Model Based Refinement and the Tools of Tomorrow

Richard Banach

School of Computer Science,
University of Manchester,
Manchester, M13 9PL, UK
banach@cs.man.ac.uk

Abstract. The ingredients of typical model based development via refinement are re-examined, and some well known frameworks are reviewed in that light, drawing out commonalities and differences. It is observed that alterations in semantics take place *de facto* due to applications pressures and for other reasons. This leads to a perspective on tools for such methods in which the proof obligations become programmable and/or configurable, permitting easier co-operation between techniques and interaction with an Evidential Tool Bus. This is of intrinsic interest, and also relevant to the Verification Grand Challenge.

Key words: Model Based Development, Refinement, Configurable Proof Obligations, Tools, Verification Grand Challenge

1 Introduction

Refinement, as a model based methodology for developing systems from abstract specifications, has been around for a long time [1]. In this period, many variations on the basic idea have arisen, to the extent that an initiate can be bewildered by the apparently huge choice available. As well as mainstream refinement methodologies such as ASM, B, Z, etc., which have enjoyed significant applications use, there are a myriad other related theories in the literature, too numerous to cite comprehensively. And at a detailed theoretical level, they are all slightly different.

From a developer's point of view, this variety can only be detrimental to the wider adoption of formal techniques in the real world applications arena — in the real world, developers have a host of things to worry about, quite removed from evaluating the detailed technical differences between diverse formal techniques in order to make the best choice regarding which one to use. In any event, such choice is often made on quite pragmatic grounds, such as the ready access to one or more experts, and crucially these days, availability of appropriate tool support. Anecdotally, the choice of one or another formalism appears to make little difference to the outcome of a real world project using such techniques — success seems to be much more connected with proper requirements capture, and with organising the development task in a way that is sympathetic to both the formal technique and to the developers' pre-existing development practices.

In this paper we examine closely what goes into a typical notion of model based refinement by examining a number of cases. As a result, we can extract the detailed similarities and differences, and use this to inform a view on how different techniques ought

to relate to one another. This in turn forms a perspective on how different techniques can meet a contemporary environment in which verification techniques and their tools can increasingly address mainstream industrial scale problems — determining how to address the spectrum of technical differences between techniques, in the face of a wider world prone to see them as intrinsically divisive, remains a significant challenge. In this paper, we contend that techniques in this field can be viewed as comprising a number of features, amongst which, the commonly occuring ones ought to be emphasised, and the more specific ones deserve to be viewed more flexibly. This line is developed to the point that a landscape can be imagined, in which different techniques, and their tools, can ultimately talk to one another.

The rest of the paper is as follows. In Section 2 we cover the common features of model based formalisms. In Section 3 we show how these generalities are reflected in a number of specific well known approaches. Section 4 reflects on the evidence accumulated in the previous one and draws some appropriate conclusions. Section 5 takes the preceding material and debates the implications for tools. It is suggested that increased programmability can substantially help to bridge the gaps between techniques, and the way that programmability features in some recent tools is discussed. These thoughts are also in sympathy with the SRI 'Evidential Tool Bus' idea [2], and can contribute positively towards the current Verification Grand Challenge [3–5]. Section 6 concludes.

2 Model Based Refinement Methods: Generalities

A typical model based formal refinement method, whose aim is to formalise how an abstract model may be refined to a more concrete one, consists of a number of elements which interact in ways which are sometimes subtle. In this section we bring some of these facets into the light; the discussion may be compared to a similar one in [6].

Formal language. All formal refinement techniques need to be quite specific about the language in which the elements of the technique are formalised. This precision is needed for proper theoretical reasoning, and to enable mechanical tools with well defined behaviour to be created for carrying out activities associated with the method. There are inevitably predicates of one kind or another to describe the properties of the abstract and concrete models, but technical means are also needed to express state change within the technique. Compared with the predicates used for properties, there is much more variety in the linguistic means used for expressing state change, although each has a firm connection with the predicates used for the modelling of properties.

Granularity and naming. All formal refinement techniques depend on relating concrete steps (or collections of steps) to the abstract steps (or collections of steps) which they refine. Very frequently, a single concrete step is made to correspond to a single abstract one, but occasionally more general schemes (in which sequences of abstract and concrete steps figure) are considered. The $(1,1)$ scheme is certainly convenient to deal with theoretically, and it is often captured by demanding that the names of operations or steps that are intended to correspond at abstract and concrete levels are the same. However, in many applications contexts, such a simple naming scheme is far removed from reality, and if naively hardwired into the structure of a tool, makes the tool much less conveniently usable in practice.

Concrete-abstract fidelity. All formal refinement techniques demand that the concrete steps relate in a suitable manner to abstract ones. Almost universally, a retrieve relation (also referred to as a refinement mapping, abstraction relation, gluing relation, etc.) is used to express this relationship. It is demanded that the retrieve relation holds between the before-states of a concrete step (or sequence of steps) and the abstract step (or sequence of steps) which simulates it; likewise it must hold for the after-states of the simulating pair. In other words (sequences of) concrete steps must be faithful to (sequences of) abstract steps. (A special case, simple refinement, arises when the retrieve relation is an identity.)

Concrete-abstract fidelity is the one feature that can be found in essentially the same form across the whole family of model based formalisms. It is also the case that this fidelity —usually expressed using a proof obligation (PO), the *fidelity PO*— is often derived as a *sufficient condition* for a more abstract formulation of refinement, concerning the overall behaviour of 'whole programs'. These sufficient conditions normally form the focus of the theory of model based refinement techniques, since they offer what is usually the only route to proving refinement in practical cases.

Notions of correctness. One of the responsibilities of a formal refinement technique is to dictate *when* there should be concrete steps that correspond to the existence of abstract ones. This (at least implicitly) is connected with the potential for refinement techniques to be used in a black-box manner. Thus if an abstract model has been drawn up which deals adequately with the requirements of the problem, then any refinement should *guarantee* that the behaviour expressed in the abstract model should be reflected appropriately in more concrete models, and ultimately in the implementation, so that the requirements coverage persists through to code.

There is much variation among refinement techniques on how this is handled, particularly when we take matters of interpretation into account. Although the mainstream techniques we discuss below are reasonably consistent about the issue, some variation is to be found, and more variety can be found among refinement variants in the literature. The formal content of these perspectives gets captured in suitable POs, and often, the policy adopted has some impact on the fidelity PO too. A similar impact can be felt in initialisation (and finalisation) POs.

Interpretation. The preceding referred (rather obliquely perhaps) to elements of model based refinement theories that are expressed in the POs of the theory, i.e. *via logic*. However, this does not determine how the logical elements relate to phenomena in the real world. If transitions are to be described by logical formulae (involving before and after states, say), then those formulae can potentially take the value false as well as true. And while determining how the logical formulae correspond to the real world is usually fairly straightforward in the true case, determining the correspondence in the false case can be more subtle. These matters of logical-to-real-world correspondence constitute the *interpretation* aspects of a formal development technique.

Trace inclusion. Trace inclusion, i.e. the criterion that every execution sequence of the system (i.e. the concrete model) is as permitted by the specification (i.e. the abstract model), is of immense importance in the real world. When an implemented system behaves unexpectedly, the principal *post hoc* method of investigation amounts to determining how the preceding behaviour failed to satisfy the trace inclusion criterion. This

importance is further underlined by the role that trace inclusion plays in model checking. The 'whole program' starting point of the derivation of many sufficient conditions for refinement is also rooted in trace inclusion. Two forms of trace inclusion are of interest. *Weak trace inclusion* merely states that for every concrete trace there is a simulating abstract one. *Strong trace inclusion* goes beyond that and states that if *Asteps* simulates *Csteps* and we extend *Csteps* to $Csteps \fatsemi C_{nxt}$, then *Asteps* can be extended to $Asteps \fatsemi A_{nxt}$ which also simulates. With weak trace inclusion, we might have to abandon *Asteps* and find some unrelated $Asteps_{different}$ to recover simulation of $Csteps \fatsemi C_{nxt}$.

Composition. It is a given that large systems are built up out of smaller components, so the interaction of this aspect with the details of a refinement development methodology are of some interest, at least for practical applications. Even more so than for notions of correctness, there is considerable variation among refinement techniques on how compositionality is handled — the small number of techniques we review in more detail below already exhibit quite a diversity of approaches to the issue.

3 ASM, B, Event-B, Z

In this section, we briefly review how the various elements of model based methods outlined above are reflected in a number of specific and well-known formalisms. For reasons of space, we restrict to the ASM, B (together with the more recent Event-B) and Z methodologies. We also stick to a forward simulation perspective throughout. It turns out to be convenient to work in reverse alphabetical order.

3.1 Z

Since Z itself [7] is simply a formal mathematical language, one cannot speak definitively of *the* Z refinement. We target our remarks on the formulations in [8, 9].

Formal language: Z uses the well known schema calculus, in which a schema consists of named and typed components which are constrained by a formula built up using the usual logical primitives. This is an all-purpose machinery; 'delta' schemas enable before-after relations that specify transitions to be defined; other schemas define retrieve relations, etc. The schema calculus itself enables schemas to be combined so as to express statements such as the POs of a given refinement theory.

Granularity and naming: Most of the refinement formulations in [8, 9] stick to a $(1,1)$ framework. Purely theoretical discussions often strengthen this to identity on 'indexes' (i.e. names) of operations at abstract and concrete levels, though there is no insistence on such a close tieup in [10, 11].

Concrete-abstract fidelity: In the above context for Z refinement, the fidelity PO comes out as follows, which refers to the contract interpretation without I/O (while the behavioural interpretation drops the 'pre *AOp*'):

$$\forall\, AState;\ CState;\ CState' \bullet \mathrm{pre}\, AOp \wedge R \wedge COp \Rightarrow \exists\, AState' \bullet R' \wedge AOp \quad (1)$$

where $AState, CState$ are (abstract and concrete) state schemas (primes denote after-states), AOp, COp are corresponding operations, R is the retrieve relation, and 'pre *AOp*', the precondition, in fact denotes the domain of *AOp*.

Notions of correctness: In Z, an induction on execution steps is used in the $(1,1)$

framework to derive trace inclusion. To work smoothly, totality (on the state space) of the relations expressing operations is assumed. To cope with partial operations, a $\perp$ element is added to the state space, and *totalisations* of one kind or another, of the relations representing the operations, are applied. The consequences of totalisation (such as (1)), got by eliminating mention of the added parts from a standard forward simulation implication, constitute the POs of, and embody the notion of correctness for, the totalisation technique under consideration. These turn out to be the same for both contract and behavioural approaches, aside from the difference in (1) noted above.

Interpretation: The two main totalisations used, express the *contract* and *behavioural* interpretations. In the former, an operation may be invoked at any time, and *the consequences of calling it outside its precondition are unpredictable* (within the limits of the model of the syntax being used), including $\perp$, nontermination. In the latter, $\perp$ is guaranteed outside the precondition (usually called the guard in this context, but still defined as the domain of the relevant partial relation), which is typically interpreted by saying the operation *will not execute* if the guard is false.

Trace inclusion: Trace inclusion has been cited as the underlying derivation technique for the POs, and since an inductive approach is used, it is strong trace inclusion. However, the 'fictitious' transitions of operations introduced by totalisation are treated on an equal footing to the original 'honest' ones, so many spurious traces, not corresponding to real world behaviour, can be generated. For instance a simulation of a concrete trace may hit a state (whether abstract or concrete) that is outside the 'natural' domain of the *next* partial operation. Then, in the contract interpretation, the trace can continue in a very unrestricted manner, despite the different way that one would view the constituent steps from a real world perspective. Things look a bit better in the behavioural interpretation, since such a trace is thereafter confined to $\perp$.

Composition: One prominent composition mechanism to be found in Z is *promotion*. In promotion, a component which is specified in a self-contained way is replicated via an indexing function to form a family inside a larger system; this interacts cleanly with refinement [8, 9]. However, the schema calculus in general is not monotonic with respect to refinement without additional caveats [12].

3.2 B

The original B Method was described with great clarity in [13], and there are a number of textbook treatments eg. [14–16].

Formal language: Original B was based on predicates for subsets of states, written in a conventional first order language, and on weakest precondition predicate transformers (wppts) for the operations. The use of predicate transformers obviates the need for explicitly adjoining $\perp$ elements to the state spaces.

Granularity and naming: Original B adheres to a strict $(1, 1)$ framework; 'strict' in the sense that tools for original B demand identical names for operations and their refinements. Abstract models of complex operations can be assembled out of smaller pieces using such mechanisms as INCLUDES, USES, SEES. However once the complete abstract model has been assembled, refinement proceeds monolithically towards code. The last step of refinement to code, is accomplished by a code generator which plugs together suitably designed modules that implement the lowest level B constructs.

Concrete-abstract fidelity: This is handled via the predicate transformers. Adapting the notation of [13] for ease of comparison with (1), the relevant PO can be written:

$$AInv \wedge CInv \wedge \mathsf{trm}\ AOp \Rightarrow [\ COp\ ] \neg [\ AOp\ ] \neg CInv \tag{2}$$

In this, *AInv* and $\mathsf{trm}\ AOp$ are the abstract invariant and termination condition (the latter being the predicate of the precondition), while *CInv* is the concrete invariant, which in original B, involves both abstract and concrete variables and thus acts also as a retrieve relation; all of these are predicates. $[\ AOp\ ]$ and $[\ COp\ ]$ are the wppts for the abstract and concrete operations, so (2) says that applying the concrete and 'doubly negated' abstract wppts to the after-state retrieve relation yields a predicate (on the before-states) that is implied by the before-state quantities to the left of the implication.

Notions of correctness: In original B, precondition (trm) and guard (fis) are distinct concepts (unlike Z), albeit connected by the implication $\neg\ \mathsf{trm} \Rightarrow \mathsf{fis}$, due to the details of the axiomatic way that these two concepts are defined. Moreover, $\mathsf{trm} \wedge \neg\ \mathsf{fis}$ can hold for an operation, permitting *miracles*, a phenomenon absent from formalisms defined in a purely relational manner. In original B, trm is a conjunct of any operation's definition, so outside trm, nothing is assumed, and when interpreted relationally, it leads to something like a 'totalisation' (though different from the Z ones). During refinement, the precondition is weakened and the guard is strengthened, the former of which superficially sounds similar to Z, though it is again different technically.

Interpretation: The interpretation of operation steps for which trm and fis both hold is the conventional unproblematic one. Other steps fire the imagination. If trm is false the step *aborts*, i.e. it can start, but not complete normally; modelled relationally by an unconstrained outcome, a bit like contract Z. If fis is false the step does not start normally, but can complete; a miracle indeed, usually interpreted by saying that the step will not take place if fis is false.

Trace inclusion: In original B, trace inclusion is not addressed directly, but as a consequence of monotonicity. Refinement is monotonic across the B constructors, including sequential composition. This yields a notion of weak trace inclusion, since the trm and fis of a composition are an *output* of a composition calculation, not an input, and in particular, cannot be assumed to be the trm and fis of the first component, as one would want if one were extending a simulation by considering the next step. And even though the sufficient condition for fidelity (2) is a strengthening of the natural B refinement condition, it does not lead to an unproblematic strong trace inclusion, since in a relational model, we have the additional transitions generated by the 'totalisation', and miracles do not give rise to actual transitions.

Composition: In original B, the interaction of refinement and composition is not a real issue. The earlier INCLUDES, USES, SEES mechanisms are certainly composition mechanisms, but they just act at the top level. Only the finally assembled complete abstract model is refined, avoiding the possibility of Z-like nonmonotonicity problems. The IMPORTS mechanism allows the combination of independent developments.

3.3 Event-B

Event-B [17–19] emerged as a focusing of original B onto a subset that allows for both more convenient practical development, and also an avoidance of the more counterintuitive aspects of the original B formalism, such as miracles.

Formal language: Event-B is rooted in a traditional relational framework, derived by restricting original B operations (henceforth called events) to have a trm which is true, and controlling event availability purely via the guard, which is the domain of the event transition relation, as in Z. Distinguishing between guard and event in the syntax enables event transitions to be defined via convenient notations (such as assignment) which are more widely defined than the desired guard. Formally, the more exotic possibilities afforded by predicate transformers are no longer needed.

Granularity and naming: Event-B relaxes the strict $(1,1)$ conventions of original B. As in original B, the syntax of the refinement mechanism is embedded in the syntax of the refining machine, so an abstraction can be refined in more than one way, but not *vice versa*. However, a refining event now names its abstract event, so an abstract event can have several refinements within the same refining machine. New events in a refining machine are *implicitly* understood to refine an abstract skip, something which needed to be stated explicitly in original B, cluttering incremental development.

Concrete-abstract fidelity: The absence of the more exotic aspects of predicate transformers gives the Event-B fidelity PO a quite conventional appearance:

$$\forall\, u, v, v' \bullet AInv \wedge CInv \wedge G_{CEv} \wedge CEv \Rightarrow \exists\, u' \bullet AEv \wedge CInv' \tag{3}$$

This says that assuming the abstract invariant and the concrete invariant (which is again a joint invariant i.e. retrieve relation) and the concrete guard and concrete transition relation for the before-states, yields the existence of an abstract event which re-establishes the joint invariant in the after-states.

Notions of correctness: The absence of preconditions distinct from guards simplifies matters considerably. The previous 'weakening of the precondition' during refinement of an operation, is now taken over by 'disjunction of concrete guard with guards of all new events is weaker than the abstract guard'. This is a quite different criterion, which nevertheless guarantees that if something can happen at the abstract level, a 'suitable' thing is enabled at the concrete level. This is also combined with guard strengthening in the refinement of individual events, and a well foundedness property to prevent new events from being always enabled relative to old events. Totalisations are no longer present in any form, which has an impact on trace inclusion (see below).

Interpretation: The absence of preconditions distinct from guards simplifies interpretational matters considerably. There is a firm commitment to the idea that events which are not enabled do not execute, avoiding the need to engage with miracles and with spurious transitions generated by totalisation.

Trace inclusion: In the Event-B context, trace inclusion wins massively. Since for a refined event, the concrete guard implies the abstract one, the implication has the same orientation as the implication in (3), so the two work in harmony to enable any concrete step joined to an appropriate abstract before-state, to be unproblematically simulated, a phenomenon not present in formalisms mentioned earlier — simulated moreover, by a 'real' abstract event, not a fictitious one introduced via totalisation. New events do not disturb this, since they are by definition refinements of skip, which can always effortlessly simulate them. So we have genuine, uncluttered, strong trace inclusion.

Composition: Event-B takes a more pro-active approach to composition than original B. Event-B's top-down and incremental approach means that system models start out small and steadily get bigger. This allows composition to be instituted via *decompo-*

sition. As a system model starts to get big, its events can be partitioned into subsystems, each of which contains *abstractions* of the events not present. These abstractions can capture how events in different subsystems need to interact, allowing for independent refinement, and avoiding the non-monotonicity problems mentioned earlier.

3.4 ASM

The Abstract State Machine approach developed in a desire to create an operationally based rigorous development framework at the highest level of abstraction possible. A definitive account is given in [6].

Formal language: Among all the methodologies we survey, ASM is the one that de-emphasises the formality of the language used for modelling the most — in a laudable desire to not dissuade users by forcing them to digest a large amount of technical detail at the outset. System states are general first order structures. These get updated by applying ASM rules, which modify the FO structures held in one or more *locations*. In a departure from the other formalisms reviewed, *all* rules with a true guard are applied simultaneously during an update.

Granularity and naming: The ASM approach tries as hard as it can to break the shackles of imposing, up front, any particular scheme of correspondence between abstract and concrete steps during refinement. Since a retrieve relation has to be periodically re-established, a practical technique that breaks a pair of simulating runs into (m,n) diagrams of m abstract steps and n concrete ones (for arbitrary finite $m+n>0$), without any preconceptions about which steps occur, is minimally demanding.

Concrete-abstract fidelity: In striving to be as unrestrictive as possible, ASM does not prescribe specific low level formats for establishing refinement. However, one technique, generalised forward simulation, established by Schellhorn [20] (see also [21]), has become identified as a *de facto* standard for ASM refinement. This demands that the (m,n) diagrams mentioned above are shown to be simulating by having a 'working' retrieve relation $\approx$, which implies the actual retrieve relation $\equiv$, which itself is referred to as an *equivalence*. The $\approx$ relation is then used in implications of the form (1)-(3), except that several abstract or concrete steps (or none) can be involved at a time. As many (m,n) diagram simulations as needed to guarantee coverage of all cases that arise must then be established.

Notions of correctness: It has already been mentioned that $\equiv$ is referred to as an equivalence. While almost all retrieve relations used in practice are in fact partial or total equivalences [22], knowing this *a priori* has some useful consequences. It leads to a simple relationship between the guards of the run fragments in simulating (m,n) diagrams, subsuming guard strengthening, and eliminating many potential complications. Refinement is defined directly via a trace-inclusion-like criterion (periodic re-establishment of $\equiv$), and for $(0,n)$ and $(m,0)$ diagrams, there is a well foundedness property to prevent permanent lack of progress in one or other system in a refinement. The analogue of 'precondition weakening' (though we emphasise that there is no separate notion of precondition in ASM) is subsumed by the notion of 'complete refinement' which demands that the abstract model refines the concrete one (as well as *vice versa*), thus ensuring that any time an abstract run is available, so is a suitable concrete one, yielding persistence of coverage of requirements down a refinement chain. Of course

not all refinements need to be complete, permitting convenient underspecification at higher levels, in a similar manner to Event-B.

Interpretation: Since states and transitions are defined directly, there are no subtle issues of interpretation associated with them. Also, ASM rule firing is a hardwiring of the 'transitions which are not enabled do not execute' convention into the formalism.

Trace inclusion: The (m, n) diagram strategy of ASM modifies the notion of trace inclusion that one can sustain. The ASM (m, n) notion, at the heart of the ASM *correct refinement* criterion, can be viewed as a generalisation of the Event-B $(1, 1)$ strategy.

Composition: With the major focus being on identifying the ground model, and on its subsequent refinement (rather as in original B), the composition of independent refinements is not prominent in [6, 21]. On the other hand, if $\equiv$ *really is* an equivalence (or as we would need to have it between two state spaces which are different, a *regular* relation a.k.a. a *difunctional* relation), there is a beneficial effect on any prospective composition of refinements. Many of the issues noted in [12] arise, because incompatible criteria about abstract sets (of states, say) which are unproblematic due to the abstract sets' disjointness, can become problematic due eg. to precondition weakening when the sets' concrete retrieve images become non-disjoint via a non-regular retrieve relation. A regular retrieve relation does much to prevent this, facilitating composition of refinements.

4 Configurable Semantics

The preceding sections very briefly surveyed a few well known refinement paradigms. Although it might not be as apparent as when one examines more of the details in each case, it is easy to be struck by how so many of the issues we have highlighted, turn out merely to be *design decisions* that happen to have been taken, about some particular feature, in the context of one or other formalism. Although some such design decisions are interrelated, one can very easily imagine, that in many cases, a given design decision about some aspect of a refinement methodology, could just as easily have been implemented in the context of a methodology different from the one in which we happen to find it. Here are a few examples.

- Regarding Z, one could easily imagine its notion(s) of correctness being substituted by the ones from Event-B or ASM. Its notion of trace inclusion would then be replaced by one not requiring the use of 'fictitious' transitions generated by totalisation.
- For B, one could easily imagine adding $\perp$ elements to state spaces etc. in order to obtain a different relational semantics, with fewer 'fictitious' transitions.
- For Event-B and ASM one could imagine bringing in some aspects of the Z modelling, though it appears that little would be gained by doing so.

Of course such ideas are not new. In many cases, for mature methodologies, alternatives of one kind or another have been investigated, whether in the normal research literature or as student research projects — making an even moderately comprehensive list of the cases covered would swell the size of this paper unacceptably.

Semantic modifications of the kind hinted at can serve a more serious purpose than mere curiosity. In ProB [23], a model checker and animator for the B-Method first implemented for original B, the original B preconditions are re-interpreted as (i.e. given

the semantics of) additional guards. The reason for this is that preconditions are *weakened* during refinement, whereas guards are *strengthened*. As already noted in Section 3.3, the orientation of the latter implication is the same as that in the fidelity PO, so the two collaborate in establishing trace inclusion. Precondition weakening is in conflict with this, so the ProB adaptation is necessary to ensure that the theoretical constructions at the heart of model checking remain valid.

Commenting from a real world developer's perspective, the fewer the extraneous and counterintuitive elements that a formalism contains, the more appealing it becomes for real world use. For example, if an applications sphere features operations that are intrinsically partial, then that is all that there ought to be to the matter, and consequently, the approach of totalising such operations becomes an artificial distraction, potentially even a misleading one if the fictitious transitions could be mistaken for real ones.

Such techniques as totalisation can be seen as making the task of *setting up* the semantics of a formal framework simpler. However, the real world developer's priorities are more focused on accurate modelling of the application scenario, and this can motivate a modification of the semantics, albeit at the price of additional formal complexity. In the Météor Project [24], the semantics of original B was modified to explicitly check well-definedness conditions for applications of (partial) functions, using techniques going back to Owe [25], in recognition of this application need. Event-B, a more recent development, has such checks built in *ab initio*, and its semantics fits model checking needs much better too, as already noted.

The above thoughts, assembled with the wisdom of hindsight, drive one to the conclusion that the semantics of formal development notations would be better designed in a more *flexible*, or *configurable* way. The idea that a single pre-ordained semantic framework can cover all cases in all needed application scenarios is hard to sustain.

Such a viewpoint has consequences of course, both theoretical and practical. Theoretically, one would have to structure the theory of a particular formalism so that contrasting design decisions could be adopted straightforwardly, in a way that avoided confusing the reader, and so that the consequences of adopting alternatives could easily be imagined. Moreover, doing this would not constitute a huge overhead since theoretical work is relatively cheap. Practically though, it is a different matter. Practically, formalisms, such as the ones we have discussed, are embodied in tools; and creating a good tool requires a considerable investment. We discuss the wider consequences of our perspective for tools in the next section.

A final thought on the topic of semantic flexibility. One cannot help notice from the above brief discussion, that the places where semantic modifications have been imposed on a technique in order to satisfy application development methodology needs, have all occurred in the 'notions of correctness' and 'interpretation' areas. Notably free from interference has been the 'concrete-abstract fidelity' area. This indicates a strong consensus among approaches that simulation (in one form or another) is *the* key criterion that techniques must establish. Other issues from Section 2, such as 'formal language', 'granularity and naming' and 'trace inclusion', 'composition', can be seen as either enablers for setting up a framework, or derivable consequences of the design decisions taken. This in turn suggests a scheme for organising theories in this field: one sets up the linguistic machinery, one sets up concrete-abstract simulation, one chooses

additional correctness and accompanying concepts, and then one derives whatever additional properties of interest follow from the preceding choices. And when comparing or combining one formalism with another, it is the *intersection* of features rather than their *union* that is of greatest importance.

5 Issues for Tools

The considerations of the preceding sections have implications for tool design, as already noted. Up to now, most tools in this arena have been based on a commitment to a particular set of design decisions about various semantic issues, and these decisions, howsoever arrived at, have been hardwired into the structure of the tool, making tools somewhat monolithic. This has the advantage that with each tool, one knows exactly what one is getting. However, it also has the disadvantage that it isolates tools from each other, and makes tool interoperability difficult or impossible.

These days, it is more and more recognised that to best address the risks inherent in the whole process of a system development, it is desirable to utilise a range of techniques and to interconnect them. A consequence of the isolation between tools is that it is difficult to simultaneously capitalise on the strengths of more than one. It also means that when an advance is made in one tool, other tools have to duplicate the work involved before similar ideas can be used in the other contexts. One way of addressing this difficulty is to not only make the various theoretical frameworks flexible and configurable, as recommended earlier, but to also make the tools that support them more *configurable* and *programmable*. We now discuss three approaches to this as exemplified within three different tool environments.

The **Rodin Toolset** [18] for supporting the Event-B methodology, is built on Eclipse [26], a flexible platform for software development which manages dependencies between development artifacts and supports a GUI for displaying them. The semantic content of a methodolgy supported by an Eclipse-based tool is captured via a collection of Eclipse plugins. Rodin is thus a collection of plugins for introducing Event-B machines and contexts, editing them, checking them, generating POs, supporting PO proof, and general housekeeping. Other plugins exist for LaTeX printing, ProB support, and support for additional development activities to aid Event-B development is planned or can easily be envisaged. Since the source of Rodin is in the public domain, one can integrate such additional activities by simply writing more plugins of one's own. If one wished to actually *alter* specific semantic elements of Event-B for any reason, one might well have to *replace* an existing plugin by a different one, since the standard semantics of Event-B is hardwired into the plugins, if not into Eclipse. This, although possible, is not trivial, since writing Eclipse plugins, especially ones that would have to collaborate closely with other existing ones, is not an easy task. Counter to this relative inflexibility, we note that a certain limited amount of semantic flexibility has been built into Rodin *ab initio*, since one can configure certain attributes of events, eg. whether they are *ordinary*, *convergent*, etc. This influences the verification conditions that are generated.

The **Frog tool** [27, 28] is an experimental tool, originally designed for mechanically supporting retrenchment [29], whose inbuilt flexibility addresses our concerns

very well. In Frog, much of what is hardwired in typical existing proof-driven development tools is programmable. Thus there is an intermediate language (Frog-CCL) for declaring the *structure* of the clauses that comprise the usual syntactic constructs that constitute a typical formal development framework. Paradigmatically, one has machine definitions, relationships between machines and the like. In Frog, the mathematical ingredients of all the constructs are specified using Z schemas, thus exploiting Z's essence as a general purpose formal mathematical notation. Since relationships between constructs, such as refinements, are themselves syntactic constructs, the precise nature of what constitutes a refinement (in terms of the POs that characterise it), can be precisely specified and configured using Frog-CCL scripts. Designing a complete formal development methodology in Frog is thus a matter of writing several Frog-CCL scripts, rather than a major development task. At least that is so in principle. Due to limited time during Simon Fraser's doctorate, certain things are still hardwired in Frog, such as: the use of Z as mathematical language, the use of the Isabelle theorem prover [30], and a strict $(1,1)$ naming convention for operations. Evidently, more flexibility could easily be contemplated for these aspects.

Of course the maximum flexibility for adapting the semantic and/or any other aspects of a methodology whilst still within a tool environment, is to work with a fairly general purpose theorem prover. There are essentially no constraints when one takes this approach, since, regardless of what features are taken as constituting the foundations of a given formal development methodology (and there is considerable variation on what is regarded as fundamental among different methodologies), the verification that a particular development is correct with respect to that particular methodology, always reduces to constructing proofs (of a fairly conventional kind) of a number of posited properties of the development, the verification conditions. The flexibility of the general purpose theorem prover approach has been demonstrated with great success in deploying the **KIV Theorem Prover** [31] to address system development in the ASM methodology (and others). The web site [32] gives full details of the mechanical verification of a number of substantial developments, carried out under mechanical formalisations of a variety of detailed refinement formalisms. The approach has enjoyed particular success in the context of the mechanical verification of Mondex [33, 34]. The generality of KIV enabled previously investigated refinement strategies to be quickly adapted to the details of Mondex, and the whole of the verification, done in competition with several international groups, to be accomplished in record time.

6 Conclusions

In this paper, we have examined some key features of a small number of well known refinement methodologies, and commented on their similarities and differences. We noted that many of their features were not especially specific to the methodologies in which they were found, and that we could just as easily transplant them into others. We also observed that applications considerations can influence and adapt such methodologies, irrespective of first principles, belying the view that their semantics are sacrosanct.

The same considerations impact tool support, but more deeply, given the investment needed to create a good tool. Accordingly, we turned our attention to strategies

for achieving greater tool flexibility: from Rodin's plugins, to Frog's scripting approach, to theorem proving using eg. KIV. While the last of these undoubtedly offers the greatest flexibility, it also requires the greatest expertise, and for more everyday development environments, some tool-imposed discipline is probably necessary. The question is how to achieve an adequate level of tool supervision without compromising openness, interoperability and flexibility. In the author's view, the Frog approach offers great promise for quick adaptability of the semantic details of a formal methodology, without demanding a huge investment in reprogramming the tool. It is easy to imagine that in a tool such as Frog, for industrial application, the programmable semantic aspects can be made editable only by senior personnel, and the majority of the development team see a tool which behaves as though its semantics was conventionally hardwired. In any event, all the approaches outlined above certainly offer promise, and further experimentation is to be expected in the near future.

All of the above is certainly in harmony with the call for an Evidential Tool Bus (ETB) [2], over which tools could communicate. In the ETB, tools are no longer envisaged as monolithic entities, isolated from each other, but rather as members of a community, each responsible for a subset of, or for a particular approach to, the overall verification task. Tools on the bus could make use of the (partial) evidence for correctness established by other tools on the bus, to enhance what they themselves would be able to achieve — they in turn publishing their own results on the bus for successor tools to benefit from. Thus the community could achieve, by cooperation, far more, far more cheaply, than any one tool could achieve on its own.

The preceding is also in harmony with the currently active Verification Grand Challenge [3–5]. This has many aims, from promoting formal techniques in the mainstream (on the basis of their by now well established capacity to deliver, to standard, on time, on budget, and overall more cheaply than by the use of conventional techniques), to establishing cadres of formally verified applications in a repository (as further evidence to encourage their uptake, and perhaps to provide thereby collections of reusable formally verified components), to encouraging the harmonisation and cooperation of formal techniques. This last aim is squarely aligned with our motivations for carrying out the analysis of refinement techniques given in this paper.

References

1. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. C.U.P. (1998)
2. Rushby, J.: Harnessing Disruptive Innovation in Formal Verification. In: Proc. IEEE SEFM-06. IEEE Computer Society Press, IEEE (2006) 21–28
3. Jones, C., O'Hearne, P., Woodcock, J.: Verified Software: A Grand Challenge. IEEE Computer **39**(4) (2006) 93–95
4. Woodcock, J.: First Steps in the The Verified Software Grand Challenge. IEEE Computer **39**(10) (2006) 57–64
5. Woodcock, J., Banach, R.: The Verification Grand Challenge. Communications of the Computer Scociety of India (May 2007)
6. Börger, E., Stärk, R.: Abstract State Machines. A Method for High Level System Design and Analysis. Springer (2003)

7. ISO/IEC 13568: Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics: International Standard. (2002) http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip.
8. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. PHI (1996)
9. Derrick, J., Boiten, E.: Refinement in Z and Object-Z. FACIT. Springer (2001)
10. Spivey, J.: The Z Notation: A Reference Manual. Second edn. PHI (1992)
11. Cooper, D., Stepney, S., Woodcock, J.: Derivation of Z Refinement Proof Rules. Technical Report YCS-2002-347, University of York (2002)
12. Groves, L.: Practical Data Refinement for the Z Schema Calculus. In: Proc. ZB-05. Volume 3455 of LNCS., Springer (2005) 393–413
13. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. C.U.P. (1996)
14. Lano, K., haughton, H.: Specification in B. Imperial College Press (1996)
15. Habrias, H.: Specification Formelle avec B. Hermes Sciences Publications (2001)
16. Schneider, S.: The B-Method. Palgrave (2001)
17. Abrial, J.R.: Event-B. to be published.
18. Rodin. European Project Rodin (Rigorous Open Development for Complex Systems) IST-511599 http://rodin.cs.ncl.ac.uk/.
19. The Rodin Platform. http://sourceforge.net/projects/rodin-b-sharp/.
20. Schellhorn, G.: Verification of ASM Refinements Using Generalised Forward Simulation. J.UCS **7**(11) (2001) 952–979
21. Börger, E.: The ASM Rrefinement Method. Form. Asp. Comp. **15** (2003) 237–257
22. Banach, R.: On Regularity in Software Design. Sci. Comp. Prog. **24** (1995) 221–248
23. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Proc. FM 2003. Volume 2805 of LNCS., Springer (2003) 855–874
24. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A Successful Application of B in a Large Project. In: Proc. FM 1999, LNCS 1708. (1999) 369–387
25. Owe, O.: Partial Logics Reconsidered: A Conservative Approach. F.A.C.S. **3** (1993) 1–16
26. The Eclipse Project. http://www.eclipse.org/.
27. Fraser, S., Banach, R.: Configurable Proof Obligations in the Frog Toolkit. In: Proc. IEEE SEFM-07. IEEE Computer Society Press, IEEE (2007) 361–370
28. Fraser, S.: Mechanized Support for Retrenchment. PhD thesis, School of Computer Science, University of Manchester (2008)
29. Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Engineering and Theoretical Underpinnings of Retrenchment. Sci. Comp. Prog. **67** (2007) 301–329
30. The Isabelle Theorem prover. http://www.cl.cam.ac.uk/research/hvg/Isabelle/
31. The Karlsruhe Interactive Verifier. http://i11www.iti.uni-karlsruhe.de/~kiv/KIV-KA.html
32. KIV: KIV Verifications on the Web http://www.informatik.uni-augsburg.de/swt/projects/.
33. Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In: Proc. FM 2006. Volume 4085 of LNCS., Springer (2006) 16–31
34. Mondex KIV: Web presentation of the Mondex case study in KIV http://www.informatik.uni-augsburg.de/swt/projects/mondex.html.