Perspicuity and Granularity in Refinement

Eerke Boiten
School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E.A.Boiten@kent.ac.uk

This paper reconsiders refinements which introduce actions on the concrete level which were not present at the abstract level. It draws a distinction between concrete actions which are "perspicuous" at the abstract level, and changes of granularity of actions between different levels of abstraction.

The main contribution of this paper is in exploring the relation between these different methods of "action refinement", and the basic refinement relation that is used. In particular, it shows how the "refining skip" method is incompatible with failures-based refinement relations, and consequently some decisions in designing Event-B refinement are entangled.

Keywords: Refinement, action refinement, stuttering steps, ASM, Event-B, Z, internal operations, weak refinement, granularity, perspicuity, divergence.

1 Introduction

This paper discusses how different ways of introducing "extra" actions in refinement (such as weak refinement, action refinement, stuttering steps) relate to the underlying refinement relations used (e.g. trace refinement, failures refinement). In particular, we aim to show how the choices in those two dimensions are interdependent. The paper is not intended to be polemic ("my formalism/refinement relation is better than yours") nor is it really meant to be a first introduction to the topic. Where it appears to state the obvious, this is in an attempt to ensure that commonalities, differences, and design decisions in refinement relations are exhibited in an unambiguous and uncontroversial way.

Before describing the issues in detail, we consider an example. The example is presented in Z, but the notation used is not essential to what follows in this paper. In general, most of what is described in this paper could be expressed in ASM [18], (Event-)B [1], Z [19], binary relations [11], UTP [15] or many other state-based formalisms; for the moment we make no assumptions about what refinement relation is "in force".

This example is due to Carroll Morgan, who presented it during an enlightening conversation at the 2009 Dagstuhl seminar "Refinement Based Methods for the Construction of Dependable Systems". The abstract specification is essentially a priority queue, stored as a bag, so taking out an element involves selecting the minimum of the bag. Obvious specifications of functions *min* on bags and (later) *sorted* on sequences are omitted. The schema *AS* describes system states, *AInit* initial states, and the schemas *Ain* and *Aout* the operations of adding and removing an element. The precondition $b \neq \llbracket\,\rrbracket$ is included explicitly in *Aout*, in recognition of it having to be an explicit guard in alternative notations such as

To appear in EPTCS.

Event-B.

$$\textit{AS} \;\; b : \operatorname{bag} \mathbb{N}$$

$$\textit{AInit} \;\; \textit{AS}' \;|\; b' = [\![\,]\!]$$

$$\textit{Ain} \;\; \Delta \textit{AS};\; x? : \mathbb{N} \;|\; b' = b \uplus [\![x?]\!]$$

$$\textit{Aout} \;\; \Delta \textit{AS};\; x! : \mathbb{N} \;|\; b \neq [\![\,]\!];\; b = b' \uplus [\![x!]\!];\; x! = \textit{min}(b)$$

The concrete specification uses a sequence to represent the queue. Removing an element is only possible when the sequence is non-empty and sorted, in which case the element to be removed is at the head of the sequence. The schema *Sort* describes the sorting of the sequence. The schema *Cycle* is mostly a red herring[1] and not part of Morgan's original example.

$$\textit{CS} \;\; s : \operatorname{seq} \mathbb{N}$$

$$\textit{CInit} \;\; \textit{CS}' \;|\; s' = \langle\,\rangle$$

$$\textit{Cin} \;\; \Delta \textit{CS};\; x? : \mathbb{N} \;|\; s' = s \frown \langle x? \rangle$$

$$\textit{Cout} \;\; \Delta \textit{CS};\; x! : \mathbb{N} \;|\; s \neq \langle\,\rangle;\; \textit{sorted}(s);\; s = \langle x! \rangle \frown s'$$

$$\textit{Sort} \;\; \Delta \textit{CS} \;|\; \textit{items}\, s = \textit{items}\, s';\; \textit{sorted}(s')$$

$$\textit{Cycle} \;\; \Delta \textit{CS} \;|\; s = \langle\,\rangle \land s' = \langle\,\rangle \lor s' = (\textit{tail}\, s) \frown \langle \textit{head}\, s \rangle$$

This paper discusses the many ways in which one may consider the concrete specification to refine the abstract one, possibly after a slight modification, or possibly not at all, depending on the notions of refinement and action refinement employed. Before we move on to that level of complication, consider the composed schema $\textit{SortOut} == \textit{Sort} \fatsemi \textit{Cout}$, whose meaning is given by

$$\textit{SortOut} \;\; \Delta \textit{CS};\; x! : \mathbb{N} \;|\; s \neq \langle\,\rangle;\; \exists s'' : \operatorname{seq} \mathbb{N} \bullet \textit{items}\, s = \textit{items}\, s'' \land \textit{sorted}(s'') \land s'' = \langle x! \rangle \frown s'$$

[1] One might use it to represent the non-determinism in a distributed implementation where individual clients have no control over the access pointer in a cyclical list, ... maybe.

Then, uncontroversially, in most sensible refinement relations, the operation *Aout* is refined by *SortOut* (or more precisely: the data type $(AS, AInit, Ain, Aout)$ is refined by $(CS, CInit, Cin, SortOut)$) under the retrieve relation $b = items\, s$. In fact, this is normally an equivalence: refinement also holds in the reverse direction[2].

The rest of this paper is structured as follows. In Section 2 we describe different basic refinement notions. Then in Section 3 we discuss the various methods in which "extra" operations may appear in refinement steps. In Section 4 we compare how these methods can be used to model the decomposition of actions into smaller grained ones, and how this impacts on the various basic refinement notions. Finally, Section 5 presents some conclusions.

2 Basic Notions of Refinement

We have given detailed fully formal descriptions and comparisons of the different basic notions of refinement for state-based and concurrent systems in many previous papers, e.g. [6, 11, 5]. Rather than repeating this and thereby fixing a formalism or even introducing a new one, we remain informal here, using various formalisms and their refinement notions as illustrations.

In basic data refinement, systems (or machines or abstract data types) are compared which have identical alphabets (or sets of labels of operations (or actions or events)). Apart from conditions on initial and possibly final states, and other details which depend on what observations can be made of these systems, operations are compared in pairs of an abstract and a concrete operation, with refinement conditions being some subset of the following properties:

(1) Consistency The effect of the concrete operation is one that is allowed by the abstract operation.

(2) Enabledness When operations can be invoked in the abstract state, they can be invoked in the concrete state as well.

(3) Restricted consistency In states where the abstract operation is enabled, the effect of the concrete operation is one that is allowed by the abstract operation.

Property (1) or its weaker variant (3) represents the essence of refinement: that a client would be unable to observe conclusively that they are using the concrete rather than the abstract system. Property (2) ensures that the client is indeed able to perform the same "experiments" on both systems. Property (1) obviously implies (3), and also a converse of (2): where concrete operations are enabled (leading to an "effect"), their abstract counterparts should be enabled, too (in order to allow comparison of effects). The properties leave out detail about what an effect is, are purposefully vague on "can be invoked" in (2) to allow a variety of interpretations, and leave any linking between abstract and concrete states implicit. They are also somewhat biased towards downward simulation. A few examples should make all this clearer. The refinement relations described below will be refered to in later sections.

Traditional (downward simulation) *Z refinement* [19, 11] is characterised by properties (2) and (3), with "can be invoked" in a state computed as individual operations' preconditions, i.e. whether their defining predicates can be satisfied for some after-state. Condition (2) is called "applicability" and typically formulated as

$$\mathrm{pre}\, AOp \land R \Rightarrow \mathrm{pre}\, COp$$

[2] A refinement linking *Ain* to $Cin \fatsemi Sort$ instead is equally possible but would require strengthening the concrete state invariant to sorted sequences; $Cin \fatsemi Sort$ then simplifies to the insert operation of insertion sort.

where $\text{pre}\,AOp == \exists AS' \bullet AOp$ denotes the computed precondition. Condition (3) is called "correctness", and typically formulated as

$$\text{pre}\,AOp \land R \land COp \Rightarrow \exists AS' \bullet R' \land AOp$$

We have sometimes called this refinement relation the "contract" model of refinement as it constrains the implementation only within the original precondition.

Trace refinement is characterised by (1) only, only requiring that anything that *does* happen in the concrete specification is consistent with the abstract one. As such, it represents preservation of safety properties only, "nothing bad happens". No concrete operations being enabled at all, for example, is an acceptable trace refinement.

Basic *Event-B refinement* (called simple refinement in [1, Ch. 14]) is characterised by (1), with (optionally) a weak alternative to (2): if the concrete state deadlocks (i.e. no events are enabled), then so should the abstract state. Enabledness of events is given by explicitly specified guards, with a "feasibility" proof obligation ensuring that they are at least as strong as any computed precondition. Abrial [1, p. 429] states that condition (2) could be imposed, but "this happens to be sometimes too strong". (We will return to this.)

Failures-based variants of refinement are characterised by (1) and (2), where (2) considers individual operations for "blocking Z refinement" and singleton failures refinement, or sets of concurrently enabled operations for failures refinement as in CSP. We refer to [6, 17, 5] for detailed discussion of these refinement relations and the finer distinctions between them, which are not relevant in the current paper.

Note that a refinement relation characterised by property (3) without property (2) is nonsensical as it is not transitive: preconditions or guards can be strengthened (lack of (2)) and then weakened (by (3)), but the composition of such steps does not respect (3).

3 Adding Operations in Refinement

The basic refinement rules described above deal only with the situation where the abstract and concrete specifications have the same alphabet of operations. There are many ways in which one could allow a refined specification to have "extra" operations – we discuss a number of them. First, we mention alphabet extension and alphabet translation [11, Ch. 14] for completeness. Then, we get to the core of this paper: stuttering steps, the introduction of internal operations, and action refinement, and how these sometimes get conflated.

3.1 Alphabet Extension and Translation

The simplest way of allowing new operations in refinement is *alphabet extension*: to just accept them without any further constraints. If we make the intuitive step of identifying a non-existent operation with one that is never enabled, alphabet extension should be perfectly acceptable in traditional Z refinement: it means we allow implementors to provide functionality that we had not asked for. In a process algebra context alphabet extension is typically not allowed, and indeed that would make sense in our intuitive view: it would go against refinement property (1), by having no matching abstract behaviour for some concrete behaviour.

In *alphabet translation*, a single abstract operation is implemented by multiple concrete ones, which requires an explicit mapping, recording for every concrete operation which abstract operation it represents, and thus which operation's behaviour it needs to correspond with. (If this mapping is not required to be total, alphabet extension is subsumed.) A typical example for this would be an abstract two-dimensional grid specification with a "move" operation, which is refined into "moveNorth", "moveEast", etc. Alphabet translation is allowed in Event-B, where it is called "splitting" an abstract event.

The semantic property established in alphabet translation is: every concrete trace (with its corresponding observations) is consistent with an abstract trace that relates to it by the given mapping (applied elementwise) with its corresponding observations.

3.2 Perspicuous Operations

State-based systems potentially change state when operations are executed. When no operation is invoked, the state does not normally change. Some formalisms take this into account by including explicitly so-called stuttering steps in their semantics: steps where the state does not change between two observations, due to no event having taken place. In the light of that, it is intuitively obvious to accept the introduction of additional concrete events as refinements of the identity operation (a.k.a. *skip*) on the abstract state. We will call these *perspicuous* concrete events, to be distinguished from "internal events" (see below) which incur additional assumptions and requirements. In particular, in subsequent refinement steps, perspicuous operations do *not* have a different status from operations that were present earlier.

Abrial [1] presents a similar motivation for the introducion of new events in Event-B, analogous to how this is done in action systems [3], and refers to it as "observing our discrete system in the refinement with a finer grain than in the abstraction". Event-B is explicit about the introduction of such events as being refinements of *modelling*: introducing not just aspects of a solution, but more detail of the model. Indeed, where refinement is viewed as only moving from a complete description of a problem to its solution, the introduction of perspicuous operations which achieve nothing in the abstract world can hardly be useful by itself[3]. Both action systems and Event-B include a relative deadlock freedom condition with this kind of refinement: the new system should deadlock (i.e., terminate, in the action systems view) no more often than the old one. The semantic relation established by this kind of generalised refinement is: for every concrete trace with its corresponding observations, an abstract trace constructed by crossing out all perspicuous actions is consistent with it.

In the running example, under most refinement relations and with the obvious retrieve relation $items\, s = b$ both concrete operations *Sort* and *Cycle* are candidate perspicuous operations, as they satisfy $items\, s = items\, s'$ and thus relate identical abstract states. They are both applicable in every concrete state and thus are refinements of an abstract *skip* even when property (2) is imposed.

For perspicuous operations, the notion of *divergence* comes into the picture. A collection of perspicuous operations is divergent if infinitely often in succession, from some state, one of its members can be invoked. In a trace-based view, where perspicuous operations could be inserted at arbitrary points between "normal" operations, non-divergence is necessary to ensure that a finite trace cannot get extended into an infinite one by that process. This is how Abrial [1] explains it[4]. With additional assumptions, such as that a system might perform perspicuous operations independently, divergence becomes a practical as well as a theoretical problem. Butler [9] explains the non-divergence requirement in Event-B by saying "The new events introduced in a refinement step can be viewed as hidden events not visible to the environment of a system and are thus outside the control of the environment" which would suggest

[3] This is *not* intended to be a controversial statement or implicit criticism on Event-B: the crux is in the phrase *by itself*, and this should become clearer later when we compare the different ways of encoding action refinement.

[4] His use of the term "reachable" is a bit unfortunate, though – this tends to be an existential property (some path is finite) rather than the required universal (all paths are finite) property required.

these are not just perspicuous events, but even *internal* events as we will discuss next. In action systems [3], which are viewed as a main inspiration for Event-B, all actions could be considered to be internal (even if the variables they modify are not), which conforms more with Abrial's explanation than with Butler's[5]. A typical method of proving non-divergence is by establishing a variant (well-founded, strictly decreasing function) on newly introduced (collections of) perspicuous operations [8, 12, 1]. If refinement is based on property (1) rather than property (3), i.e., an action cannot gain behaviour in refinement, then non-divergence is preserved by subsequent refinement steps.

In the example, both perspicuous operations are divergent. This is obvious from the fact that they are enabled in *every* concrete state. *Sort* allows an infinite sequence of invocations of which only the first does not necessarily correspond to a concrete *skip*. For formalisms that use infinite traces and allow stuttering steps, such as TLA, this may not be a problem. Removing divergence on each of the operations can be done using several possible small modifications. The divergence problem for *Sort* could be fixed by including a guard $\neg sorted(s)$, but this makes it a refinement of *skip* only if property (2) is not imposed and guards can be strengthened. Another way would be to add a flag that ensures *Sort* is invoked exactly once after every occurrence of *Cin* or *Cycle* (possibly also preventing the next *Cin* until after sorting). A counter could be used to remove divergence in *Cycle*, with each of the other operations (excluding *Sort*) setting the counter to fix the maximal number of occurrences of *Cycle* to follow it, and *Cycle* decrementing it at every step until it is 0. None of those modifications would retain the property that *Sort* or *Cycle* refines *skip* if the prevalent refinement relation respects (2).

3.3 Internal Operations

An internal operation is a perspicuous operation with a special status: it is assumed to be invisible to the environment, and under internal control of the system only. In process algebras, internal operations naturally occur in a number of ways. In CSP [14] they arise from channels being hidden, for example encapsulating an internal communication channel when considering a system of communicating subsystems. They may also be used, for example in LOTOS [7], to encode internal choice when only external choice is available as a basic operator. Butler first considered the introduction of internal events in B refinement [8], and based on this approach we introduced "weak refinement" for Z [12, 10], which was analysed and compared to ASM refinement in detail by Schellhorn [18].

The requirements imposed in this context are inspired by how process algebras deal with internal actions, for example in defining "weak" bisimulation: where standard refinement conditions refer to a single action, their "weak" equivalents consider the same action possibly prefixed and postfixed by occurrences of internal actions. Thus, the refinement consistency property, e.g., will state that for every concrete action, with internal concrete behaviour before and after, its effect is consistent with the abstract action, possibly also pre- and postfixed with (abstract) internal behaviour. E.g. in [12] the restricted consistency (correctness) condition for weak refinement in Z (downward simulation) is phrased as

$$\mathrm{pre}(Int_A \mathbin{⨾} AOp) \wedge R \wedge (Int_C \mathbin{⨾} COp \mathbin{⨾} Int_C) \Rightarrow \exists AS' \bullet R' \wedge (Int_A \mathbin{⨾} AOp \mathbin{⨾} Int_A)$$

where Int_C is arbitrary internal behaviour in the concrete state, i.e. the transitive reflexive closure of the union of internal operations, and similar for Int_A. Taking this process algebra inspired approach has a few consequences:

[5] Note however that Abrial [1] does recognise (on page 414) a different class of operation that "*is not part of the protocol: it corresponds to a "daemon" acting ...*".

- internal actions have a special status which goes beyond the refinement step where they are introduced. They can not only be introduced this way, but must also be taken into consideration or can even be removed in subsequent refinement steps.
- there is an assumption that if internal actions are necessary for progress, they will "eventually" happen, so external operations are viewed as "enabled" if their before-state is reachable through internal behaviour; in timed process algebras in particular, internal actions are often taken as "urgent" meaning they happen as soon as they are enabled.
- there need not be independent refinement conditions for internal operations: all internal behaviour is viewed in the context of its composition with external behaviour. Thus, internal operations need *not* be refinements of *skip*. Of course, all internal operations being perspicuous, with external operations corresponding as normal, is *one* way of satisfying the refinement conditions like the one above, but it is not the only way. In fact, in some refinement relations, it may not be a viable way, see below.

The approaches for B and Z mentioned above only included *prevention* of divergence in weak refinement steps. A more general approach, also consistent with the process algebraic view, is to *preserve* or *reduce* any divergence that was already present in the abstract specification. This is worked out in detail in [6], and the impact of differing notions of "livelock" or divergence is discussed in [4]. The semantic relation established in this case is roughly that for every concrete trace, an abstract trace exists that is consistent with it, with both traces' subsequences of *external* actions being identical[6].

3.4 Action Refinement

Alphabet translation described above allows for arbitrary matchings of an occurence of an abstract action with the occurrence of a single concrete action. The most explicit way of changing the granularity of actions is to allow for matchings between *sequences* of abstract and concrete actions. This has been called "action refinement" [2] or "non-atomic refinement" [10]. In its most[7] general form, action refinement corresponds to ASM 1-to-n diagrams with n possibly greater than 1 [18], generalising the normal commuting simulation diagram to one where the concrete effect is achieved in n steps, without requiring a relation between abstract and *intermediate* concrete states. In this view, all concrete operations resulting from the decomposition are of the same status, with only their order having an impact on refinement conditions. This is also the view we took in definining non-atomic refinement for Z [10], work which was continued by Derrick and Wehrheim [13]. This kind of action refinement is even possible without changing the state space involved. It requires an explicit matching between abstract actions and concrete action sequences, which also extends to traces. The semantic relation aimed for is that concrete traces are consistent with abstract traces under this extended matching relation. The concrete and the abstract models end up having different interfaces with this approach – this may be exactly what is required, though. For example, [11, Ch. 13] has an example of a watch which in the abstract model has a *ResetTime* operation, which in the concrete model is represented by a series of executions of *ButtonA* and *ButtonB* operations.

Considering for simplicity now only the case that $n = 2$, the refinement requirements are like the introduction of sequential composition in refinement calculus [16]. Splitting an operation in two means

[6] In fact it is a somewhat more subtle matching: non-determinism included in a single operation on one abstraction level may be represented through a different choice of sequence of internal actions on the other level, so it is really a relation between sets of abstract vs. concrete traces with the same external subsequence.

[7] Avoiding for now the generalisation to m-to-n diagrams with $m \neq 1$.

finding an intermediate state (predicate) such that the first “half” lands in the intermediate state, and the second “half” moves from the intermediate to the original after-state. The problematic issue is what is or is not allowed to happen in the intermediate state. In a concurrent context, this comes under the heading of “interference” – when the first “half” of an operation has been executed, should other operations be disabled (non-interference, as e.g. discussed for action systems in [3]), or should their execution cancel out the effect of this one? This is a well-known problematic area, discussed also in [10], which we will not focus on here, as it is orthogonal to the issues discussed: when an action is split with part of it being perspicuous or internal, that also creates an intermediate state with the same potential interference problems.

4 How to Reduce Granularity in Refinement

From the discussion above, it should be clear that there are at least three semantic models for reducing the granularity of actions in refinement:

- by introducing perspicuous actions that take on some of the “work” – possibly requiring non-divergence;
- by introducing internal actions to the same effect – either using the limited refinement rules for perspicuous actions, or by using the more general “weak refinement” rules;
- by giving explicit decompositions of actions in which all parts have the same status.

We limit ourselves for now to the case where we are decomposing an action into two actions, where the first part could be viewed as “prepatory work”, and the second part as the “real work” – in other words, the situation in our example of refining *Aout* into *Sort* and *Cout*, where we expect *Sort* to be executed before *Aout*. However, in order to concentrate on the general situation, let us consider refining *AWork* into *Prepare* and *CWork*.

For the methods of reducing granularity by refining *skip*, we aim for *Prepare* to be perspicuous, and for *CWork* to be a refinement of *AWork*. Now consider an abstract state in which the operation *AWork* was applicable. If in every corresponding concrete state it would be possible to apply *CWork*, then we have a degenerate situation: we are introducing a new action *Prepare* whose contribution is unnecessary in all situations (i.e., it might as well be a *concrete skip*, too). Thus, in any relevant case of reducing granularity, *CWork* can be applicable in only a subset of the corresponding concrete states – namely those where *Prepare* has nothing (left) to do. Indeed, because *Prepare* is a refinement of an abstract *skip*, if its before-state is linked to a particular abstract state, then so should its after-state. Again in order to ensure that *Prepare* does something useful in some circumstances, there should be some abstract states linked to the before-states of *Prepare*.

This is where the prevalent notion of refinement makes a difference. If condition (2) (“enabledness”) is in force, we have made it impossible for *CWork* to be a refinement of *AWork*, because *CWork* is only applicable in a strict subset of the corresponding concrete states. This holds a fortiori for stronger versions of condition (2) such as failures refinement.

Thus, condition (2) excludes reduction of granularity by introducing perspicuous actions. It also excludes reduction of granularity by introducing internal actions using the “perspicuous actions” conditions. However, the more general “weak refinement” rules can be used in combination with condition (2), as we have shown in [6] in a context with condition (1) in force, and in [10] with condition (3) in force. This is explained by not being constrained to considering the concrete operation in isolation, but rather only considering it in the context of possible internal concrete behaviour.

The other way in which condition (2) is problematic for the refinements of *skip* is any requirements for perspicuous actions to be non-divergent. If they are refinements of *skip* respecting condition (2), then they are by definition applicable in all states and thus always applicable "again" and by definition divergent.

Returning to the example, ignoring *Cycle* for now, refinement reducing granularity is possible in several ways:

- by having *Sort* perspicuous, and guarded by $\neg sorted(s)$ if it is also required to be non-divergent. This works for trace refinement (just (1)), Event-B refinement, but not the other forms.
- by having *Sort* internal, provided it is guarded by $\neg sorted(s)$. This works according to the rules for Event-B, establishing normal Event-B refinement. However, it can also work for stronger refinement relations respecting condition (2), but then the more general weak refinement rules need to be used to establish it. In particular, it would mean that *Aout* is compared for refinement with $Sort^* \fatsemi Cout$.
- for explicit action refinement of *Aout* by *Sort* followed by *Cout*, there is no requirement for *Sort* to be guarded (compare the watch example referred to above: as conceptionally the user presses *ButtonB*, there is no guard preventing the user from doing that infinitely often), and refinement can be any kind, including relations respecting property (2) or even (3). In fact, including a guard on *Sort* would disallow the combined concrete output operation on states which are already sorted, and thus be unacceptable if the refinement relation obeys property (2).

5 Conclusion

The paradox that led to the discussion with Carroll Morgan referred to earlier was the following. If the work of one abstract operation is split between two concrete ones, and one of the concrete operations makes no progress that can be detected abstractly[8], why do we need this action at all? And if we do need it, how can the other concrete operation, achieving some but not all of the work of its abstract counterpart, be a refinement of the abstract one? The answer is hopefully somewhat clarified above. It requires a notion of refinement that allows for guards to be strengthened. The underlying issue may well have been known in "folklore" but it is not presented in any published papers we are aware of.

Coming back to Event-B specifically, two of its design decisions are thus closely entangled:

- to have essentially a trace semantics with only global deadlock prevention;
- to use stuttering step refinements for reducing granularity.

Both lead to relatively simple refinement obligations, which is attractive. In order for Event-B to strengthen refinement to preserve stronger properties such as encoded in various refusal-based semantics, it would also have to give up its simple notion of reduction of granularity. It could do this in at least two ways: either by going the way of ASM and having explicit recipes for decomposing operations with their corresponding conditions, or by going the way of process algebra, and giving certain operations explicit "internal" status which they then would need to retain subsequently. In either case, the price of gaining semantic strength is a considerable amount of complication of refinement conditions, which may be too big a price to pay, particularly for a formalism which now has so much (automated) proof tool support available. Would that be what Abrial had in mind when he wrote that (condition (2)) "happens to be sometimes too strong"?

[8] Thus, some degree of data refinement is implied: a refinement of *skip* on the *same* state really cannot make any progress.

Postscript

Finally, returning to the running example once more, a last word on the *Cycle* operation. It makes no useful progress whatsoever, but the constraints put upon this completely irrelevant operation in refinement in any "stuttering steps" approach (namely: taming its divergence), have been no more and no less than on the supposedly enormously useful *Sort* operation. Surely that is somewhat disappointing.

Acknowledgements

To Carroll Morgan for his explanations, to Michael Butler, John Derrick, Steve Dunne and Gerhard Schellhorn for useful discussions, and to the reviewers for their comments.

References

[1] J.-R. Abrial (2010): *Modelling in Event-B*. CUP.

[2] L. Aceto (1992): *Action Refinement in Process Algebras*. CUP.

[3] R.J.R. Back (1993): *Refinement of Parallel and Reactive Programs*. In M. Broy, editor: *Program Design Calculi*, pp. 73–92.

[4] E.A. Boiten & J. Derrick (2009): *Modelling divergence in Relational Concurrent Refinement*. In M. Leuschel & H. Wehrheim, editors: *IFM 2009: Integrated Formal Methods*, *LNCS* 5423, Springer Verlag, pp. 183–199, doi:10.1007/978-3-642-00255-7-13.

[5] E.A. Boiten & J. Derrick (2010): *Incompleteness of Relational Simulations in the Blocking Paradigm*. *Science of Computer Programming* 75(12), pp. 1262–1269, doi:10.1016/j.scico.2010.07.003.

[6] E.A. Boiten, J. Derrick & G. Schellhorn (2009): *Relational Concurrent Refinement Part II: Internal Operations and Outputs*. *Formal Aspects of Computing* 21(1-2), pp. 65–102, doi:10.1007/s00165-007-0066-z.

[7] T. Bolognesi & E. Brinksma (1988): *Introduction to the ISO Specification Language LOTOS*. *Computer Networks and ISDN Systems* 14(1), pp. 25–59, doi:10.1016/0169-7552(87)90085-7.

[8] M. Butler (1997): *An approach to the design of distributed systems with B AMN*. In J.P. Bowen, M G. Hinchey & D. Till, editors: *ZUM'97: The Z Formal Specification Notation*, *Lecture Notes in Computer Science* 1212, Springer-Verlag, pp. 223–241, doi:10.1007/BFb0027291.

[9] M. Butler (2009): *Decomposition Structures for Event-B*. In M. Leuschel & H. Wehrheim, editors: *IFM*, *Lecture Notes in Computer Science* 5423, Springer, pp. 20–38, doi:10.1007/978-3-642-00255-7-2.

[10] J. Derrick & E.A. Boiten (1999): *Non-atomic refinement in Z*. In J.M. Wing, J.C P. Woodcock & J. Davies, editors: *FM'99*, *Lecture Notes in Computer Science* 1708, Springer-Verlag, Berlin, pp. 1477–1496, doi:10.1007/3-540-48118-4_28.

[11] J. Derrick & E.A. Boiten (2001): *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT, Springer Verlag, doi:10.1007/978-1-4471-0257-1.

[12] J. Derrick, E.A. Boiten, H. Bowman & M.W.A. Steen (1998): *Specifying and Refining Internal Operations in Z*. *Formal Aspects of Computing* 10, pp. 125–159, doi:10.1007/s001650050007.

[13] J. Derrick & H. Wehrheim (2003): *Using coupled simulations in non-atomic refinement*. In D. Bert, J. Bowen, S. King & M. Walden, editors: *ZB 2003: Formal Specification and Development in Z and B*, *Lecture Notes in Computer Science* 2651, Springer, pp. 127–147, doi:10.1007/3-540-44880-2-10.

[14] C.A R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall.

[15] C.A.R. Hoare & He Jifeng (1998): *Unifying Theories of Programming*. Prentice Hall.

[16] C.C. Morgan (1994): *Programming from Specifications*, 2nd edition. International Series in Computer Science, Prentice Hall.

[17] S. Reeves & D. Streader (2008): *Data refinement and singleton failures refinement are not equivalent*. *Formal Aspects of Computing* 20(3), pp. 295–301, doi:10.1007/s00165-008-0076-5.

[18] G. Schellhorn (2005): *ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison*. *Theoretical Computer Science* 336(2-3), pp. 403–436, doi:10.1016/j.tcs.2004.11.013.

[19] J.C.P. Woodcock & J. Davies (1996): *Using Z: Specification, Refinement, and Proof*. Prentice Hall.