 3.4 A theory T is $(\mathsf{E}_\mathsf{T}, \sqsubseteq_\mathsf{T})$ where E_T is a set of entities and $\sqsubseteq_\mathsf{T} \subseteq \mathsf{E}_\mathsf{T} \times \mathsf{E}_\mathsf{T}$ is a refinement relation.

By considering only theories where refinement is given by Definition 3.1, *i.e.* $\sqsubseteq_\mathsf{T} \triangleq \sqsubseteq_{\Xi_\mathsf{T}, O_\mathsf{T}}$, our theories can equally well be defined by the tuple

$$(\mathsf{E}_\mathsf{T}, \Xi_\mathsf{T}, O_\mathsf{T})$$

In the next section we generalise our general model further by viewing a theory as a *layer* in the larger scheme of things.

3.3 Theory morphisms

We use a semantic mapping $[\![\_]\!]_v$ to interpret high-level entities as low-level entities, and a separate semantic mapping vA to interpret low-level entities as high-level entities.

We view this pair of functions as a *theory morphism*. To be useful they must reinterpret both high-level refinement as low-level refinement and low-level refinement as high-level refinement. We do this by constructing a *Galois connection* between two theories.

In top-down development, such a theory morphism may be preceded by some high-level refinement steps and may itself precede low-level refinement steps (see Fig 3). The theory morphism replaces a high-level entity by a low-level entity,

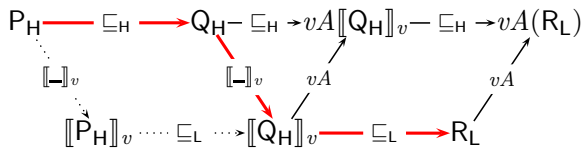

Fig. 3. Refinement $\longrightarrow$ within and between theories

a high-level context by a low-level context and high-level observation by low-level observation.

We will define the effect of a theory morphism on $\sqsubseteq_\mathsf{H} \triangleq \sqsubseteq_{\Xi_\mathsf{H}, O_\mathsf{H}}$ (high-level refinement) in terms of the theory morphism applied to both the contexts and observations:

$$[\![\sqsubseteq_\mathsf{H}]\!]_v \triangleq \sqsubseteq_{[\![\Xi_\mathsf{H}]\!]_v, [\![O_\mathsf{H}]\!]_v}$$

We have chosen to let the semantic mappings $[\![\_]\!]_v$ and vA be polymorphic in that we apply them to entities, contexts, observations and refinement relations. In practice we will need only to define one function that will be amended in an obvious way to give the various functions of the required type.

We find it useful to think of theories as being layers in a structured development of a system, as Fig 3 suggests. Thinking of the various layers of abstraction we move through in a protocol stack might be a useful analogy here.

Definition 3.5 Semantic mappings $[\![\_]\!]_v^{\mathsf{HL}}$ and vA^{HL} between $(\mathsf{E_H}, \Xi_\mathsf{H}, O_\mathsf{H})$, a high-level theory, and $(\mathsf{E_L}, \Xi_\mathsf{L}, O_\mathsf{L})$, a low-level theory, form a theory morphism when they are a Galois connection:

$$\forall \mathsf{X_H} \in \mathsf{E_H}, \mathsf{Y_L} \in \mathsf{E_L}.[\![\mathsf{X_H}]\!]_v^{\mathsf{HL}} \sqsubseteq_{\Xi_\mathsf{L}, O_\mathsf{L}} \mathsf{Y_L} \Leftrightarrow \mathsf{X_H} \sqsubseteq_{\Xi_\mathsf{H}, O_\mathsf{H}} vA^{\mathsf{HL}}(\mathsf{Y_L})$$

We choose to call a theory morphism a *vertical refinement* $\sqsubseteq_v = ([\![\_]\!]_v, vA)$ because it defines a guaranteed relation between the more abstract high-level entities and the more concrete low-level entities.

The two functions $[\![\_]\!]_v^{\mathsf{HL}}$ and vA^{HL} define how to *interpret* one theory in the other and consequently:

$\mathsf{H} \sqsubseteq_v^{\mathsf{HL}} \mathsf{L}$ **guarantees** that the high-level vA-interpretation of entity L behaves like (can be observed to have a subset of the observations of) entity H (*e.g.* $\mathsf{P_H}$ in Fig 3) whenever it is placed in any high-level context Ξ_H and only the high-level observations O_H are made.

We now have general refinement, Definition 3.1 and vertical refinement Definition 3.5 taken together they constitute what we call *flexible refinement*.

3.4 Subsets are simple theory morphisms

In this section we are interested in the special case of theories A and C where $\Xi_\mathsf{A} \subseteq \Xi_\mathsf{C}$ and $\mathbb{O}_\mathsf{A} \subseteq \mathbb{O}_\mathsf{C}$.

It is well-known ([13, p155] [12, 4.1]) that subset relations like $\Xi_\mathsf{A} \times \mathbb{O}_\mathsf{A} \subseteq \Xi_\mathsf{C} \times \mathbb{O}_\mathsf{C}$ form a simple theory morphism which we denote by $\sqsubseteq_{sub}^{\mathsf{AC}}$, where the interpretation mappings are:

embedding of the abstract in the more complex concrete, where for any $\mathsf{P_A} \in \mathsf{E_A}$ (using the definitions $\Xi_{\mathsf{C}\backslash\mathsf{A}} \triangleq \Xi_{\mathsf{C}} \backslash \Xi_{\mathsf{A}}$ and $\mathbb{O}_{\mathsf{C}\backslash\mathsf{A}} \triangleq \mathbb{O}_{\mathsf{C}} \backslash \mathbb{O}_{\mathsf{A}}$) :

$$[\![\mathsf{P_A}]\!]^{\mathsf{AC}}_{sub} \triangleq [\![\mathsf{P_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \cup \{(x, o) \mid x \in \Xi_{\mathsf{C}\backslash\mathsf{A}} \vee o \in \mathbb{O}_{\mathsf{C}\backslash\mathsf{A}}\};$$

projection of the concrete back into the abstract, where for any $\mathsf{P_C} \in \mathsf{E_C}$:

$$subA^{\mathsf{AC}}(\mathsf{P_C}) \triangleq [\![\mathsf{P_C}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}}.$$

We can establish that $\sqsubseteq^{\mathsf{AC}}_{sub}$ is a theory morphism, *i.e.* that:

$$\forall \mathsf{X_A} \in \mathsf{E_A}, \mathsf{Y_C} \in \mathsf{E_C}.[\![\mathsf{X_A}]\!]^{\mathsf{AC}}_{sub} \sqsubseteq_{\mathsf{C}} \mathsf{Y_C} \Leftrightarrow \mathsf{X_A} \sqsubseteq_{\mathsf{A}} subA^{\mathsf{AC}}(\mathsf{Y_C})$$

by checking that:

$$\forall \mathsf{X_A} \in \mathsf{E_A}, \mathsf{Y_C} \in \mathsf{E_C}.[\![\mathsf{X_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \cup \{(x, o) \mid x \in \Xi_{\mathsf{C}\backslash\mathsf{A}} \vee o \in \mathbb{O}_{\mathsf{C}\backslash\mathsf{A}}\} \supseteq [\![\mathsf{Y_C}]\!]_{\Xi_{\mathsf{C}}, O_{\mathsf{C}}}$$

$$\Leftrightarrow [\![\mathsf{X_A}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}} \supseteq [\![\mathsf{Y_C}]\!]_{\Xi_{\mathsf{A}}, O_{\mathsf{A}}}$$

From the guarantee in Section 3.3 we see that:

$\mathsf{A} \sqsubseteq^{\mathsf{AC}}_{sub} \mathsf{C}$ **guarantees** the high-level vA-interpretation of any entity from C behaves exactly like an entity from A whenever it is placed in any abstract context Ξ_{A} and only the abstract observations O_{A} are made.

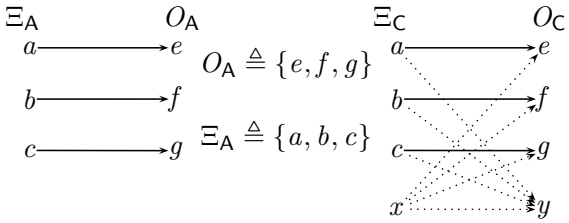

Fig. 4. $\mathsf{A} \sqsubseteq^{\mathsf{AC}}_{sub} \mathsf{C}$

Recall that our theory can be applied both to single operations and to machines. Thinking of Fig 4 as representing a single operation then we observe that this is neither how refinement is normally defined in the literature nor, if we consider x and y to be $\perp$, is it the same as any definition of lifting and totalising that we can find in the literature.

Our subset theory morphism formalise the addition of new observations, the y in Fig 4. The intuitive justification for adding $\{(a, y), (b, y), (c, y), (x, y)\}$ is that in the abstract specification $\Xi_A \times \mathbb{O}_A$ the y observation had not been considered (recorded). Although this definition of vertical refinement may seem unusual when considering the entity to be a single operation, it is no more than an application of Galois connections as have appeared widely in the literature. It is the adding of the new observations that makes our formal model (of both single operations and machines) so flexible. In addition it is the preservation of the guarantee that allows us to view theory morphisms as refinements.

4 Flexible refinement with state-based interfaces

Theory morphisms and general refinement, which together constitute what we call flexible refinement, have been given without being specific as to what means of *representation* is to be used nor what will appear in the interface between entity and context. Machines consist of both state and operations (sometimes called events or actions). Here we focus on the situation where the interface is considered to consist of state and not operations.

4.1 State-based interfaces and operations

In this section we will specialise the general definitions to the concrete case of single operations which transform some state from the state space *State* by:

one representing entities by partial relations from $\mathbb{E} \triangleq State \times State$;

two using initial states from $State_{Pre} \subseteq State$ as the contexts $\Xi_S \triangleq State_{Pre}$; and

three making an observation a pre-, post- pair of states from $\mathbb{O}_S \triangleq State_{Pre} \times State_{Post}$ where $State_{Post} \subseteq State$.

The relational semantics in Definition 3.3 is between the contexts Ξ_S in which an entity finds itself, which for an operation is some "starting" state from $State_{Pre}$, and $\mathbb{O}_S$, which tells us what can be observed when the entity is executed. So, a context is a state in the precondition of the operation, *i.e.* those states for which the operation is properly defined. Also, $\mathbb{O}_S$ gives us the states in which the operation can terminate given its stating state, so $\mathbb{O}_S$ can be given by pre- and post-state pairs from $State_{Pre} \times State_{Post}$. Consequently we are concerned with relations in $\Xi_S \times \mathbb{O}_S \subseteq State_{Pre} \times (State_{Pre} \times State_{Post})$.

Notice that the set of start states of the observation already appears as the domain of the whole relation, so the repeat is redundant and hence we can omit it. Thus the semantics need be no more than a subset of $State_{Pre} \times State_{Post}$, which accords with the usual way of giving the semantics of a single operation.

As an example, we can apply the subset morphism to introduce the error value X to the operation $\mathsf{Put_A}$ (see Fig 5, where $State_{Pre} = State_{\mathsf{A}} \times \mathbb{N}$ and $State_{Post} = State_{\mathsf{A}}$) for both the Z and the underlying relational semantics.

We can consider the contexts and observations to delineate a "frame of reference" for an operation such that no guarantee about it is given outside of this frame of reference. In this case it means that $\mathsf{Put_X}$ (which, note, uses the augmented state from $\mathsf{Set_C}$ in Fig 1) is free to have any behaviour in contexts $\{(X, n) \mid n \in \mathbb{N}\}$ and, in addition, the post-state X, since it is not in any observation, can be reached from any context. Clearly $\mathsf{Put_X}$ and $\mathsf{Put_C}$ are not the same but as we will explain in Section 6 we will make use of $\mathsf{Put_X}$ in the stepwise development of $\mathsf{Set_C}$.

Notice that in this refinement X needs to be added to the observations and the whole set of pairs $\{(X, n) \mid n \in \mathbb{N}\}$ needs to be added to the contexts. We write $\sqsubseteq_{sub}^{\{X\}}$ as shorthand for this.

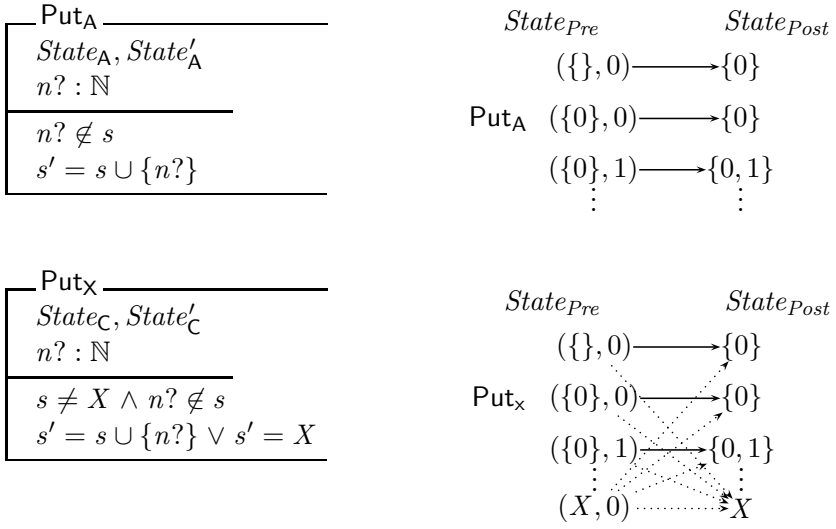

Fig. 5. $\mathsf{Put_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Put_X}$

The refinement $\mathsf{Put_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Put_X}$ guarantees that any old observation, $State_{\mathsf{A}}$, that can be made of $\mathsf{Put_X}$ could have been made of $\mathsf{Put_A}$.

4.2 State-based interfaces and machines

In Section 4.1 we defined a theory morphism $\sqsubseteq_{sub}^{\{X\}}$ and the example $\mathsf{Put_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Put_X}$. It should be easy to see that we can lift this to machines by extending the abstract machine state and hence the state of all the machines operations. Hence we write $\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X}$ (see Fig 6), for an application of the state-subset theory morphism to all the operations in the abstract machine $\mathsf{Set_A}$.

But what guarantee does this refinement provide? Put abstractly and generally:

> it is the responsibility of the context, *i.e.* program, to call only operations that keep the state of the entity in one of the abstract states

which for our example means any operation cannot finish in X.

Put slightly differently:

> the specification makes no guarantee about the behaviour of concrete operations outside of $\Xi_{\mathsf{A}} \times \mathbb{O}_{\mathsf{A}}$ but inside $\Xi_{\mathsf{A}} \times \mathbb{O}_{\mathsf{A}}$ the concrete relational semantics is a subset of the abstract relational semantics.

It should be clear that state-based flexible refinement can only introduce nondeterminism by allowing the operations of the abstract machine to terminate in *new states*, *i.e.*states not in the abstract machine. The intuitive reason for this being that the new states, X in our example, are thought not to have been considered in the abstract specification.

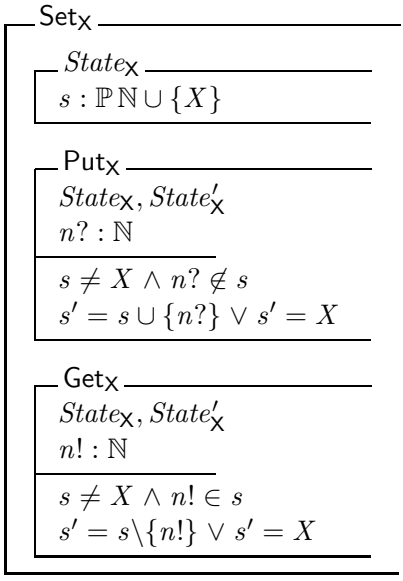

Fig. 6. $\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X}$

5 Lax refinement with operation-based interfaces

By a "machine" all we mean is a set of operations or named partial relations over a state. Examples of machines include abstract data types, processes, collections of states and operations in Z like $\mathsf{Set_A}$, and B machines.

So, a machine M from the set of all machines $\mathbb{M}$ is defined to be a tuple consisting of its state space $State_M$ and a function from names (the alphabet of the machine Alp_M) to relations over $State_M$:

$$M \triangleq (State_M, init_M, Alp_M \to State_M \times State_M)$$

In this section we will make the abstract definitions more concrete by defining:

(i) entities as machines $\mathbb{E} \triangleq \mathbb{M}$;

(ii) contexts as programs, *i.e.* unbranching sequences of operations, $\Xi_M \triangleq Alp_M^*$;

(iii) observations to be pre-, post-state pairs, $\mathbb{O}_M \triangleq State_M \times State_M$;

(iv) the observation function O_M returns the sequential composition of the relational semantics of the individual operations in the program.

Hence the names of the operations are in the entity context interface and the state is in the context user interface. The relational semantics of a machine M is:

$$\Xi_M \times \mathbb{O}_M = Alp_M^* \times (State_M \times State_M)$$

It has been shown [1,11] that this special theory is a standard theory of data refinement. To this we have added vertical refinement and as we have previously said subset morphisms are a simple type of vertical refinement.

Let us assume that the concrete machine is the same as the abstract except for the existence of new operations *i.e.* new named relations. Firstly by definition the abstract machine can not be placed in any program that uses one of these new operations. Secondly if the concrete machine is placed in a context that only calls operations with the same name as operations in the abstract machine then what can be observed of the concrete machine is exactly what can be observed of the abstract machine. From the previous two statements it can easily be seen that the abstract and concrete machines are related by a subset morphism.

6 Example

Here we re-work the example given in [4]. Starting with $\mathsf{Set_A}$, the first step is to define a subset theory morphism to introduce the error state X as in Section 4.2. This gives us the first step, $\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X}$ in the stepwise refinement of $\mathsf{Set_A}$ into $\mathsf{Set_C}$.

Having introduced the new state X we will have introduced nondeterminism, as an operation started from any initial state may now end at the new post state X (see Fig 5). This nondeterminism can be removed by "ordinary" refinement (Definition 3.1). Thus $\mathsf{Set_X} \sqsubseteq \mathsf{Set_B}$ in Fig 7 is the second refinement step towards $\mathsf{Set_C}$.

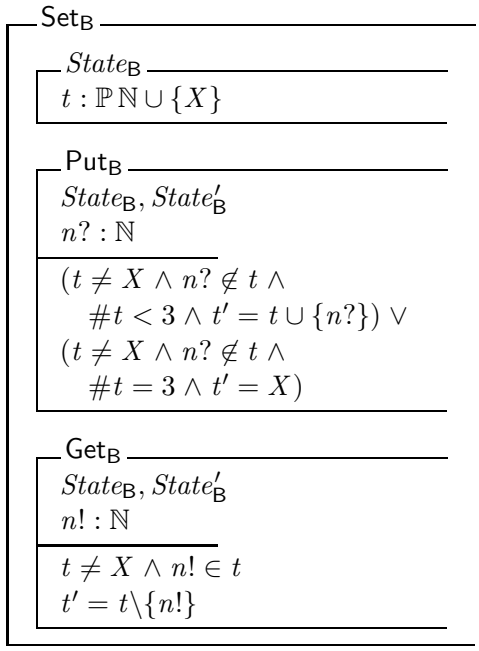

Fig. 7. $\mathsf{Set_X} \sqsubseteq \mathsf{Set_B}$

The third and final step that introduces the new operation Reset is another subset theory morphism but this time as $\mathsf{Set_B} \sqsubseteq_{sub}^{\{\mathsf{Reset}\}} \mathsf{Set_C}$ it is applied to the

whole machine, as in Section 5.

We can now be sure, from the three refinement steps, what is guaranteed by the flexible refinement of $\mathsf{Set_A}$ into $\mathsf{Set_C}$

$$\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X} \sqsubseteq \mathsf{Set_B} \sqsubseteq_{sub}^{\{\mathsf{Reset}\}} \mathsf{Set_C}$$

The first refinement $\mathsf{Set_A} \sqsubseteq_{sub}^{\{X\}} \mathsf{Set_X}$ tells us that $\mathsf{Set_X}$ behaves like $\mathsf{Set_A}$ when it is not in the error state X and when operations are called only when they do not end in error state X. The second refinement step is a simple reduction of nondeterminism. It is in this step that the developer decides that the set is to have no more than three elements. Hence the guarantee is unchanged when sets never have more than three elements but any operation that attempts to increase the size to greater than three is free to return the new state X. The third and final refinement $\mathsf{Set_B} \sqsubseteq_{sub}^{\{\mathsf{Reset}\}} \mathsf{Set_C}$ further restricts the contexts to programs that call only the "Put" and "Get" operations.

Together these guarantees form exactly the guarantee we wanted, as given in Section 2.1, and they have been captured formally via flexible refinement.

7 Conclusion

We propose developing tools in two parts: the front-end that gives languages of the various types known in the literature an operational semantics in a formalism that takes the role of a *common intermediate language*; and the back-end that manipulates the operational semantics.

Here we are only interested in the back-end and have presented a formal model that allows great flexibility in what development steps can be formalised as a refinement.

We have considered what the nature of a guarantee in the context of specifications and their implementations might be.

We have also considered the usual informal notion of refinement. By seeking to preserve as much of that informal notion as possible, we have been led to a new and general definition of refinement. We have shown how to apply this definition to both expand the state space of an entity and its set of operations.

The usefulness of these ideas is illustrated by providing a formal refinement of an example that has been used in the literature to show that more traditional and less flexible definitions of refinement are too restrictive, leading to suggestions for weaker notions than refinement which may not preserve any guarantees about how the concrete version of a system is related to a more abstract version.

We note that the latest addition to Event B [14] does allow addition of operations as a refinement step as long as the the invariant is preserved. So as there is no invariant on the state in our example Event B could make the third refinement step in our example $\mathsf{Set_B} \sqsubseteq_{sub}^{\{\mathsf{Reset}\}} \mathsf{Set_C}$. However, none of the refinements in the literature allow the addition of states or the addition of operations where an invariant changes.

Future work: Combining our framework, as presented here, with the work

reported in [15,16] could yield a proof assistant to support use of our very general notion of flexible refinement.

References

[1] Reeves, S., Streader, D.: Comparison of data and process refinement. In Woodcock, J., Dong, J., eds.: Proceedings of ICFEM 2003. Number 2885 in Lecture Notes in Computer Science. Springer-Verlag (2003) 266–285

[2] Reeves, S., Streader, D.: Flexible refinement. Working paper series No. 02/2007, University of Waikato (2007) http://researchcommons.waikato.ac.nz/cms_papers/35/.

[3] Reeves, S., Streader, D.: Feature refinement. In: SEFM 2007. to appear. IEEE (2007)

[4] Banach, R., Derrick, J.: Filtering retrenchments into refinements. In: Proceedings of SEFM'06, IEEE (2006) 60–69

[5] Bolton, C., Davies, J.: A singleton failures semantics for communicating sequential processes. Formal Aspects of Computing **18** (2006) 181–210

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

[7] Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice Hall (1996)

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

[9] Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Formal Approaches to Computing and Information Technology. Springer (2001)

[10] Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)

[11] Reeves, S., Streader, D.: State- and Event-based refinement. Technical report, University of Waikato (2006) Computer Science Working Paper Series 09/2006, ISSN 1170-487X, http://researchcommons.waikato.ac.nz/cms_papers/12/.

[12] Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)

[13] Taylor, P.: Practical Foundations of Mathematics. Cambridge University Press (1999) Cambridge studies in advanced mathematics 59.

[14] Abrial, J.R., Cansell, D., Méry, D.: Refinement and reachability in Event B. In Treharne, H., King, S., Henson, M.C., Schneider, S., eds.: ZB05: Formal Specification and Development in Z and B. Volume 3455 of Lecture Notes in Computer Science., Springer (2005) 222–241

[15] Bortin, M., Johnsen, E.B., Luth, C.: Structured formal development in Isabelle. Nordic J. of Computing **13** (2006) 2–21

[16] Isobe, Y., Roggenbach, M.: A generic theorem prover of CSP refinement. In Halbwachs, N., Zuck, L., eds.: TACAS. Number 3440 in Lecture Notes in Computer Science. Springer-Verlag (2005)