Under consideration for publication in Formal Aspects of Computing

Introducing extra operations in refinement

Eerke A. Boiten

School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, UK.
E.A.Boiten@kent.ac.uk www.cs.kent.ac.uk/~eab2

Abstract. This paper reconsiders refinements which introduce actions on the concrete level which were not present at the abstract level. It considers a range of different basic refinement relations, covering the standard ones for formalisms like Event-B, Z, action systems, and CSP. It also describes a number of ways in which new operations may be introduced: extended interfaces, internal actions, stuttering steps, and action refinement.

The main contribution of this paper is in exploring the interaction between those two dimensions. 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 more entangled than previously highlighted.

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

1. Introduction

The term "refinement" is used in two different but related senses in computer science. The older use of it, e.g. in Wirth's famous paper on "stepwise refinement" [Wir71] refers to a top-down program development method, gradually replacing informally described abstractions by finer, and more formal, detail. The current usage within e.g. the community around the Refinement Workshop series[1] focuses on a different aspect, namely on the preservation of correctness between specifications or programs in formal program development. "Stepwise refinement" then merely acknowledges the transitivity of this semantic relationship. The Event-B [Abr10] approach explicitly includes both interpretations, viewing refinement as a process that introduces detail to descriptions of the problem *as well as* to descriptions of its solution.

Clearly a refinement method in the "correctness preserving" sense also needs to encompass the gradual introduction of detail. In a state-based method like Z [WD96], ASM [Sch05], or Event-B, specifications consist of descriptions of state spaces, with operations on those states. Extra detail is brought into the *state space* through the process of data refinement [HHS86, dRE98]; in the functionality of *operations* it appears through the introduction of extra operations. The latter has been explored for standard Z refinement relations in [DB01, Ch.11-14], and for Event-B in [ACM05]. However, neither of these provides a complete picture. The monograph [DB01] pays some attention to basic refinement relations for languages like Z, but does not fully discuss how the choice of these (traces, failures, etc. – see below) interacts with the various

Correspondence and offprint requests to: E.A. Boiten

[1] See `http://www.refinenet.org.uk/`, and various special issues of this journal.

methods for adding operations. Some design decisions for Event-B refinement are discussed in passing in [ACM05, Abr10], but also there the interaction between the choice of basic refinement relation and options for operation decomposition is not explored.

The motivation of this paper is to examine the interaction between these two "dimensions" of refinement, i.e., how these two "design decisions" for a specification notation interact: how can implementation freedom in operations and state spaces be resolved, and how can the set of operations be extended to include extra ones, possibly at a lower level of granularity? A large number of options are available for the basic refinement relation, see e.g. [vG01], and several methods exist for adding operations (such as weak refinement, action refinement, stuttering steps). We consider the possibilities for combining such choices – in particular, show how the choices in those two dimensions are interdependent, and we highlight how this affects Event-B in particular, where the current choices appear to have led to a "sweet spot".

This paper extends the workshop paper [Boi11] with a formal rather than informal description of the basic refinement relations, more explicit motivation and conclusions, and significantly extended discussion throughout including on the role of retrieve relations, on interference, of the running example, and of related literature.

Before describing the issues in detail, we consider an example. The example is presented in Z, but it is important to note that the notation used is not essential to what follows. In general, most of what is described in this paper could be expressed in ASM, (Event-)B, Z, binary relations [DB01], UTP [HH98] or other state-based formalisms; for the moment we make no assumptions about what basic 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 *sorted* (in increasing numerical order – used in the next specification) on sequences are omitted. The schema AS describes system states, $AInit$ characterises possible initial states, and the schemas Ain and $Aout$ define the operations of adding and removing an element. As in [DB01], we formalise the "states and operations" style for Z by encapsulating its elements in an "abstract data type" (ADT): a tuple $(State, Init, \{Op_i\}_{i \in I})$, i.e. in this particular case it is $(AS, AInit, \{Ain, Aout\})$. In the operation $Aout$, the precondition $b \neq [\![\,]\!]$, which is already implied by the other predicates, is included explicitly, in recognition of it having to be an explicit guard in other notations such as Event-B.

AS

$b : \mathrm{bag}\,\mathbb{N}$

$AInit$

AS'

$b' = [\![\,]\!]$

Ain

ΔAS
$x? : \mathbb{N}$

$b' = b \uplus [\![x?]\!]$

$Aout$

ΔAS
$x! : \mathbb{N}$

$b \neq [\![\,]\!]$
$b = b' \uplus [\![x!]\!]$
$x! = min(b)$

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 found at the head of the sequence. The schema *Sort* describes the sorting of the sequence. The schema *Cycle* is an entirely

artificial operation[2], used to make a point later on, and was not part of Morgan's original example.

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

CInit
CS'

$s' = \langle\,\rangle$

Cin
ΔCS
$x? : \mathbb{N}$

$s' = s \frown \langle x? \rangle$

Cout
ΔCS
$x! : \mathbb{N}$

$s \neq \langle\,\rangle$
$sorted(s)$
$s = \langle x! \rangle \frown s'$

Sort
ΔCS

$items\ s = items\ s'$
$sorted(s')$

Cycle
ΔCS

$s = \langle\,\rangle \land s' = \langle\,\rangle \lor$
$s' = (tail\ s) \frown \langle head\ s \rangle$

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 $SortOut == Sort \fatsemi Cout$, whose meaning is given by

SortOut
ΔCS
$x! : \mathbb{N}$

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

Then, in most sensible refinement relations, uncontroversially, the operation *Aout* is refined by *SortOut* (or more precisely: the data type $(AS, AInit, \{Ain, Aout\})$ is refined by $(CS, CInit, \{Cin, SortOut\})$). The retrieve relation which validates this data refinement is $b = items\ s$. In fact, this is normally an equivalence: refinement also holds in the reverse direction. However, the relationship of the abstract datatype to the data type with *three* operations $(CS, CInit, \{Cin, Cout, Sort\})$ is more open to discussion.

Finally, note that one might also consider the composition $InSort == Cin \fatsemi Sort$. The data type $(CS, CInit, \{InSort, COut\})$ is also a refinement of the abstract data type, under the retrieve relation $b = items\ s \land sorted(s)$. In that case, the "intermediate" concrete state between *Cin* and *Sort* does not generally relate to any abstract state, as its sequence s may not be sorted. As it is only ever applied to sorted sequences s, the operation *InSort* simplifies to the "insert" operation of insertion sort.

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 by 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

For a more detailed coverage of different basic notions of refinement and concurrent systems, see e.g. the monograph [DB01] or the research papers [BDS09, BD10]. These include detailed descriptions in terms of binary relations, and derivations of formulations in Z for data refinement, using both upward and downward

[2] 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 cyclic list, . . . maybe.

simulations, representing a wide range of commonly used refinement relations for state-based systems and process algebras. In the current paper, we gloss over a few aspects, namely by providing only downward simulation rules for data refinement, omitting rules relating to initialisation and finalisation, and describing all conditions in Z directly. It also omits most consideration of inputs or outputs, and more generally of what observations can be made of such data types.

In *basic* data refinement, systems (which may sometimes be called "machines" or "abstract data types") are compared which have identical sets of labels of operations (a.k.a. "actions" or "events"), also known as "alphabets". Informally, the conditions on operations are some subset and interpretation of:

(1) Consistency The effect of the concrete operation corresponds to an effect that is allowed by the abstract operation in a corresponding state.

(2) Restricted consistency In states where the abstract operation is enabled in a corresponding state, the effect of the concrete operation corresponds to an effect that is allowed by the abstract operation in such a state.

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

In establishing such a data refinement, the choice of what "corresponding" states are between the two systems can be chosen freely in any way that jointly satisfies these conditions – this choice is represented in a "retrieve relation" or "coupling invariant".

We now describe formalisations of these conditions, and how they are embodied in different refinement relations as defined for different specification formalisms.

2.1. Consistency

Property (1) or its weaker variant (2) 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 (1) obviously implies (2). It also implies a converse of (3): where concrete operations are enabled (leading to an "effect"), their abstract counterparts should be enabled, too (in order to allow comparison of effects).

The standard Z formulation of consistency requires an exact characterisation of how abstract and concrete states correspond, and this retrieve relation is represented in Z as a predicate (schema) on the two state spaces. Consistency between operation sets $\{AOp_i\}_{i\in I}$ and $\{COp_i\}_{i\in I}$, assuming that R relates abstract state $AState$ and concrete state $CState$ (and thus R' establishes the same relation for after-states $AState'$ and $CState'$), is

$$\forall\, AState;\ CState;\ CState';\ i : I \bullet R \land COp_i \Rightarrow \exists\, AState' \bullet AOp_i \land R' \tag{4}$$

and for restricted consistency it is

$$\forall\, AState;\ CState;\ CState';\ i : I \bullet R \land COp_i \land \exists\, AState' \bullet AOp \Rightarrow \exists\, AState' \bullet AOp_i \land R' \tag{5}$$

The added (third) conjunct on the left-hand side corresponds to the computed precondition of AOp, defined as $\mathrm{pre}\, AOp == \exists\, AState' \bullet AOp$. Restricted consistency is the standard ("non-blocking") condition for Z [WD96, DB01], and often called the "correctness" condition. It is also sometimes called the "contract" approach as it only constrains the implementation of operations within their agreed preconditions. In a relational form the two conditions above are

$$\mathsf{R} \fatsemi \mathsf{COp} \subseteq \mathsf{AOp} \fatsemi \mathsf{R}$$
$$(\mathrm{dom}\, \mathsf{AOp}) \lhd \mathsf{R} \fatsemi \mathsf{COp} \subseteq \mathsf{AOp} \fatsemi \mathsf{R}$$

In this relational form, the consistency condition is straight from the standard theory [HHS86], whereas the restricted consistency condition can be derived from it under a different interpretation of the domain of a relation, see [WD96, DB01] for details.

Trace refinement is characterised by just (1), 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.

2.2. Enabledness

Property (3) ensures that the client is indeed able to perform the same "experiments" on both systems. Leaving it out is, informally, justified by a contrapositive characterisation of refinement: refinement holds whenever it is impossible to disprove it, i.e. when it is impossible to perform an experiment that demonstrates a breach of the consistency property. In describing non-reactive systems, i.e. when there is only a notion of "possible behaviour" rather than "possible interaction", and experiments are not interactive, this is a reasonable point of view.

Most of the subtle differences between refinement relations are caused by varieties in how operations' readiness to be invoked is being observed: e.g. in particular states, or following particular behaviours; on the basis of sets of operations, or for each operation individually; and recording operations that *can*, or that *cannot* be invoked.

A condition similar to (3) but even weaker[3] is used in basic *Event-B refinement* (called simple refinement in [Abr10, Ch. 14]): if the concrete state deadlocks (i.e. no events are enabled), then so should the abstract state. In a Z formulation, this would be

$$\forall\, AState;\ CState \bullet (R \land \forall\, i : I \bullet \neg \operatorname{pre} COp_i) \Rightarrow \forall\, i : I \bullet \neg \operatorname{pre} AOp_i \tag{6}$$

Although "deadlock" has a negative connotation, there is also a positive view of this, namely that the situation where no events are enabled represents successful termination, e.g. of an action system [Bac93], and this condition ensures that the concrete system does not terminate prematurely. Enabledness of individual events in Event-B is given by explicitly specified guards, with a "feasibility" proof obligation ensuring that they are at least as strong as any computed precondition. Abrial [Abr10, p. 429] states that a stronger condition than (6) could be imposed, but "this happens to be sometimes too strong". (We will return to this.)

In standard Z refinement, condition (3) is called "applicability" and typically formulated as

$$\forall\, AState;\ CState;\ i : I \bullet \operatorname{pre} AOp_i \land R \Rightarrow \operatorname{pre} COp_i \tag{7}$$

The same condition also appears in downward simulation conditions for CSP-based refinement relations, such as failures refinement.

2.3. Consistency and Enabledness

Combining the two aspects, we consider the following collection of refinement relations.

T Pure trace refinement, as characterised by only (4).

EB Event-B simple refinement, as characterised by (4) and (6).

Z Traditional (downward simulation) Z refinement, characterised by (5) and (7).

ZB Blocking Z refinement, characterised by (4) and (7).

F Failures refinement, characterised by the same basic conditions as **ZB**, i.e. (4) and (7), but crucially stronger conditions [BD06] in the (here omitted) upward simulation rules, which allow for postponement of non-deterministic choice. We refer to [vG01, BD06, BDS09, RS08, BD10] for detailed discussion of these refinement relations and their close relatives such as singleton failures refinement, and the finer distinctions between them, which are not relevant in the current paper.

We will refer to these refinement relations by the short names in bold.

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

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

[3] ..., at least in a system with ≥ 2 operations.

specification to have "extra" operations – we discuss the ones that have been proposed in the literature on state-based specification languages. First, we mention alphabet extension and alphabet translation [DB01, Ch. 14] for completeness. Then, we get to the core of this paper: stuttering steps, the introduction of internal operations in various ways, 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 **Z**: it means we allow implementors to provide functionality that we had not asked for. In a process algebra context, e.g. for **ZB** or **F**, or indeed any refinement relation based on (1) rather than (2), alphabet extension is typically not allowed, and indeed that makes sense in our intuitive view: it would go against refinement property (1), by having no matching abstract behaviour for some concrete behaviour.

The practical risk of alphabet extension (i.e., arbitrary addition of new procedures/methods/...) is related to *implicit invariants*. Data types may include explicit invariants on their state space, and any new operations could not violate these. However, there may also be invariants which are not prescribed, but effectively established by initialisations and maintained by all operations ([DB01, Ch. 2]), and analysis based on state space exploration from some initial state rather than direct proof (e.g. in testing or model checking) will not find the safety problems that might arise in "unreachable" states. Newly added methods however may break implicit invariants, and consequently potentially make "unsafe" states reachable, thus invalidating the results of previous safety analysis. This relates to the issue of interference in action decomposition discussed below, for "intermediate states" that were previously unreachable. However, even in this respect alphabet extension is conceptually consistent with *restricted* consistency, as this *also* allows more states to become reachable.

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 as described above is subsumed within alphabet translation.) 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. This was introduced in [ACM05] and described in detail in [Abr10].

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. Using such an explicit mapping, it could feasibly be combined with any of the refinement relations considered here, in line with the observation in [DB01, Ch. 14] that many of the generalisations to refinement presented in that monograph are orthogonal.

3.2. Perspicuous Operations

Having considered "extra" operations that relate directly to new behaviour, or directly to existing behaviour, we now move on to concrete operations whose abstract counterparts are not so obvious.

3.2.1. Refining skip

State-based systems potentially change state when operations are executed. When no operation is invoked, the state does not normally change. Some formalisms [Lam94, Hes05] take this into account by explicitly including so-called stuttering steps in their semantics: steps where the state does not change between two observations, due to no event's 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, as a more general concept than "internal events" (see below) which come with additional assumptions and requirements. In particular, in subsequent refinement steps, perspicuous operations do *not* generally have a different status from operations that were present earlier.

Refining an abstract *skip* by a concrete *skip* is, of course, both automatic and useless. The possibility of refining an abstract *skip* to a concrete operation which is not *skip* only arises when the retrieve relation is not a function from abstract to concrete states. (However, refinement as the *introduction* of implementation detail makes it more likely that the retrieve relation is functional in the opposite direction, i.e., from concrete to abstract.) More generally, a concrete operation's being labelled as perspicuous depends on the retrieve relation used. However, this does not mean that the labelling as "perspicuous" is arbitrary: the retrieve relation considered also needs to ensure that data refinement conditions are satisfied for all *other* operations. In some cases indeed different retrieve relations, leading to different perspicuous operations, can prove essentially the same refinement – see the work by Banach and Schellhorn [BS10] on "synchronisation points".

Abrial [Abr10] presents a related motivation for the introduction of new events in Event-B, analogous to how this is done in action systems [Bac93], 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 a perspicuous operation which achieves nothing in the abstract world can hardly be useful by itself[4]. 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. Thus, we obtain the natural generalisation of condition (6) which accounts for a larger concrete alphabet: let J be the set of newly introduced perspicuous actions, then it is:

$$\forall\, AState;\ CState \bullet (R \wedge \forall\, i : I \cup J \bullet \neg \operatorname{pre} COp_i) \Rightarrow \forall\, i : I \bullet \neg \operatorname{pre} AOp_i \tag{8}$$

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 enabledness property (3) is imposed. None of the other operations are potentially perspicuous as *skip* would not normally have inputs, and certainly no outputs.

3.2.2. Divergence

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 is enabled. 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 [Abr10] explains it[5]. With additional assumptions, such as that a system might perform perspicuous operations independently, divergence becomes a practical as well as a theoretical problem. Butler [But09] 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 these are not just perspicuous events, but even *internal* events like we will discuss next. In action systems [Bac93], 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[6]. A typical method of proving non-divergence is by establishing a variant (a well-founded, strictly decreasing function) on newly introduced (collections of) perspicuous operations [But97, DBBS98, Abr10]. If refinement is based on property (1) rather

[4] 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.

[5] His use of the term "reachable" is possibly 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.

[6] Note however that Abrial [Abr10] does recognise (on page 414) a different class of operation that "*is not part of the protocol: it corresponds to a "daemon" acting ...*", which corresponds to the normal interpretation of an internal operation.

than property (2), i.e., an action cannot gain behaviour in refinement, then non-divergence is preserved by subsequent refinement steps[7].

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 [Lam94], this may not be a problem: nearly all the new concrete steps are stuttering steps, too.

Removing divergence on each of the operations individually can be done using several possible small modifications. The divergence problem for *Sort* could be fixed by including a guard $\neg sorted(s)$, i.e.,

SortIfNeeded

ΔCS

$items\ s = items\ s'$

$\neg sorted(s) \land sorted(s')$

but this makes it a refinement of *skip* only if no enabledness property (3) is imposed and guards can be strengthened.

Another way would be to add a flag that ensures *Sort* is invoked exactly once after every sequence of occurrences of *Cin* or *Cycle*, i.e.,

*CS*2

$s : \mathrm{seq}\,\mathbb{N};\ gosort : \mathbb{B}$

*CInit*2

$CS2'$

$s' = \langle\,\rangle \land \neg gosort'$

*Cin*2

$\Delta CS2$

$x? : \mathbb{N}$

$s' = s \frown \langle x? \rangle \land gosort'$

*Cout*2

$\Delta CS2$

$x! : \mathbb{N}$

$s \neq \langle\,\rangle$

$sorted(s) \land \neg gosort'$

$s = \langle x! \rangle \frown s'$

*Sort*2

$\Delta CS2$

$items\ s = items\ s'$

$gosort \land \neg gosort' \land sorted(s')$

*Cycle*2

$\Delta CS2$

$s = \langle\,\rangle \land s' = \langle\,\rangle \lor$

$s' = (tail\ s) \frown \langle head\ s \rangle \land gosort'$

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. Showing just one of the other operations (*CInit*3 and *Cin*3 fail to mention *cycles*,

[7] This is another point where Event-B design decisions come together nicely: in subsequent refinement steps, perspicuous Event-B events will not have a special status even if their interpretation may be as "internal" events. One of the reasons for retaining such a special status might be the non-divergence requirement on such operations; however, the use of (1) rather than (2) in basic Event-B refinement ensures that this requirement is automatically preserved rather than having to be re-checked.

which implies it is non-deterministically set to any natural number):

*CS*3
$s : \operatorname{seq} \mathbb{N};\ cycles : \mathbb{N}$

*CInit*3
$CS3'$
$s' = \langle\,\rangle$

*Cin*3
$\Delta CS3$
$x? : \mathbb{N}$
$s' = s \frown \langle x? \rangle$

*Cycle*3
$\Delta CS3$
$cycles > 0$
$s = \langle\,\rangle \land s' = \langle\,\rangle \lor$
$s' = (tail\ s) \frown \langle head\ s \rangle \land cycles' = cycles - 1$

Divergence of both operations could be removed by appropriate combinations of these modifications.

Each of these fixes would work for **T**, i.e., it would establish refinement between the abstract data type and the modified concrete data type while guaranteeing non-divergence. They would also work for **EB** as condition (6) holds trivially, due to the fact that always *some* operation is enabled. None of those modifications would retain the property that *Sort* or *Cycle* refines *skip* if the prevalent refinement relation respects enabledness condition (3), because in each case the enabledness of the perspicuous operation is strictly strengthened. Fundamentally, there is no way to remove divergence of perspicuous operations that would preserve **Z**, **ZB** or **F** – due to refinements of *skip* necessarily having to be everywhere enabled like *skip* itself.

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. Thus, the introduction of internal operations does *not* extend the alphabet of available actions that a client could use. In process algebras, internal operations naturally occur in a number of ways. In CSP [Hoa85] 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 [**?**], 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 [But97], and based on this approach we introduced "weak refinement" for Z [DBBS98], as a generalisation of **Z**, which was analysed and compared to ASM refinement in detail by Schellhorn [Sch05].

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 equivalent of the refinement consistency property, e.g. property (4), 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 [DBBS98] the restricted consistency (correctness) condition for weak refinement in Z (downward simulation) is phrased as

$$\operatorname{pre}(Int_A \fatsemi AOp) \land R \land (Int_C \fatsemi COp \fatsemi Int_C) \Rightarrow \exists\, AS' \bullet R' \land (Int_A \fatsemi AOp \fatsemi Int_A) \tag{9}$$

where Int_C is arbitrary finite 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:

- 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*.
One way of satisfying the refinement conditions is still to ensure that all internal operations are perspicuous, 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 [BDS09], essentially for **F** although it could be done for **NB** as well, and the impact of differing notions of "livelock" or divergence is discussed in [BD09]. 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[8].

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" [Ace92] or "non-atomic refinement" [DB99]. In its most[9] general form, action refinement corresponds to ASM 1-to-n diagrams with n possibly greater than 1 [Sch05], 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 [DB99] (using **Z** as the basic refinement relation), work which was continued by Derrick and Wehrheim [DW03]. 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, [DB01, 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 [Mor94]. Splitting an operation in two means 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 [Bac93]), or should their execution cancel out the effect of this one? This is a well-known problematic area, discussed also in [DB99]. If an action is split with part of it being perspicuous or internal, that also creates an intermediate state. However, the potential interference problems arising from that are being adequately addressed in the refinement conditions: for a perspicuous concrete action, its before and after state are equivalent as far as the abstract data type is concerned, and thus are subject to the same consistency and enabledness checks for all *other* operations. For internal actions, using weak refinement, the inclusion of internal behaviour in *every* operation's verification conditions (such as (9)) ensures that interference is dealt with appropriately.

[8] 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.
[9] Avoiding for now the generalisation to m-to-n diagrams with $m \neq 1$.

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 under conditions that prevent divergence, i.e., allow only finite subtraces consisting of only perspicuous actions;
- 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, possibly including some conditions to prevent interference in intermediate states.

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 (3) ("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 strong versions of condition (3) such as failures refinement.

Thus, condition (3) 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 (3), as we have shown in [BDS09] for **F**, and in [DB99] for **Z**. 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 (3) 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 (3), 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 **T** ands **EB**, but not the other forms.
- by having *Sort* internal, provided it is guarded by $\neg sorted(s)$. This works for **EB**, however, it can also work for stronger refinement notions like **ZB** and **F**, 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 it is the user who 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 (3).

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[10], 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 those 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 (3)) "happens to be sometimes too strong"?

Summarizing the overall discussion, we have the following compatibility table for the various methods of extending alphabets and decomposing operations for the five refinement relations discussed.

	Z	**ZB**	**T**	**EB**	**F**
Alphabet extension	√[DB01]	×[DB01]	×	×	×
Alphabet translation	√[DB01]	√[DB01]	√	√[Abr10]	√
Perspicuous operations (no divergence constraint)	√	√	√	√	√
Perspicuous operations (non-divergent)	×	×	√	√[Abr10]	×
Internal operations (non-divergent, perspicuous)	×	×	√	√[But09]	×
Internal operations (weak refinement)	√[DBBS98]	√[BDS09]	√	√([But97]?)	√[BDS09]
Action refinement [Sch05]	√[DB99, DW03]	√	√	√?	√

In this table, √ indicates that a particular combination is possible, with citations indicating where it has been worked out in some detail; × indicates that the combination cannot be usefully exploited, or that it does not appear to fit with the spirit of the approach.

Postscript

Finally, returning to the running example once more, a 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.

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

References

[Abr10] Abrial J-R (2010) *Modelling in Event-B*. CUP

[Ace92] Aceto L (1992) *Action Refinement in Process Algebras*. CUP

[ACM05] Abrial J-R, Cansell D, Méry D (2005) Refinement and reachability in Event-B. In Treharne H, King S, Henson MC, Schneider SA (eds) *ZB*, volume 3455 of *LNCS*, pages 222–241. Springer

[Bac93] Back RJR (1993) Refinement of parallel and reactive programs. In Broy M (ed) *Program Design Calculi*, pages 73–92

[BB88] Bolognesi T, Brinksma E (1988) Introduction to the ISO Specification Language LOTOS. *Comput Networks ISDN* 14(1):25–59

[BD06] Bolton C, Davies J (2006) A singleton failures semantics for Communicating Sequential Processes. *Form Asp Comp* 18:181–210

[BD09] Boiten EA, Derrick J (2009) Modelling divergence in relational concurrent refinement. In Leuschel M, Wehrheim H (eds) *IFM 2009*, volume 5423 of *LNCS*, pages 183–199. Springer

[BD10] Boiten EA, Derrick J (2010) Incompleteness of relational simulations in the blocking paradigm. *Sci Comput Program* 75(12):1262–1269

[BDS09] Boiten EA, Derrick J, Schellhorn G (2009) Relational concurrent refinement II: Internal operations and outputs. *Form Asp Comp* 21(1-2):65–102

[Boi11] Boiten EA (2011) Perspicuity and granularity in refinement. In Derrick J, Boiten EA, Reeves S (eds) *Refinement Workshop 2011*, volume 55 of *EPTCS*, pages 155–165

[BS10] Banach R, Schellhorn G (2010) Atomic actions, and their refinements to isolated protocols. *Form Asp Comp* 22(1):33–61

[But97] Butler M (1997) An approach to the design of distributed systems with B AMN. In Bowen JP, Hinchey MG, Till D (eds) *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 223–241. Springer

[But09] Butler M (2009) Decomposition structures for Event-B. In Leuschel M, Wehrheim H (eds) *IFM*, volume 5423 of *LNCS*, pages 20–38. Springer

[DB99] Derrick J, Boiten EA (1999) Non-atomic refinement in Z. In Wing JM, Woodcock JCP, Davies J (eds) *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 1477–1496. Springer

[DB01] Derrick J, Boiten EA (2001) *Refinement in Z and Object-Z*. Springer

[DBBS98] Derrick J, Boiten EA, Bowman H, Steen MWA (1998) Specifying and refining internal operations in Z. *Form Asp Comp* 10:125–159

[dRE98] De Roever WP, Engelhardt K (1998) *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP

[DW03] Derrick J, Wehrheim H (2003) Using coupled simulations in non-atomic refinement. In Bert D, Bowen JP, King S, Waldén M (eds) *ZB 2003*, volume 2651 of *LNCS*, pages 127–147. Springer

[Hes05] Hesselink WH (2005) Eternity variables to prove simulation of specifications. *ACM T Comput Log* 6(1):175–201

[HH98] Hoare CAR, He Jifeng (1998) *Unifying Theories of Programming*. Prentice Hall

[HHS86] He Jifeng, Hoare CAR, Sanders JW (1986) Data refinement refined. In Robinet B, Wilhelm R (eds) *Proc. ESOP 86*, volume 213 of *LNCS*, pages 187–196. Springer

[Hoa85] Hoare CAR (1985) *Communicating Sequential Processes*. Prentice Hall

[Lam94] Lamport L (1994) The temporal logic of actions. *ACM T Prog Lang Sys* 16(3):872–923

[Mor94] Morgan CC (1994) *Programming from Specifications*. International Series in Computer Science. Prentice Hall, 2nd edition

[RS08] Reeves S, Streader D (2008) Data refinement and singleton failures refinement are not equivalent. *Form Asp Comp* 20(3):295–301

[Sch05] Schellhorn G (2005) ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theor Comput Sci* 336(2-3):403–436

[vG01] Van Glabbeek RJ (2001) The linear time - branching time spectrum I. The semantics of concrete sequential processes. In Bergstra JA, Ponse A, Smolka SA (eds) *Handbook of Process Algebra*, pages 3–99. North-Holland

[Wir71] Wirth N (1971) Program development by stepwise refinement. *Commun ACM* 14:221–227

[WD96] Woodcock JCP, Davies J (1996) *Using Z: Specification, Refinement, and Proof*. Prentice Hall