Fundamenta Informaticae XXXXXXXXX (2066) 101–140

IOS Press

Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1, 1) Forward Refinements

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

Michael Poppleton
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
email: mrp@ecs.soton.ac.uk

Czeslaw Jeske
School of Computer Science
University of Manchester
Manchester M13 9PL, UK
email: cj@cs.man.ac.uk

Susan Stepney
Department of Computer Science
University of York
York YO10 5DD, UK
email: susan.stepney@cs.york.ac.uk

Abstract. Some of the success stories of model based refinement are recalled, as well as some of the annoyances that arise when refinement is deployed in the engineering of large systems. The way that retrenchment attempts to alleviate such inconveniences is briefly reviewed. The Mondex Electronic Purse formal development provides a highly credible testbed for examining how real world refinement difficulties can be treated via retrenchment. The contributions of retrenchment to integrating the real implementation with the formal development are surveyed, and the extraction of commonly occurring 'retrenchment patterns' is recalled. One of the Mondex difficulties, the 'Balance Enquiry Quandary' is treated in detail, and the way that retrenchment is able to account for the system behaviour is explained. The problem is reconsidered using generalised forward refinement, and the simplicity of the resolution of the quandary, both by retrenchment, and by generalised forward refinement, inspires the creation of a genuine $(1, 1)$ forward refinement for Mondex, something long thought impossible. The forward treatment exhibits a similar balance enquiry quandary to the backward refinement, as it must, given that both are refinements of an atomic action to a non-atomic protocol, and the forward quandary is dealt with as easily by retrenchment as is the backward case. The simplicity of the retrenchment treatment foreshadows a general purpose retrenchment *Atomicity Pattern* for dealing with atomic-versus-finegrained situations.

Keywords: Retrenchment, Refinement, Verification, Mondex Purse, Atomicity

1. Introduction

Model based refinement is well known as the standard technique for progressing abstract system designs towards implementations. The abstract designs are typically expressed in a modelling language permitting the maximum of expressivity, abstraction, mathematical rigour, and succinctness, without concern for executability. The lower level models lean increasingly towards the actual capabilities of real computing devices, and the algorithms that they must utilise. There are a number of specific formulations of model based refinement, which can differ as regards particular technical details, but they all share the same overall strategy for establishing the correctness of an implementation: namely that for every run of the concrete system, there must be a run of the abstract system which maintains the desired notion of correct correspondence between them.[1] Among the more well known techniques we can mention Z [41, 51, 32], B [1, 40, 35], VDM [28, 33, 16], RAISE [36, 50] and ASM [25, 19, 37, 38].

Besides being well established in the academic sphere, refinement has had notable successes on the industrial front in recent years. We can cite the Mondex Purse [45, 46] and Multos Operating System [44, 43] for Z, the MÉTÉOR project [12] and numerous other railway system projects in France and elsewhere for B, and a number of language systems and other developments for ASM [42, 23, 21, 24, 14, 13, 20, 22, 49]. Earlier more limited success can be cited for VDM [30].

Despite these undoubted successes, refinement practitioners have known for some time that when refinement is used as the sole means of progressing from an abstract model to a concrete one, then certain difficulties can plague the development process due to the exacting nature of typical refinement proof obligations. This is not a technical difficulty with refinement, rather it is a manifestation of human inclination to view certain things as abstractions/concretisations of the same phenomenon, that some given refinement formalism does not permit to be so viewed. Since the human notion of abstraction is inevitably imprecise, and the mathematical notion of abstraction pertaining to any specific refinement formalism is *de facto* extremely precise, some dislocation between the two is bound to occur sometimes.

Usually, if the scale of the problem is small, this dislocation can be overcome easily enough. Frequently it is sufficient to make some small adjustment to one or other of an abstract/concrete pair of models to bring them into line. Often it is the abstract model that must be changed, so that it does not make impossible demands of some aspect of the implementation (assuming the concrete model is intended to not misrepresent what the hardware is doing at the code level). However, when the problem size is large, such manipulations can become impractical.

Let us illustrate this on a simple and commonly occurring example: natural number arithmetic. Implementable whole numbers are invariably bounded. So arithmetic always generates within-bounds and out-of-bounds cases. If there are n different quantities in the model, then for a typical operation there will be one all-within-bounds case, and easily of the order of $2^n - 1$ out-of-bounds cases[2] of various flavours. If one is fastidious about representing correctly what happens in all these different cases in the specification, their syntactic descriptions can easily swamp that of the single all-within-bounds case, which is the one of most interest. This in turn makes it highly desirable to idealise the arithmetic and use unbounded naturals at the abstract level. Unfortunately in the overwhelming majority of refinement formalisms there is no refinement from unbounded naturals to bounded ones that handles a sensible selection of the operations that are normally needed. One is therefore faced with a choice. One takes on board the proliferation of cases at the abstract level with its concomitant syntactic overload, or some

[1] Usually this is augmented with an applicability condition, to exclude trivial implementations.
[2] Or more, depending on the details of the required operation.

aspect of the refinement falls short of what refinement is supposed to be.

While the above illustrates some difficulties that are intrinsic to the way refinement techniques are tied to very specific details of the models involved, there are others. Applications that are large enough, are seldom tackled on a whim or purely for the research literature. The costs involved mean that there is usually a predetermined commercial goal to be satisfied. As a result, there will be many stakeholders engaged in the enterprise other than refinement specialists, and these people are prone to seeing the development, and the models that comprise it, in a very different light to that of the formalists. We point up some tangible consequences of this as regards the feasibility of altering the specification in response to some technical infelicity:

- The customer *may not permit* a change in the specification (in order to make it refinable to the lower level models). The specification document serves several different purposes, including certification and validation, as well as being used as a starting point for refinement towards an implementation. Often it would be more costly, overall, to change the specification, than to get the formal specialists to work around some perceived imperfection in the refinement.

- Textbook or research examples and methods typically start from a blank sheet of paper. Real engineering applications *never* do so. There are invariably given fixed points in the development, making a completely purist approach unrealistic.

- There might simply be *no time* to change the specification. If some problem shows up during the proof of the last lemma, whose resolution requires the reworking of aspects of the specification, there may simply not be enough time to do the detailed reworking, within the schedules, deadlines and budgets that obtain. In a pure research environment the job would simply be considered unfinished and publication would be delayed: the researchers' livelihoods would most likely not be immediately imperiled. In an engineering environment, the job must be completed somehow despite the circumstances: in the face of noncompletion the engineers' livelihoods could be imperiled quite easily, and engineering compromises become necessary.

- Changes to specifications arising from 'low level failures' of refinement can depend on the precise design route chosen. If a single specification is targeted at multiple platforms, such low-level-originating changes that impact the specification can easily be *incompatible*. (Think of natural numbers targeted at hardware having different intrinsic integer bounds.) In such cases there may be no single abstraction necessarily incorporating some low level features that caters adequately for all the platforms.

- Specifications not only serve to initiate refinements, but also serve as an important means of *communication* between the various parties in a development. A refinement structure may not always organise the system's requirements in a way that makes sense to domain experts; refinement is essentially a process of conservative extension, and if this structure is at odds with domain experts' approaches, then it will not communicate as effectively as it should.

For reasons such as the above, the stakeholders in the development other than refinement specialists *may simply not agree* to changes in the models as suggested by the formalists, regardless of the latters' protestations. Thus the human aspects of the development milieu become paramount. This is nothing more than a corollary of the fact that the construction of large systems is an engineering problem, and not

purely a problem in formal system construction. The key desiderata in the two domains are just different. In its project management aspects, the Mondex development [46] —which is the focus of this paper— exhibited most of the phenomena mentioned above, quite aside from the more technical issues regarding the refinement proof which we discuss below.

Retrenchment [4, 5, 6] was introduced in order to be able to address the difficulties caused when what humans need the notion of 'abstraction' to do, is in conflict with what 'abstraction' as supplied by some notion of model based refinement can supply.

Retrenchment proceeds by inverting the usual trajectory from broad principles to proof obligations found in refinement. In refinement, the starting point is a notion of correct correspondence between abstract and concrete models, from which proof obligations are derived. In retrenchment by contrast, the proof obligations are manipulated so that they can encompass situations such as those discussed above, and whatever broad principles can be derived therefrom, are sought. Certainly the typical guarantees offered by refinement are forfeit.

We illustrate this manipulation by taking a paradigmatic forward refinement correctness proof obligation (c.f. (6) below) and turning it into a retrenchment proof obligation. For the former we take:

$$\begin{aligned} R(u,v) \land RIn_{Op}(i,j) \land \mathrm{pre}(Op_a)(u,i) \land Op_b(v,j,v',p) \Rightarrow \\ (\exists\, u', o \bullet Op_a(u,i,u',o) \land R(u',v') \land ROut_{Op}(o,p)) \end{aligned}$$

In the above u, v are (Abstract/Concrete) states (primed for after-states), i, j are (Abstract/Concrete) inputs, o, p are (Abstract/Concrete) outputs and Op_a and Op_b are (Abstract/Concrete) versions of the operation Op. R is the retrieve relation, while $RIn_{Op}, ROut_{Op}$ are input/output relations respectively. Especially when strengthened by suitable assumptions about $R, RIn_{Op}, ROut_{Op}$, the above is equivalent to typical operation POs in the literature. To turn it into a retrenchment PO (c.f. (8)), we modify it to:

$$\begin{aligned} R(u,v) \land W_{Op}(i,j,u,v) \land Op_b(v,j,v',p) \Rightarrow \\ (\exists\, u', o \bullet Op_a(u,i,u',o) \land ((R(u',v') \land O_{Op}(o,p;\ u',v',i,j,u,v)) \lor \\ C_{Op}(u',v',o,p;\ i,j,u,v))) \end{aligned}$$

Now, $RIn_{Op} \land \mathrm{pre}(Op_a)$ has been generalised to the within relation $W_{Op}(i,j,u,v)$ which is an arbitrary relation in the before-values; $ROut_{Op}$ has been generalised to the retrenchment output relation $O_{Op}(o,p;\ u',v',i,j,u,v)$ which now allows all the variables to occur; and, most importantly, there is a concedes relation $C_{Op}(u',v',o,p;\ i,j,u,v)$ to describe what happens if the retrieve relation cannot be re-established by a given pair of (Abstract/Concrete) steps.[3] From such a starting point, one derives what broad principles one is able to.

The flexibility introduced into formal development by retrenchment lends itself to many uses. At one extreme, it can be restricted to the kind of situation outlined above, namely handling irritating restrictions forced onto the development by the finiteness or other limitations of implementable data types [4]. At the other extreme, it can be used to capture very general evolution of system definitions, as new considerations impact preliminary models on the route to the final system description [7]. How far along this scale of possibilities one happens to be, depends on one's perspective about a particular change in system description. The same change in the system might be viewed by one person as a system evolution, and

[3]The semicolons in O_{Op}, C_{Op} are purely cosmetic, separating the variables of 'most interest' from others, which are permitted, but less often needed.

thus as residing firmly in the requirements engineering arena, while for another person, it could be very much tied up with the road to an implementation, and thus be viewed as a development step; much can depend on whether the individual is focused on user needs, or technology capabilities.

It is important to see that, from a purely engineering vantage point, there is no hard and fast boundary between these two activities: requirements evolution can blend smoothly into development and implementation. This is in contrast to the formal refinement vantage point in which the boundary is clear: anything that cannot be captured by a refinement must be a requirements evolution step. Amongst other things, retrenchment can build a dialogue between these two perspectives. In effect, this has been demonstrated for Mondex in [8, 9, 10], in which a collection of horizontal retrenchment 'rungs' connects two vertical refinement 'columns' of models, the two columns forming separate developments of 'the same' requirements, with each column incorporating differing and incompatible (from a refinement vantage point) levels of real world detail. The entire aggregation is nominated the *Tower Pattern*, and is a commonly occurring schema for the deployment of retrenchment, largely independent of the nature of the issue being captured by retrenchment.

The above extended discussion has indicated that retrenchment can usefully address many issues that arise in bridging the gap between the world of ideal refinements and real developments. However that is not its only virtue. The flexibility that retrenchment brings to formal development can act as a spur for progress outside this limited sphere. In the present paper, the main concern is the Balance Enquiry Quandary in Mondex. Once the precise nature of the main problem has been appreciated, the solution via retrenchment turns out to be remarkably simple. This very simplicity, a whisker away from a refinement, acts as a stimulus to investigate alternative refinement theoretic approaches to Mondex. Thus the latter parts of the paper explore for Mondex, generalised forward refinements, and $(1,1)$ forward refinements respectively (the original Mondex refinement was $(1,1)$ backward). While the first of these overcomes the quandary in the manner one would expect, the $(1,1)$ forward refinement would most likely not have been discovered had the ground not been prepared in a suitably tempting manner by the simplicity of the retrenchment theoretic treatment. The combination of different refinement techniques, focused on the same problem, illuminates their various pros and cons in a particularly useful manner.

The remainder of the paper is as follows. Section 2 briefly recalls the Mondex refinement, and highlights the 'retrenchment opportunities' it offers, i.e. those places in the development where the desire to establish a refinement meant that certain requirements issues were omitted. Section 3 reviews the technical details of refinements and retrenchments we need later. Section 4 reviews the Mondex A and B models in an appropriately simplified form, discusses the refinement between them, and gives a precise explanation of the Balance Enquiry Quandary. Section 5 shows how retrenchment can deal with the situation relatively simply. Section 6 addresses the same situation using generalised forward refinement and shows how it overcomes the problem; the pros and cons of the generalised forward refinement approach are discussed briefly. Inspired by the simplicity of the preceding treatment, Section 7 discusses the issue of resolution of nondeterminism with more care, outlining a way in which a forward refinement might be accomplished. A $(1,1)$ forward refinement is then given, something long thought impossible for Mondex. The forward approach also exhibits a Balance Enquiry Quandary similar to the backward one, and this is resolved using retrenchment in an essentially identical manner to the original version. Section 8 concludes, and looks forward to a general purpose retrenchment *Atomicity Pattern* for dealing with atomic-versus-finegrained situations.

In order to keep the size of this paper reasonable, proofs are presented in outline form. Enough background is supplied that readers will be able to fill in the missing details without difficulty.

2. The Mondex Refinement and its Retrenchment Opportunities

The Mondex Electronic Purse [46] is, as its name suggests, an electronic implementation of a container for money. A purse can pay out money to another purse, decreasing its own balance, and can accept payments from another purse, increasing its own balance. This transfer is implemented by the value transfer protocol, a sequence of messages passed between the purses. The security implications are obvious. If the mechanisms for purses' financial transactions are in any way vulnerable, the financial institution underwriting the purses' funds could be seriously impacted. For this reason, the developers of Mondex (formerly a part of NatWest Bank), employed state of the art methods to ensure the implementation was as robust as possible. At the time of its creation (the mid 1990's), the Mondex Purse achieved an ITSEC [2] rating of 6, the highest ITSEC level ever reached up to that point, and equivalent, in terms of contemporary standards, to a Common Criteria EAL7 rating [31].

ITSEC E6 requires a formal abstract model, a formal concrete model, and proof of correspondence between them. In the Mondex case, this proof of correspondence was a refinement proof, discharged by hand. The Mondex Purse remains an impressive achievement, and its development was a trailblazer for showing that fully formal techniques could be applied within realistic time and cost limitations on industrial scale applications.

Since Mondex, the JavaCard [11] has enjoyed an even more exacting development, in which the formal refinement proof was checked by machine, rather than by mere humans. More recently, the authors of [39] have succeeded in providing a machine-checked proof of the reduced version of Mondex as published in [46]. They have further simplified the specification somewhat (which was possible since they were not labouring under the E6 requirements of the original full development down to implementation level), and found some minor omissions in the hand proof of [46] (similar in severity to those found when mechanising another hand proof [48]). The whole job took around a month according to [39], which is a relatively modest cost.[4] The ability to do such a proof now in a fully machine-checked manner, and relatively economically, demonstrates the impressive improvement in proof technology in the decade since the original Mondex proof was performed. It further increases confidence in the results, and reduces the effort needed to produce such verified systems. However, it must be remembered that the majority of the effort of the original Mondex development never went into performing the proof itself, but into formulating the specification, the invariants, the security properties, and the proof obligations.

The Mondex development, as described in [46] consists of three models called model A, model B, model C, and refinements from model A to model B, and from model B to model C. Model A is a highly abstract expression of atomic value transfer between purses, allowing for an abstract (i.e. atomic) notion of loss in transit. It is a model targeted purely at the principal security properties of the entire purse system, which are 'No value created' (i.e. in the worst case, the total value in the entire community of purses can reduce but not increase), and 'All value accounted' (i.e. even if some value is lost in transit, it is known to be so). In particular, it is to be noted that model A does not capture all the many other system requirements of the complete development. Thus one cannot strictly call it a *specification* of the whole system. Model B captures the key elements of the distributed value transfer protocol, and is thus non-atomic. As well as containing the value transfer protocol operations, model B is strengthened by a number of invariants. These invariants, while provable by induction on the length of the execution sequence (assuming at least that the cryptographically protected messages of the protocol remain un-

[4] However it should be noted that the authors of [39] brought to the task extensive prior experience of formalising formulations of refinement, and of formalising significant case studies within such formalised refinement notions.

forgeable), need nevertheless to be *assumed*, because the refinement from model A to model B in [46] is a backward refinement. The backward refinement reconciles the *early* resolution of the nondeterminism between successful value transfer and loss in transit in model A, with its *late* resolution in model B. The job of establishing the invariants by induction is relegated to the (forward) refinement from model B to model C. It is thus shown that model C is a refinement of model A.

While admiring the success of the model A to model C refinement, it is important to realise that the choice of which aspects of the development went into any of the A, B, C models and which were left out, was influenced not only by needs/requirements considerations, but also by whether or not a refinement could indeed be established between adjacent models if one chose to treat a particular topic in a particular way. The overriding desire for a refinement meant that a number of aspects of the system, in principle deserving to be included within the formal development, were nevertheless omitted from it. This is acceptable under the ITSEC E6 requirements, which do not require fully formal development all the way to code, and these formal omissions were properly handled in the later semi-formal steps.

One of the main motivations for retrenchment is the desire to be able to connect such issues more closely, i.e. formally, with the fully formal development. As a consequence, the tension that arises about whether some feature should be included or not in the refinement-based development, is eased, since versions with and without the feature may be formally related via a retrenchment and the development paths with and without the feature may be thereby drawn together.

In the case of Mondex, the semi-formal compromises mentioned above give rise to a number of 'retrenchment opportunities', allowing a more complete treatment of the topic in question to be given. Here is a brief summary of them.

1. Sequence Number: The integrity of the protocol depends partly on the sequence number of the transaction in progress. Sequence numbers occur in the Mondex B and C concrete models where they are naturals; in reality they are bounded, but large.

2. Log Full: Transfers completing abnormally are aborted and logged locally by purses. The relationship between local purse logs and the 'All value accounted' security property is rather complicated, as will become abundantly clear below. Suffice it to say here that purses' log contents are essential for this property. Logs occur in the B, C models where they are unbounded; in reality they are finite, and small.

3. Hash Function: The permanent record of the 'lost value' component of the abstract model is, in the concrete B and C models, an off-card archive into which purses' log contents are saved. A purse needs to be assured that the data is safely in the archive before it can clear it from its own, highly constrained, log memory. Safe archival is signalled to the purse using a 'clear' code. The purse log contents are assumed to be in total injective correspondence with the clear codes, as that property is required in the proof of correctness. In reality of course a cryptographic hash function is used, which is not injective, but is informally argued to be 'sufficiently injective'.

4. Balance Enquiry: Each purse has a balance enquiry operation. If this is invoked at a particular point in the middle of a concrete model value transfer, a temporary discrepancy can occur between the reported abstract and concrete balances due to differences in where nondeterminism is resolved in the two models. This is handled formally by a modelling trick, using finalisation instead of the enquiry operation to observe the state, and to confirm that nothing is in fact amiss.

Let us sketch what retrenchment can contribute regarding these various topics.

1. As regards the finiteness of sequence numbers, this is like many simple retrenchment examples in the research literature. It is not hard to write down a model which is like the Mondex concrete model except that the sequence numbers are bounded, and then to draw up a provable retrenchment between them. The greater interest lies in integrating this development step with the rest of the Mondex development. It turns out that the detail introduced in the new model can be lifted to the level of abstraction of the higher models of the development, and moreover, if one is particularly careful about how the new model has been constructed, it can turn out to be a refinement of the abstract one, though this is by no means a robust property.[5] See [8]. A byproduct of the retrenchment formulation is that it allows validation of the adopted sequence number limit to focus on an ingredient of the formal models (the relevant concession), which would not be possible in a purely refinement based treatment.

2. As regards the finiteness of the purse logs, this has many aspects that resemble the previous case (although sequence numbers always increase monotonically, whereas purse logs can be cleared when they get full). Again it is easy enough to construct a model that has a finite log, and to pursue a strategy that lifts the detail to the higher level models of the Mondex development. See [9]. The differences from the previous case centre on validation aspects. For sequence numbers, the aim is to analyse the concession in order to choose a limit that will never be encountered in a purse's lifetime. For the log, since the actual log size is fixed at around 5, the preceding aim is unrealistic, and validation focuses on confirming that once the log is full, no new transactions are initiated, since the security invariants depend on failing transactions being appropriately logged. This amounts to a different treatment of the models, and of the incompatibility between them captured in the concessions that connect them.

Note that both of the above cases feature a 'finite limit' phenomenon. A complete lower level model will contain both phenomena (as well as other things) and the description of each operation will therefore break up into (at least) four cases depending on whether the sequence number and/or purse log are still within bounds. This is already an example of the $2^n - 1$ case proliferation noted above. It becomes clear that relegating the concerns regarding these details to a lower level model, as permitted by the retrenchment approach, leaving earlier models to concentrate on core functionality, is very worthwhile.

3. As regards the non-injectivity of the hash function, it is evident that one can write a model in which the abstract total injective function from purses' log contents to the clear codes is replaced by a hash function which is less than injective. The concession of the retrenchment between these models refers to the loss of the 'All value accounted' global security property. The property states that even in the face of failing transactions, sufficient records are maintained across the system that all the original funds in the system can be properly accounted for. The validation of this scenario focuses on the statistical likelihood that the concession might be made valid, i.e. that a purse receives in error a clear-log message with just the right properties to make it believable. See [10].

The preceding retrenchment opportunities are all 'localised' in that each of the discrepancies discussed could be viewed as being rooted in a single operation at a time.[6] This is the characteristic feature of situations in which the *Tower Pattern* [8] is applicable. On one side of the tower we have a refinement development which is free of the messy details and thus more perspicuous (but which is, strictly speak-

[5]The latter hinges on the fact that in Mondex, incrementing the sequence number happens as the first step of a transaction. Since transactions are allowed to fail in both abstract and concrete models, failure to increment can be amalgamated with other kinds of failure in a carefully constructed model.

[6]For example, even though sequence numbers are incremented within several purse operations, each such operation could be individually retrenched, without needing to refer to the others.

ing, unimplementable). On the other side, we have a refinement development weighed down by a proper account of these details, and considerably less transparent as a result (but which is more implementable). In between, and connecting them, there is the requisite collection of retrenchment rungs, describing how corresponding levels of the two refinements are related.

4. The fourth retrenchment opportunity and the focus of this paper, the Balance Enquiry Quandary, is more complicated, necessitating the consideration of sequences of operations. The lead-in is as follows.

The purses' environment is assumed hostile. Specifically, each purse is in effect on its own and makes no assumptions about the environment's inclination to act in any particular manner. Therefore each purse operation must alone preserve the system's security invariants: 'No value created' and 'All value accounted'. The most straightforward way of achieving this is to have each concrete purse operation refine some abstract one. Since the abstract level transfer is atomic whereas the concrete level one is not, various concrete operations must be refinements of what are in effect null abstract operations. This would appear to offer the chance of matching the nondeterminism resolution points in models A and B, but the consequence of this would be to cause a mismatch in the nondeterminism *introduction* points, caused by the non-atomicity of the concrete protocol. In fact, in [46], the nondeterminism of the value transfer protocol is resolved early in model A, and late in model B. Now, inserting a balance enquiry into the middle of a transfer[7] can reveal the temporary difference between abstract and concrete balances caused by the differing resolution points at the two levels. This is the Balance Enquiry Quandary. Of course it is entirely innocuous. One of the main aims of this paper is to show how this can be smoothly handled via retrenchment.

3. Refinements and Retrenchments, Forwards and Backwards

In this section we briefly review the notions of refinement and retrenchment that we use. Since the Mondex formal development was done entirely in Z, we remain with that notation for the rest of the paper. For refinement, we mostly follow the formulation in [27], a liberalisation of the rules in [41], as this was what was used in the Mondex development. This formulation of Z refinement uses the 'contract' or 'non-blocking' interpretation of applicability, in contrast to the 'blocking' or 'behavioural' interpretation; see eg. [29] for a comparison. The system nomenclature in our definitions will be in line with that needed for the various models in our discussion of Mondex below.

We cast all our notions in terms of the A and B models, since they are the only ones that occur here. We assume an abstract model given by the ADT $(A, AInit, \{AOp, AI_{Op}, AO_{Op} \mid AOp \in \mathsf{Ops}\})$ and a concrete model given by the ADT $(B, BInit, \{BOp, BI_{Op}, BO_{Op} \mid BOp \in \mathsf{Ops}\})$. So schemas A, B give the abstract and concrete state spaces, and the corresponding per-operation I/O spaces are given by schemas AI_{Op}, AO_{Op} and BI_{Op}, BO_{Op}. We assume a retrieve relation $R_{ab} : [A;\ B]$ between the two state spaces, and for each operation Op, input and output mapping relations $RI_{ab,Op} : [AI_{Op};\ BI_{Op}]$ and $RO_{ab,Op} : [AO_{Op};\ BO_{Op}]$.

As noted above, the A to B refinement is a backward refinement. For us, backward refinement is

[7] Ordinarily it makes no sense to do this, but purses cannot afford to rely on common sense. See also the second physical arrangement described in Section 4.2.

Note that the balance enquiry operation itself is not present in the publicly available account [46] of the Mondex development. This is because the modelling 'trick' used to discover the balance (through finalisation) renders the operation specification itself confusingly trivial. Section 4.3 explains how this state of affairs came about.

given by three proof obligations (POs), *initialization* (1), *applicability* (2), and *correctness* (3):

$$\forall B';\ A' \bullet R'_{ab} \land BInit \Rightarrow AInit \tag{1}$$

$$\forall B;\ BI_{Op} \bullet (\forall A;\ AI_{Op} \bullet R_{ab} \land RIn_{ab,Op} \Rightarrow \mathrm{pre}\, AOp) \Rightarrow \mathrm{pre}\, BOp \tag{2}$$

$$\forall B;\ BI_{Op};\ B';\ BO_{Op};\ A';\ AO_{Op} \bullet (\forall A;\ AI_{Op} \bullet R_{ab} \land RIn_{ab,Op} \Rightarrow \mathrm{pre}\, AOp) \land BOp \land R'_{ab} \land ROut_{ab,Op} \Rightarrow (\exists A;\ AI_{Op} \bullet AOp \land R_{ab} \land RIn_{ab,Op}) \tag{3}$$

Backward refinement is given thus for easy comparison with retrenchment. Compared with [27], our presentation does not explicitly list input initialisation and state and output finalisation as formal proof obligations here. The issues that this raises are defered to Sections 4.3 and 4.4 below. Nevertheless it should be noted that the finalisation PO is essential in backwards refinement to provide a base case for induction that precludes the trivial retrieve relation $R = \mathsf{false}$.

We recall now the forward refinement POs too. For us there will also be three of them, *initialization* (4), *applicability* (5), and *correctness* (6), built from the same data as above. Again we omit input initialisation and the finalisations:

$$\forall B' \bullet BInit \Rightarrow (\exists A' \bullet AInit \land R'_{ab}) \tag{4}$$

$$\forall A;\ AI_{Op};\ B;\ BI_{Op} \bullet R_{ab} \land RIn_{ab,Op} \land \mathrm{pre}\, AOp \Rightarrow \mathrm{pre}\, BOp \tag{5}$$

$$\forall A;\ AI_{Op};\ B;\ BI_{Op};\ B';\ BO_{Op} \bullet R_{ab} \land RIn_{ab,Op} \land \mathrm{pre}\, AOp \land BOp \Rightarrow (\exists A';\ AO_{Op} \bullet AOp \land R'_{ab} \land ROut_{ab,Op}) \tag{6}$$

We will need retrenchments, both forward and backward between the A and B models. Firstly, the more familiar forward retrenchment. We assume a retrieve relation between the state spaces as above. Furthermore, on a per-operation basis, we have the within, output and concedes relations. The *within* relation is between the input-state spaces $W_{ab,Op} : [AI_{Op};\ A;\ BI_{Op};\ B]$. The *output* and *concedes* relations are defined over both full input-state-output frames with types $O_{ab,Op};\ C_{ab,Op} : [AI_{Op};\ A;\ A';\ AO_{Op};\ BI_{Op};\ B;\ B';\ BO_{Op}]$, though in practice, we often omit such parts of these signatures as are not needed. Now we can formulate the POs of forward retrenchment, *initialisation* (7), and *correctness* (8):

$$\forall B' \bullet BInit \Rightarrow (\exists A' \bullet AInit \land R'_{ab}) \tag{7}$$

$$\forall A;\ AI_{Op};\ B;\ BI_{Op};\ B';\ BO_{Op} \bullet R_{ab} \land W_{ab,Op} \land BOp \Rightarrow (\exists A';\ AO_{Op} \bullet AOp \land ((R'_{ab} \land O_{ab,Op}) \lor C_{AB,Op})) \tag{8}$$

Secondly, backward retrenchment. We assume the same retrieve relation as before. We also have on a per-operation basis, the within, output and concedes relations. However the signatures of the latter are different from those in the forward case, viz. $W_{ab,Op} : [AI_{Op};\ A;\ A';\ AO_{Op};\ BI_{Op};\ B;\ B';\ BO_{Op}]$ and $O_{ab,Op};\ C_{ab,Op} : [AI_{Op};\ A;\ BI_{Op};\ B]$. These differences appear natural when we examine the two POs of backward retrenchment, *initialisation* identical to (7), and *correctness* (9):

$$\forall B;\ BI_{Op};\ B';\ BO_{Op};\ A';\ AO_{Op} \bullet ((R'_{ab} \land O_{AB,Op}) \lor C_{ab,Op}) \land BOp \Rightarrow (\exists A;\ AI_{Op} \bullet AOp \land R_{ab} \land W_{ab,Op}) \tag{9}$$

4. The Balance Enquiry Quandary

In this section we outline the value transfer protocol at abstract and concrete levels, and we show how the Balance Enquiry Quandary arises, looking forward to its handling by retrenchment in the next section. In Section 4.1 we outline the straightforward abstract model, while in Section 4.2, we give a version of the concrete model. Section 4.3 outlines the backward refinement of abstract to concrete, and shows how the Balance Enquiry Quandary arises, which turns out to be quite a subtle matter due to the backward nature of the refinement. Section 4.4 reconsiders forwards and backward refinements in general.

Regarding the concrete model, we do not have the space to reproduce all the details that appear in [46], so our account will be significantly paraphrased and simplified regarding aspects that can be taken for granted compared with [46]. We mention the main points here to avoid confusion with what we *do* include in our concrete model later. Mostly such aspects concern 'packaging'. In [46] the various models genuinely reflect a community of purses undergoing transactions: at the abstract level the *from* and *to* purse names are parameters to atomic transfer operations, while at the concrete level the lower level individual purse operations are integrated into a global model using the familiar technique of Z promotion [51, 29, 47]. None of this is material to the issues dealt with in this paper, so for expository simplicity, we simplify drastically, flattening out the packaging, and hardwiring the individual purse names into the needed state components of our models, which will be restricted to just the two *to* and *from* purses. This saves considerable space.

Another area of unpackaging concerns I/O. In [46] all the different message types that arise are properly tagged and embedded in a *MESSAGE* type, and messages are then unpacked to access their contents. Here we short circuit this to a degree to avoid verbosity.

Other aspects of [46] dispensed with are: the ether of messages, which inputs are taken from, and into which outputs are placed; and the central archive of failed transactions into which the contents of purses' individual exeption logs may be decanted. In this paper, operations interface directly to the external environment (about which suitable assumptions are made, reflecting the properties formalised in the ether of [46]), and all error records stay in purses' local logs.

For clarity, all abstract variables are called *Afromvar* or *Atovar* and abstract schemas are called *AbSchema*. Concrete counterparts are *Bfromvar* or *Btovar* or just *Bvar*, and *BSchema*; only the A and B models are of interest in this work.

4.1. The Abstract Model

At the abstract level, we simplify the state of the system to two purses, *AfromPurse* and *AtoPurse*. Each of these has a 'balance' and a 'lost' component, so the abstract state is just:

AbWorld

$$Afrombalance, Atobalance : \mathbb{N}$$
$$Afromlost, Atolost : \mathbb{N}$$

We take for granted an initialisation *AbInit* of *Afrombalance* to *N* (pounds, say), and of the other components to 0.

At the abstract level, the principal operation is *AbIgnore* which skips. (It also discards any input, which we ignore; c.f. message packaging above.)

AbIgnore

$$\Xi AbWorld$$

Not only is *AbIgnore* a top level operation in its own right, it is also a nondeterministic option in all other abstract operations (except, in this paper, *AbtoBalanceEnquiry*). This has two consequences: it makes all operations total, avoiding any issues regarding availability (since all operations are always available); and it allows a purse to do nothing at any time.

The other operations of interest are *AbToBalanceEnquiry*, *AbFromBalanceEnquiry*[8] and *AbTransfer*. The balance enquiry operations are modelled as follows, the state remaining unchanged:

AbToBalanceEnquiry

$$AbIgnore$$
$$Abal! : \mathbb{N}$$
$$\hline$$
$$Abal! = Atobalance$$

AbFromBalanceEnquiry

$$AbIgnore$$
$$Abal! : \mathbb{N}$$
$$\hline$$
$$Abal! = Afrombalance$$

The *AbTransfer* operation decomposes as follows:

AbTransfer

$$AbIgnore \lor AbTransferOkay \lor AbTransferLost$$

AbTransferOkay describes a successful transfer of amount *Avalue* from *AfromPurse* to *AtoPurse*, and *AbTransferLost* describes a transfer in which *Avalue* moves between *AfromPurse*'s *Afrombalance* and *Afromlost* attributes.

AbTransferOkay

$$\Delta AbWorld$$
$$Avalue? : \mathbb{N}$$
$$\hline$$
$$Avalue? \leq Afrombalance$$
$$Afrombalance' = Afrombalance - Avalue?$$
$$Atobalance' = Atobalance + Avalue?$$
$$Afromlost' = Afromlost$$
$$Atolost' = Atolost$$

AbTransferLost

$$\Delta AbWorld$$
$$Avalue? : \mathbb{N}$$
$$\hline$$
$$Avalue? \leq Afrombalance$$
$$Afrombalance' = Afrombalance - Avalue?$$
$$Atobalance' = Atobalance$$
$$Afromlost' = Afromlost + Avalue?$$
$$Atolost' = Atolost$$

It is clear from the above that value transfer is atomic; a balance enquiry cannot occur 'part way through' an abstract transfer. It is also clear that the 'No Value Created' security property holds since

$$Afrombalance + Atobalance$$

is nonincreasing through any operation. Likewise the 'All Value Accounted' property holds since

$$Afrombalance + Atobalance + Afromlost + Atolost$$

is invariant through any operation. (Note that *Atolost* actually never changes; however in the real system, involving a large community of purses, any *to* purse might in a later transaction become a *from* purse, and its *lost* component may thus become relevant.)

[8] For the majority of the paper we only need the *to* balance enquiry, since the *from* one is completely unproblematic till Section 7 (see also Section 6). Also both the abstract and concrete balance enquiry operations here are more 'realistic' than the ones in the original formal development from which [46] was derived, for reasons that are explained at the end of Section 4.3.

4.2. The Concrete Model and Protocol

Before we embark on the details of (our simplified presentation of) the concrete protocol, it will be helpful to give some background, specifically on what is, and what is not, within the remit of the purse's security concerns. This will help to make the workings of the protocol more understandable.

Consider a £20 note. In essence, its only concern is not to be forged — a concern carefully attended to during its manufacture, before the note takes its chances in the world at large. If you own the note, you may destroy it, but (ideally) you will not be able to duplicate it. Moreover, if you use the note in a transaction to pay for goods or services, it is not the note's responsibility to ensure you receive what you anticipate. Neither is it the note's responsibility to check that the person you are dealing with is who you think they are, nor that their intentions are as you anticipate.

The Mondex value transfer protocol reflects these properties of pure cash in an electronic way. Thus the only concern is non-forgeability — services to assist in the identification of purses and their owners may well be provided by the smartcards physically containing the value in order to enhance mutual trust, but this is not a concern of the protocol.

The protocol itself proceeds by message passing, and this lends itself to at least two physical arrangements. In the first, the *to* and *from* purses are inserted into a device (called a 'wallet') which acts as a communication medium. The purse owners type in their instructions, the protocol runs, and the transfer is completed more or less instantaneously. In the second, the two purses are far apart, and are inserted into two devices, connected eg. via the internet. Mutual identification issues notwithstanding, the protocol runs as before, except that this time a transaction can have a substantial duration due to communication latency.

Now we turn to the protocol itself. Again we start with an informal sketch to aid intuition. See Fig. 1, which indicates the principal states, operations and messages which comprise it.

At the start of the protocol, *BfromPurse* and *BtoPurse* are both in the *BeaFrom* ('expecting any from') state, the basic idle state of the system. *BfromPurse* performs a *BStartFrom* operation, passing to the *Bepr* ('expecting payment request') state. *BtoPurse* performs a *BStartTo* operation, sending a *req*(uest) message to *BfromPurse* asking for the funds, and passes to the *Bepv* ('expecting payment value') state. Upon receipt of the *req* message, *BfromPurse* performs a *BfromReqPurse* operation, decrements its balance, sending a *val*(ue) message containing the actual funds to be transfered to *BtoPurse*, and passes to the *Bepa* ('expecting payment ack') state. Upon the arrival of the *val* message, *BtoPurse* performs a *BtoValPurse* operation, increments its balance, sending an *ack*(nowledgement) message to *BfromPurse*, and passes to the *BeaTo* state (another idle state of the system[9]). Upon the arrival of the *ack* message, *BfromPurse* performs a *BfromAckPurse* operation, returning to the *BeaFrom* state.

While the above gives a good intuitive idea, it neglects many issues that need to be understood in order to have a mathematically robust protocol. To get a grip on these we turn to the more detailed formal version, simplified (compared with [46]) so as to discard as much complexity as is reasonable. In fact we simplify to the extent that the transfer protocol remains secure in the face of permitted failures only provided **all transactions involve different and nonzero amounts**. This allows us to dispense with the complication of an ether and with sequence numbers, as the information in the 'paydetails' part of the concrete state is sufficient for secure operation in this case.

Each of *BfromPurse* and *BtoPurse* has a 'status', a 'balance', a 'paydetails' record, and an 'excep-

[9] In fact, there is no logical need for a *BeaTo* state separate from the *BeaFrom* state. However the concrete model was built to reflect an existing (and unalterable) implementation, so the evident simplification of the specification was not an option.

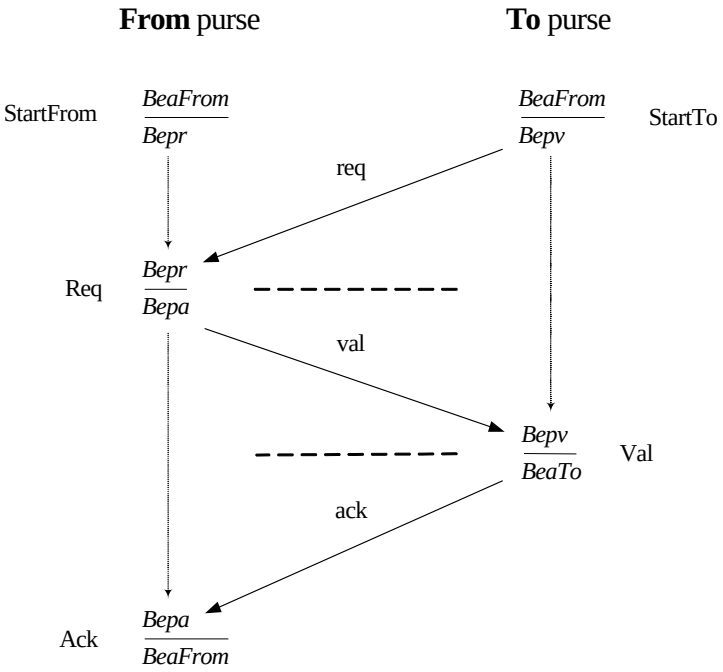

Figure 1. The Mondex Concrete Protocol.

tion log'. This constitutes the core concrete state. To this are added a couple of derived variables, *Bdefinitelylost* and *Bmaybelost*, which refer to monetary value which is (from the protocol's point of view) not safely lodged in one or other of the balances. These variables also abbreviate some of the reasoning later, and are used in the A to B retrieve relation R_{ab}.

$$BSTATUS == \{BeaFrom, BeaTo, Bepr, Bepv, Bepa\}$$

BetweenWorld

$$\begin{array}{l} Bfromstatus, Btostatus : BSTATUS \\ Bfrombalance, Btobalance : \mathbb{N} \\ Bfrompaydetails, Btopaydetails : \mathbb{N} \\ Bfromexlog, Btoexlog : \mathbb{P}\,\mathbb{N} \\ Bdefinitelylost : \mathbb{P}\,\mathbb{N} \\ Bmaybelost : \mathbb{P}\,\mathbb{N} \\ \hline Bdefinitelylost = \\ \quad Btoexlog \cap \\ \quad (Bfromexlog \cup (\mathsf{if}\ Bfromstatus = Bepa\ \mathsf{then}\ \{Bfrompaydetails\}\ \mathsf{else}\ \varnothing)) \\ Bmaybelost = \\ \quad (\mathsf{if}\ Btostatus = Bepv\ \mathsf{then}\ \{Btopaydetails\}\ \mathsf{else}\ \varnothing) \cap \\ \quad (Bfromexlog \cup (\mathsf{if}\ Bfromstatus = Bepa\ \mathsf{then}\ \{Bfrompaydetails\}\ \mathsf{else}\ \varnothing)) \end{array}$$

We take for granted an initialisation of *Bfrombalance* to *N* (pounds, say, as previously), and of the other components to 0 or $\varnothing$ as appropriate.

Regarding the operations, we have *BIgnore* as in the abstract case:

BIgnore

Ξ*BetweenWorld*

and we also have *BfromAbort* and *BtoAbort*, used to clean up uncompleted or uncompletable transfers when the environment decides to initiate a fresh transaction. These operations also ignore their inputs if any, which we disregard. Also, henceforth the phrase 'RestSame.....' indicates that state variables whose *after* values are not mentioned in the predicate part of the schema (i.e. they are not 'visibly assigned to') remain unchanged,[10] aside that is, from the dependent variables *Bdefinitelylost*, *Bmaybelost*, which are required to change in line with the others, according to their definitions:

BfromAbort

Δ*BetweenWorld*

$Bfromstatus' = BeaFrom$

$Bfromexlog' =$

$Bfromexlog \cup (\text{if } Bfromstatus = Bepa$

$\text{then } \{Brompaydetails\}$

$\text{else } \varnothing)$

RestSame.....

BtoAbort

Δ*BetweenWorld*

$Btostatus' = BeaFrom$

$Btoexlog' =$

$Btoexlog \cup (\text{if } Btostatus = Bepv$

$\text{then } \{Btopaydetails\}$

$\text{else } \varnothing)$

RestSame.....

The balance enquiry operations are straightforward. As for the abstract model, the *BtoPurse* enquiry is the one of most interest until Section 7:

BtoBalanceEnquiry

BIgnore

$Bbal! : \mathbb{N}$

$Bbal! = Btobalance$

BfromBalanceEnquiry

BIgnore

$Bbal! : \mathbb{N}$

$Bbal! = Bfrombalance$

Regarding value transfers, we supplement our preceding intuitive account with the following additional details.

According to one or other of the physical arrangements discussed above, the device holding a given purse receives its instructions from the environment. Upon receipt of the instructions, the device calls the *BStartFrom* operation in *BfromPurse* (and/or the *BStartTo* operation in *BtoPurse* if appropriate). Both of these operations are intended to initiate a new transaction, starting from a completely arbitrary state of affairs. Thus the *BStart..* operation might need to: (i) *BIgnore*, since that is always an option; (ii) *B..Abort* (since a preceding transaction might still be running) and then *BIgnore*; (iii) *B..Abort*, and then actually start the transaction.

The *BIgnore* option skips. The *B..Abort* option logs the details of any uncompleted transfer if it needs to. (Incomplete transfers are detected by examining *B..status*, which should be in either the *BeaFrom* state or the *BeaTo* state, the idle states of the system.)

[10] As if they were in a suitable Ξ schema.

The third option aborts as just described, and having intialised the purse (to the *BeaFrom* state) in the *B..Abort* operation, performs a *BStartFromPurseEafromOkay* or *BStartToPurseEafromOkay* operation as appropriate. The inputs to operations *BStartFromPurseEafromOkay* and *BStartToPurseEafromOkay* contain the value to be exchanged. This is recorded locally in the *Bfrompaydetails* and *Btopaydetails* variables. *BfromPurse* moves into the *Bepr* state, and *BtoPurse* moves into the *Bepv* state; *BtoPurse* sends a *req* message to *BfromPurse* citing the details of the desired payment.

BStartFrom

$$BIgnore \lor BfromAbort \lor (BfromAbort \fatsemi BStartFromPurseEafromOkay)$$

BStartTo

$$BIgnore \lor BtoAbort \lor (BtoAbort \fatsemi BStartToPurseEafromOkay)$$

BStartFromPurseEafromOkay

$\Delta BetweenWorld$
$inval? : \mathbb{N}$

$0 \leq inval? \leq Bfrombalance$
$Bfromstatus = BeaFrom$
$Bfromstatus' = Bepr$
$Bfrompaydetails' = inval?$
RestSame.....

BStartToPurseEafromOkay

$\Delta BetweenWorld$
$inval?, req! : \mathbb{N}$

$Btostatus = BeaFrom$
$Btostatus' = Bepv$
$Btopaydetails' = inval?$
RestSame.....
$req! = inval?$

Next, in operation *BfromReqPurse* (assuming it chooses not to *BIgnore*), once the *req* message arrives at *BfromPurse*, *BfromPurse* decrements its balance, and sends a *val* message to *BtoPurse*, moving to state *Bepa*. Note that the summed purse balances are at this point less than the money deposited at the bank. This satisfies the 'No Value Created' property should all trace of this transaction suddenly disappear.

BfromReqPurse

$$BIgnore \lor BfromReqPurseOkay$$

BfromReqPurseOkay

$\Delta BetweenWorld$
$req? : \mathbb{N}$
$val! : \mathbb{N}$

$req? = Bfrompaydetails$
$Bfromstatus = Bepr$
$Bfromstatus' = Bepa$
$Bfrombalance' = Bfrombalance - Bfrompaydetails$
RestSame.....
$val! = req?$

Next, in operation *BtoValPurse* (assuming it chooses not to *BIgnore*), once the *val* message arrives at *BtoPurse*, *BtoPurse* adds the value to its balance, and sends an *ack*, moving to the *BeaTo* state. At this juncture, the summed balances are once more what they ought to be.

BtoValPurse

$$BIgnore \lor BtoValPurseOkay$$

BtoValPurseOkay

$$\Delta BetweenWorld$$
$$val? : \mathbb{N}$$
$$ack! : \mathbb{N}$$

$$val? = Btopaydetails$$
$$Btostatus = Bepv$$
$$Btostatus' = BeaTo$$
$$Btobalance' = Btobalance + Btopaydetails$$
RestSame.....
$$ack! = val?$$

Finally, in operation *BfromAckPurse* (assuming it chooses not to *BIgnore*), once the *ack* arrives at *BfromPurse*, *BfromPurse* returns to the *BeaFrom* state.

BfromAckPurse

$$BIgnore \lor BfromAckPurseOkay$$

BfromAckPurseOkay

$$\Delta BetweenWorld$$
$$ack? : \mathbb{N}$$

$$ack? = Bfrompaydetails$$
$$Bfromstatus = Bepa$$
$$Bfromstatus' = BeaFrom$$
RestSame.....

This completes the protocol description for a straightforward successful run. Aside from the nondeterminism in the two *BStart*... operations, the protocol is sequential. Since it consists of several steps, it is clearly possible to interlace a *BtoBalanceEnquiry* step into the middle of it, if the environment perversely wishes to do that.[11]

To the above we must add our assumptions about allowed failures. The failures we permit are *BfromAbort*s and *BtoAbort*s at any time, and the loss of messages. A further assumption we rely on

[11]The second physical arrangement makes it entirely plausible that an impatient recipient would make a balance enquiry in the middle of a long lived transfer, to determine whether his funds had arrived yet.

however, is that *req*, *val*, *ack* messages are **unforgeable** (this in reality being implemented by some unspecified cryptological function). This means for example that if a *BfromReqPurseOkay* operation is performed, then its *req*? input (which equals *Bfrompaydetails*) came from a unique invokation of a *BStartToPurseEafomOkay* operation, which output a *req*! (which equaled $Btopaydetails'$ at the time) and that *req*? $=$ *req*!. Similarly for the other two messages. We refer to such arguments as 'unforgeability reasoning' for short, in the proofs below.

Since the concrete model is a refinement of the abstract one, the 'No Value Created' and 'All Value Accounted' security properties, which are functional properties of the model state, will hold for it.[12] However to gain better intutition, it is instructive to see independently how they hold in the concrete model.

Firstly, every prefix of any run of the protocol preserves the 'No Value Created' security property. Furthermore, despite being no longer a nonincreasing quantity as in the abstract case,

$$Bfrombalance + Btobalance$$

never exceeds the total at the beginning of the transfer. This would not be so if, rather than following the order of steps given, *BtoPurse* first incremented its balance, and *BfromPurse* later decremented its one.

Secondly, let us overview how the additional state in the models ensures that the 'All Value Accounted' security property is maintained in the face of protocol failures. The protocol failures we consider are that an unexpected abort is called from the environment, or one of the *req*, *val*, *ack* messages disappears.

It is clear from Fig. 1, that the critical period is between the dashed horizontal lines, i.e. during the time when the *val* message is in transit, since that is the only period during which not all of the value is to be found in one or other of the balances (assuming a successful transaction).

Unfortunately, $Btostatus = Bepv$ does not reveal whether the *from* purse has crossed the first dashed line; likewise for $Bfromstatus = Bepa$ and the *to* purse crossing the second dashed line. In both cases, and either side of the dashed line, an abort (which is the mechanism by which either protagonist gets detached from the protocol), causes the same thing to happen (i.e. a dump of 'paydetails' into 'exlog'). We must distinguish the cases when the paydetails dumps are significant for 'All Value Accounted', and when they are not. We claim that the concrete 'All Value Accounted' invariant is:

$$Bfrombalance + Btobalance + \Sigma\, Bdefinitelylost + \Sigma\, Bmaybelost$$

We must argue that if, for a specific transaction, $Bfromstatus = Bepa \wedge Btostatus = Bepv$ holds, or an abort has happened between the two dashed lines, then $\Sigma\, Bdefinitelylost + \Sigma\, Bmaybelost$ contributes *val* to the value of the invariant, and if it doesn't hold, or an abort hasn't happened, it doesn't contribute. Restricting to the scenario of a single isolated transaction for simplicity, there are thus three cases:

1. If failure occurs before *BfromPurse* goes into state *Bepa* (that is, before it sends the value), then any abort of *BfromPurse* adds nothing to *Bfromexlog*, (regardless of whether the *to* purse has aborted yet or not). Consequently the second intersectants of both *Bdefinitelylost* and *Bmaybelost* are empty, and so $\Sigma\, Bdefinitelylost + \Sigma\, Bmaybelost$ contributes nothing to the value of the invariant.

[12] At least they will do so provided failures other than the permitted ones do not occur. If one were able to forge *BStartTo* calls and appropriate *val* messages, one could persuade *BtoPurse* to increase its balance by arbitrary amounts, without corresponding *BfromPurse* decrements. The messages are cryptographically protected to stop this from happening.

2. If failure occurs after *BtoPurse* goes into state *BeaTo*, then any abort of *BtoPurse* adds nothing to *Btoexlog*, (regardless of whether the *from* purse has aborted yet or not). Consequently neither Σ *Bdefinitelylost* nor Σ *Bmaybelost* contributes anything to the value of the invariant, since both of their first intersectants are empty.

3. If failure occurs between the preceding two events, then $Bfromstatus = Bepa \wedge Btostatus = Bepv$ holds (at the moment of message loss or just prior to an abort). Either *BfromPurse* has aborted or not; either way the second intersectants of both *Bdefinitelylost* and *Bmaybelost* contribute the value required to the invariant, compensating for the reduced value of the summed balances. Either *BtoPurse* has aborted or not. In the former case, the first intersectant of *Bdefinitelylost* contributes the value required to the invariant (and *Bmaybelost* contributes nothing), and the intersection takes care of the double counting. In the latter case, the first intersectant of *Bmaybelost* contributes the value required to the invariant (and *Bdefinitelylost* contributes nothing), and the intersection takes care of the double counting. In all cases, the reduced value of the balances is suitably compensated.

To the preceding must be added an awareness of what happens when there is more than just a single transaction. Since the *to* purse finishes a transaction earlier than the *from* purse, and the *from* purse can start a transaction later than the *to* purse, the two purses may simultaneously be in the *Epa*/*Epv* states for *different* transactions. What distinguishes such a case from the preceding one, according to our assumptions, is that the *from* and *to* paydetails records will be different at such a moment, and that the *to* transaction, being later than the *from* one, cannot possibly have been aborted by *BfromPurse* yet, i.e. *Btopaydetails* is not in *Bfromexlog*.

The double counting evident in case 3 explains why our protocol is safe only for a system history containing transactions for distinct amounts. Suppose a transaction for £10 starts, and the *req* message is lost. A *BtoPurse* abort puts a £10 entry into *Btoexlog*; *BfromPurse* is not unhappy. A second transaction for £10 starts, and the *ack* message is lost. A *BfromPurse* abort puts a £10 entry into *Bfromexlog*; *BtoPurse* is not unhappy. Now there is a spurious £10 contribution to the invariant arising from *Bdefinitelylost*. Of course in the real protocol of [46], sequence numbers and purse names occur in paydetails records to disambiguate such situations. For brevity, we will continue with our simpler, but more artificial, picture.

4.3. The Abstract-Concrete Refinement

In this section we examine the refinement from the abstract to the concrete model in order to show exactly how the Balance Enquiry Quandary arises. Although this refinement in effect covers ground thoroughly discussed in [46], our presentation is consistent with the significant repackaging and simplification of the previous section, and thus makes the present paper conveniently self contained. Besides, due to the backward orientation of the refinement proof, the way the quandary arises is quite subtle. Everything goes quite smoothly almost to the very end. All the expected refinement proof obligations discharge unproblematically, and it is only the global consistency of the outputs that fails. A detailed treatment is needed to bring this out clearly.

The Balance Enquiry Quandary can be illustrated thus. Consider a *to* purse with a balance *bal*, which sends a *req* for £*v*, and, before it receives the *val*, responds to a balance enquiry demand from the environment. What value should it display? In the abstract case, the protocol has already completed, either succesfully (in which case the *to* balance is $£(bal + v)$), or unsuccessfully (in which case the *to* balance is £*bal*, and the *from* purse has registered the loss). But in the concrete case, we do not yet

know whether the protocol will succeed or fail, so do not yet know whether to display *bal* or $bal + v$. Choosing *bal* seems more sensible as it indicates that the value has not yet arrived, but runs up against subtle technical problems; choosing $bal + v$ presumes the arrival of the *val*, which may never happen.

Turning to the technical details, we note that our discussion of the maintenance of the 'All Value Accounted' invariant necessitated the proper handling of the double counting in *Bdefinitelylost* and *Bmaybelost* in the concrete model. This suggests that the relationship between models is likely to be nondeterministic. This is borne out by the retrieve relation R_{ab}:

$$R_{ab} \mathrel{\widehat=} \exists\, chosenlost : \mathbb{P}\,\mathbb{N} \bullet R_{abCl}$$

where

R_{abCl}

$$\begin{array}{l} AbWorld \\ BetweenWorld \\ chosenlost : \mathbb{P}\,\mathbb{N} \\ \hline chosenlost \subseteq Bmaybelost \\ Abfromlost = \Sigma\, Bdefinitelylost + \Sigma\, chosenlost \\ Abtolost = 0 \\ Abfrombalance = Bfrombalance \\ Abtobalance = Btobalance + \Sigma\, Bmaybelost - \Sigma\, chosenlost \end{array}$$

The relationships in R_{abCl} between abstract and concrete entities reinforce our intuition about the names *Bdefinitelylost* and *Bmaybelost*. It is clear that *Bdefinitelylost* refers to transactions that have proceeded far enough for it to be known that they have failed, while *Bmaybelost* refers to those whose fate is not yet sealed for sure.[13] The various possible values for *chosenlost* correspond to different possible outcomes of a protocol run which is not yet fully played out.

We now indicate how the various operations arise as refinements of abstract ones. The main complication arises from the $\exists\, chosenlost\ldots$ in R_{ab}. Since R'_{ab} occurs in the antecedent of the correctness PO (3) of the (backward) refinement from model A to model B, we need to show that the consequent, requiring a suitable choice of *chosenlost*, can be derived for all possible values of $chosenlost'$.

There is also a simplification, which arises from the totality of all the operations, and which follows directly from the disjunctive presence of the *Ignore* operations in all non-skip operations; this allows us to ignore the applicability PO (2). A further simplification comes from the very marginal role played by I/O in the abstract model (in particular). Thus $RIn_{ab,Op}$ and $ROut_{ab,Op}$ in the correctness PO (3) will be ignored except when something nontrivial is at stake.

Initialisation: Since the initial $Bmaybelost = \varnothing$, $chosenlost = \varnothing$ is the only possibility. Therefore (1) holds.

BIgnore, ***BfromAbort***, ***BtoAbort*** all refine ***AbIgnore***: Since *BIgnore* and *AbIgnore* both skip, the *BIgnore* case is trivial.

For *BfromAbort*, if $Bfromstatus \neq Bepa$, then aside from $Bfromstatus' = BeaFrom$, *BfromAbort* skips, and so discharging (3) is trivial. So suppose $Bfromstatus = Bepa$. Suppose $Btostatus = Btostatus'$

[13] Oh, how the first author wishes that the variable had instead been called *Bmayyetsucceed*.

$\neq$ *Bepv*. Then *Bmaybelost* $=$ *Bmaybelost*$'$ $= \varnothing$, so *chosenlost* $=$ *chosenlost*$'$ $= \varnothing$ is forced. The rest of the discharging of (3) is now similar to that for the skipping case.

Suppose *Btostatus* $=$ *Btostatus*$'$ $=$ *Bepv*. Then either *Bfrompaydetails*$'$ $=$ *Btopaydetails*$'$ or not. In the former case, *Bfrompaydetails* $=$ *Bfrompaydetails*$'$ $=$ *Btopaydetails* $=$ *Btopaydetails*$'$; and this amount is in neither of *Bfromexlog*, *Btoexlog*, by unforgeability reasoning. So we force *Bmaybelost* $=$ $\{$*Bfrompaydetails*$\}$ and *Bmaybelost*$'$ $= \varnothing$. The latter implies *chosenlost*$'$ $= \varnothing$, and we have a choice of either $\varnothing$ or $\{$*Bfrompaydetails*$\}$ for *chosenlost*. If we choose *chosenlost* $= \{$*Bfrompaydetails*$\}$, the movement of *Bfrompaydetails* out of *chosenlost* matches the change in *Bmaybelost*, and leads to a discharge of (3) similar to that for the skipping case. In the latter case (i.e. *Btopaydetails*$'$ $\neq$ *Bfrompaydetails*$'$), *Btopaydetails*$'$ $\notin$ *Bfromexlog*$'$, as argued earlier. So *Bmaybelost* $=$ *Bmaybelost*$'$ $= \varnothing$ is forced, a case already covered.

For *BtoAbort*, if *Btostatus* $\neq$ *Bepv*, then aside from *Btostatus*$'$ $=$ *BeaFrom*, *BtoAbort* skips, and so discharging (3) is trivial. So suppose *Btostatus* $=$ *Bepv*. If *Bfromstatus* $=$ *Bfromstatus*$'$ $\neq$ *Bepa* then either *Btopaydetails* is in *Bfromexlog* or not. This means that either *Bmaybelost* $= \{$*Bfrompaydetails*$\}$ (in the former case), or *Bmaybelost* $= \varnothing$ (in the latter case). In any event, *Bmaybelost*$'$ $= \varnothing$ holds. Both cases correspond to ones already discussed.

Suppose finally that *Bfromstatus* $=$ *Bfromstatus*$'$ $=$ *Bepa*. This is as in the corresponding case for *BfromAbort*.

BStartFrom and ***BStartTo*** both refine ***AbIgnore***: To start with, we argue that the suboperations ***BStartFromPurseEafromOkay*** and ***BStartToPurseEafromOkay*** both refine ***AbIgnore***. For *BStartFromPurseEafromOkay*, since neither of *Bfromstatus*, *Bfromstatus*$'$ is *Bepa*, and aside from these *BStartFromPurseEafromOkay* skips, *Bmaybelost* $=$ *Bmaybelost*$'$ and any value acceptable for *chosenlost*$'$ is therefore also acceptable for *chosenlost*, discharging (3).

For *BStartToPurseEafromOkay*, the argument is a little different since *Btostatus*$'$ $=$ *Bepv*. We show that *Bmaybelost*$'$ $= \varnothing$ holds. This follows because firstly, if *Bfrompurse* and *Btopurse* are in the same transaction then *Bfromstatus*$'$ $\neq$ *Bepa*, since the *from* purse only goes into the *Bepa* state upon receipt of the *req* message which the *BStartToPurseEafromOkay* operation is only just sending out; and secondly, if the two purses are in different transactions, then, as argued earlier, the *to* purse is in a later one than the *from* purse, so the *from* purse cannot have aborted the *to* purse's transaction yet. So *Bmaybelost*$'$ $= \varnothing$ and clearly *Bmaybelost* $= \varnothing$ too, so this case is essentially as for a skip.

On the basis of the above, since *AbIgnore* $\fatsemi$ *AbIgnore* $=$ *AbIgnore*, and since there are no non-trivial applicability issues, it is easy enough to conclude that *BStartFrom* and *BStartTo* both refine *AbIgnore*.

BfromReqPurseOkay refines ***AbTransfer***: On the basis of unforgeability reasoning as refered to earlier, our *BfromReqPurseOkay* operation entails the receipt of a suitable *req* message, emitted from an earlier *BtoStart* whose payment details matched those of *BfromStart*. This left *BtoPurse* in status *Bepv* until it either receives the *val* emitted from our *BfromReqPurseOkay*, or *BtoAbort*s.

Suppose *Btostatus* $=$ *Btostatus*$'$ $\neq$ *Bepv*. Then *BtoAbort* must have happened, putting the payment details into *Btoexlog*. Also *Bmaybelost* $=$ *Bmaybelost*$'$ $= \varnothing$, and therefore *chosenlost* $=$ *chosenlost*$'$ $= \varnothing$ are all forced. The decrease in *Bfrombalance* now matches the increase in *Bdefinitelylost* and leads to the discharge of (3) for the refinement of *AbTransferLost* in this case.

Suppose *Btostatus* $=$ *Btostatus*$'$ $=$ *Bepv*. Since *Bfromstatus*$'$ $=$ *Bepa*, we conclude *Bmaybelost*$'$ $=$

$\{Bfrompaydetails'\}$. Since $Bfromstatus \neq Bepa$, $Bmaybelost = \varnothing$.[14] Now there are two possibilities for $chosenlost'$: namely $\varnothing$, and $\{Bfrompaydetails'\}$; and a single possibility for $chosenlost$: namely $\varnothing$.

The $chosenlost' = chosenlost = \varnothing$ option is consistent with the decrease in *Bfrombalance* matching the increase in *Abtobalance* (the latter implied by the increase in *Bmaybelost*), and leads to the discharge of (3) for the refi nement of *AbTransferOkay* in this case.

The $chosenlost' = \{Bfrompaydetails'\}$ and $chosenlost = \varnothing$ option is consistent with the decrease in *Bfrombalance* matching the increase in *Abfromlost* (the latter implied by the increase in *chosenlost*), and leads to the discharge of (3) for the refi nement of *AbTransferLost* in this case.

On the basis of the above we quickly conclude that *BfromReqPurse* refi nes *AbTransfer*. In all three cases we also derive that $Avalue? = req?$, the expected relation between inputs.

BtoValPurseOkay refi nes ***AbIgnore***: Via unforgeability reasoning, our *BtoValPurseOkay* operation entails the receipt of a suitable *val* message, emitted from an earlier *BfromReqPurseOkay* whose payment details matched those of *BtoPurse*. This left *BfromPurse* in status *Bepa* until it either receives the *ack* emitted from our *BtoValPurseOkay*, or *BfromAbort*s.

Suppose $Bfromstatus = Bfromstatus' \neq Bepa$. Then *BfromAbort* must have happened, putting the payment details into *Bfromexlog*. Also $Bmaybelost' = \varnothing$ is forced (because of $Btostatus'$), implying $chosenlost' = \varnothing$. Since $Btostatus = Bepv$, $Bmaybelost = \{Btopaydetails\}$, so *chosenlost* is either $\varnothing$ or $\{Btopaydetails\}$. The choice $chosenlost = \varnothing$ is consistent with the increase in *Btobalance* matching the decrease in *Bmaybelost*, and leads to the discharge of (3).

Suppose $Bfromstatus = Bfromstatus' = Bepa$. Since $Btostatus = Bepv$, we conclude $Bmaybelost = \{Btopaydetails\}$, leading to $chosenlost = \varnothing$ or $chosenlost = \{Btopaydetails\}$. Since $Btostatus' \neq Bepv$, $Bmaybelost' = \varnothing$, fi xing $chosenlost' = \varnothing$. The choice $chosenlost = \varnothing$ enables the argument for the preceding case to be reused to discharge (3).

On the basis of the above, we swiftly conclude that *BtoValPurse* refi nes *AbIgnore*.

BfromAckPurseOkay refi nes ***AbIgnore***: Via unforgeability reasoning, our *BfromAckPurseOkay* operation entails the receipt of a suitable *ack* message, emitted from an earlier *BtoValPurseOkay* whose payment details matched those of *BfromPurse*. This left *BtoPurse* in status *BeaTo* until the next *BtoAbort*.

Now if $Btostatus = Bepv$, then *BtoPurse* has embarked on a future transaction, which could not be in *Bfromexlog* as noted earlier. In this case, and also if $Btostatus \neq Bepv$, $Bmaybelost = Bmaybelost' = \varnothing$ is forced, implying $chosenlost = chosenlost' = \varnothing$. Since, aside from the change in *Bfromstatus*, *BfromAckPurseOka* and *AbIgnore* both skip, the invariance of *chosenlost* is consistent with a trivial discharge of (3).

On the basis of the above, we swiftly conclude that *BfromAckPurse* refi nes *AbIgnore*.

The above covers the A to B backwards refi nement of [46] in our simplifi ed form. We have one additional detail to address (this being the point of the whole paper), namely that despite the innocuity of both operations essentially being skips, *BtoBalanceEnquiry* inappropriately refi nes *AbToBalanceEnquiry*. As becomes clear in the details below, the backward PO (3) adroitly sidesteps the issue, since the dependency between before- and after- states in the PO is opposite to causal. Only in the global reconciliation of outputs do we hit on the problem.

BtoBalanceEnquiry *inappropriately* refi nes ***AbToBalanceEnquiry***: Firstly we point out that during a

[14] Since we assumed that all payments are for different amounts, the value in *Bfrompaydetails* cannot also be in *Bfromexlog*, ruling out the only other way that *Bmaybelost* might be nonempty.

BtoBalanceEnquiry, if $Btostatus = Btostatus' = Bepv$ and $Bfromstatus = Bfromstatus' = Bepa$ are not both true, then there is no problem. This follows since in such a case $Bmaybelost = Bmaybelost' = \varnothing$. This in turn follows from either $Btostatus = Btostatus' \neq Bepv$, or from the fact that *BtoPurse*'s current transaction cannot be in *Bfromexlog*. Since $Bmaybelost = Bmaybelost' = \varnothing$, $chosenlost = chosenlost' = \varnothing$ too, and the skip-like nature of both operations makes (3) easy to discharge.

Suppose then that $Btostatus = Btostatus' = Bepv$ and $Bfromstatus = Bfromstatus' = Bepa$ and $Bfrompaydetails = Bfrompaydetails' = Btopaydetails = Btopaydetails'$ (the critical case, a situation that obtains while the *val* message is in flight during a successful transaction[15]). Then $Bmaybelost = Bmaybelost' = \{Btopaydetails\}$. So there are two independent choices ($\varnothing$ and $\{Btopaydetails\}$) for each of *chosenlost* and *chosenlost'*. We examine the two *chosenlost'* options in turn: the correctness PO stipulates that we must find a suitable *chosenlost* for each *chosenlost'* that makes the PO antecedent valid. What happens next depends on how we treat outputs.[16]

Suppose $chosenlost' = \{Btopaydetails\}$. Then $Abtobalance' = Btobalance'$, and noting the skip-like nature of both operations, the choice $chosenlost = \{Btopaydetails\}$ leads to $Abtobalance = Btobalance$ too; moreover $Abal! = Bbal!$. Now the normal appropriate relationship between outputs is expressed as $ROut_{ab,ToBalanceEnquiry} = (Abal! = Bbal!)$. So this is true here, the PO antecedent is true, and hence (3) discharges OK. An alternative policy on outputs is to ignore them (as done in [46]), expressed via $ROut_{ab,ToBalanceEnquiry} = \mathsf{true}$. This weakens the PO antecedent, so since the case under consideration validated the stronger antecedent, it will validate the weaker one, and (3) discharges OK here too.

Suppose $chosenlost' = \varnothing$. Then $Abtobalance' = Btobalance' + Btopaydetails$ and noting the skip-like nature of both operations, the choice $chosenlost = \varnothing$ is forced, forcing $Abtobalance = Btobalance + Btopaydetails$ also. Moreover $Abal! = Abtobalance'$ and $Bbal! = Btobalance'$ by definition, which in turn forces $Abal! = Bbal! + Btopaydetails$. Now the output policy makes a difference. If $ROut_{ab,ToBalanceEnquiry} = (Abal! = Bbal!)$, then because it's a backward refinement, the PO antecedent is false, so there is nothing to prove and (3) discharges OK. If $ROut_{ab,ToBalanceEnquiry} = \mathsf{true}$, then the PO antecedent is true and (3) discharges OK because of the skip-like behaviour on the states.

For the $chosenlost' = \varnothing$ option, both output policies have inappropriate aspects. Making the choice $ROut_{ab,ToBalanceEnquiry} = \mathsf{true}$ clearly violates the domain level requirements of a balance enquiry operation but allows the PO to be discharged with an antecedent valid in the case of interest. Choosing $ROut_{ab,ToBalanceEnquiry} = (Abal! = Bbal!)$ reflects the requirements appropriately, but causes the PO to discharge spuriously, i.e. via a false antecedent.[17]

Although the operation satisfies the PO (3) in the last case, despite $(Abal! = Bbal!)$ being false, the inappropriateness shows up globally, as any runs of the abstract/concrete systems featuring this case would have output streams that did not match up pointwise. In backward refinement, output matching is policed by the output finalisation PO, omitted from Section 3, but which in our terminology reads:

$$\forall\, BO_{Op} \bullet \exists\, AO_{Op} \bullet ROut_{ab,Op} \tag{10}$$

Used under the assumption that the predicate(s) in its body must be made valid by all abstract/concrete output pairs that occur in a simulation (in the same manner as eg. the initialisation POs (4) and (1)),

[15] The case with unequal paydetails leads as usual to an empty *Bmaybelost*

[16] Note that this is the first time that we have had cause to mention outputs at all.

[17] A forward refinement PO, eg. (6) would demand $ROut_{ab,ToBalanceEnquiry}$ in the consequent, which would consequently fail. See Section 7.

it demands that $ROut_{ab,Op}$ holds for all corresponding pairs of outputs. In the $ROut_{ab,ToBalanceEnquiry} = (Abal! = Bbal!)$ case, this obviously fails in the critical case for successful transfers.

The output finalisation PO acts in concert with the state finalisation PO, also omitted from Section 3, and which reads:

$$\forall B \bullet BFin \Rightarrow (\exists A \bullet AFin \wedge R_{ab}) \tag{11}$$

This is also used under the assumption that the predicates in its body must be made valid by all abstract/concrete possible final state pairs (which in fact means all abstract/concrete reachable state pairs), so that all possible reachable state pairs satisfy the retrieve relation.[18] Together, these additional statements ensure the antecedents of the correctness PO (3) are validated each time it is used in the backwards induction from final to initial configuration, so that the PO is not discharged vacuously.

In our case, the validation of (10) by the relevant abstract/concrete output pair cannot be discharged for the critical case, thus preventing the refinement from being proved. This fact motivated the choice of vacuous output relations in the Mondex development, and the resulting trivialisation of the balance enquiry operations' specifications.[19] Ultimately, it made no sense to retain such unintuitive balance enquiry operations in [46] and they were removed.

Before moving on, we make a final comment. Why not sweep away the whole problem of balance enquiries by preceding each concrete balance enquiry with an abort, to bring the abstract and concrete states into correspondence, as is done when initiating new value transfers? The answer lies in the second physical arrangement discussed above. When transactions are long lived due to communication latency, an impatient recipient who makes an abort-preceded balance enquiry before the value has arrived, assuredly aborts it, increasing his frustration. It was a high level requirement of Mondex that this was not to happen, with the consequences that we have seen.

4.4. Forward and Backward Refinements

At this point it is appropriate to contrast forward and backward refinement. In the case that applicability issues are trivial as in Mondex, regarding logic alone, the two refinements are exact duals of one another.[20] Forward refinement establishes retrieving initial states (4), and proceeds by induction to the final states, generating outputs. Input initialisation, which insists on eg. pointwise satisfaction of the input relation, interpreted as above:

$$\forall BI_{Op} \bullet \exists AI_{Op} \bullet RIn_{ab,Op} \tag{12}$$

guarantees that the antecedent of each implication in the induction is valid. Consequently all the states and outputs generated by the induction satisfy the retrieve and output relations. Backward refinement works the opposite way round. Starting with retrieving final states (11), it proceeds by induction to the

[18] Note how (11) precludes $R_{ab} = \mathsf{false}$ by having R_{ab} only in the consequent; c.f. (4).

[19] The workaround for the vacuous output relations in Mondex was to use state observations via the state finalisation PO (11) applied to the abstract/concrete state pair that the balance enquiries have reached. Since the abstract/concrete state pair is one reachable without balance enquiries, the state observation confirms that the balance enquiries do not after all produce outputs that could not be justified.

[20] If applicability issues are not trivial this is no longer true, and the asymmetry between forwards and backwards directions comes from the asymmetry of the totalisation procedure used to derive the POs for partial operations (see [27]). This is shown up most forcefully in the visible difference between the applicability POs (5) and (2).

initial states, generating inputs. Output finalisation (10), which insists on eg. pointwise satisfaction of the output relation, guarantees that the antecedent of each implication in the induction is valid. Consequently all the states and inputs generated by the induction satisfy the retrieve and input relations.

Regarding causality however, the two approaches can be distinguished. Operations work by being offered inputs in before-states and they produce after-states and outputs. The reverse, producing outputs for after-states, and subsequently being offered inputs for before-states is not an option. Thus, in forward refinement, it is reasonable to assume that abstract/concrete operations will be compared only when their before-states and inputs match up suitably (since this is under the control of the environment), which sanctions the relegation of input initialisation to a validation issue. (With before-states and inputs matched, the PO (6) ensures that outputs and final states will satisfy the required properties too.) In backward refinement it is much less reasonable to regard the final state and outputs as being under the control of the environment. Thus it is considerably harder to relegate state and output finalisations to a validation concern. The issue gets exacerbated if we allow system executions to become infinite.[21] Then pure induction from initial states is insufficient, since unlike the (backward) finalisation PO, the backward initialisation PO has the retrieve relation in the antecedent, making everything that is derived, contingent on the hypothesised but nonexistent final states. In this scenario one must use stronger tools, such as finite branching assumptions and König's Lemma [34], to establish the refinement globally.

Despite all of this, our formulation will categorise issues regarding input initialisation and (state and output) finalisations as validation issues, even though in the context of refinement, the 'validation' in question is nothing more than the discharge of the POs as described above. This is to preserve the structural analogy with retrenchment. In retrenchment, there is *no* possibility of using induction based on a preserved retrieve relation as a generic tool. So *all* questions of the appropriateness of the data appearing in the correctness PO (whether forward or backward) need to be thoroughly validated at the domain level. The analogy with retrenchment is necessary, since any interworking between retrenchment and refinement is only able to treat formally, those aspects of the models that both retrenchment and refinement are able to regard as formal.

5. Retrenching the Balance Enquiry Quandary

In this section we show how retrenchment can handle the issues raised by the Balance Enquiry Quandary. We present the backward retrenchment account, and then, more briefly, how forward retrenchment would view the situation, looking ahead to Section 7.

We construct a backward retrenchment. Since all operations other than *ToBalanceEnquiry* are unproblematically refinements, the within, output, and concedes relations for them (in the retrenchment) will default to (schemas of appropriate signatures with predicate parts given by) $\mathsf{true}, \mathsf{true}, \mathsf{false}$ respectively. This makes the backward retrenchment correctness PO (9) reduce to the backward refinement one (3). In the absence of any nontrivial applicability issues, there is no departure from refinement for these other operations. For the *ToBalanceEnquiry*s, it is sufficient to have only the output relation be anything other than the defaults just mentioned:[22]

[21] We only consider 'infinite into the future' since 'infinite into the past' cannot even be embarked upon.

[22] The quantification over the *from* variables in $Q_{bb,ToBalanceEnquiry}$ corresponds to the fact that the *from* variables are inaccessible to the *to* purse. This reflects a tacit assumption that the frame of any retrenchment data pertaining to an operation should not exceed the variables accessible to the agent performing it. However this assumption is not mathematically indispensable.

$O_{ab,ToBalanceEnquiry}$

$AbWorld'$
$BetweenWorld'$
$Abal! : \mathbb{N}$
$Bbal! : \mathbb{N}$

$(\exists\, Bfromstatus', Bfrompaydetails' \bullet$
$\quad (\neg(Btostatus' = Bepv \land Bfromstatus' = Bepa \ \land Bfrompaydetails' = Btopaydetails')$
$\qquad \land\ Abal! = Bbal!) \quad \lor$
$\quad ((Btostatus' = Bepv \land Bfromstatus' = Bepa \ \land Bfrompaydetails' = Btopaydetails')$
$\qquad \land\ Abal! - Bbal! = Atobalance' - Btobalance' = Btopaydetails'))$

It is clear that this enables us to easily discharge (9). It says that either the critical case ($Bepv/Bepa$ global state plus $Bfrompaydetails' = Btopaydetails'$) holds or not. If it does not, then all is well. If it does, then it states that the dissonance in the outputs produced by the abstract and concrete versions is not arbitrary, but is tightly related to the temporary dissonance at that point, between the abstract and concrete *to* purse balances. Moreover it is already implicit that the said dissonance is entirely justifiable in the $Bepv/Bepa$ global state, since precisely the same situation arises at that point in the refinement of the protocol (that refinement being entirely unproblematic in itself), in the absence of any *BalanceEnquiry* operations at all. This constitutes a justification of what the two balance enquiry operations do at the given point in the protocol, and thus, validates the retrenchment used to account for the facts.

Going further, a little reflection shows that the policy on outputs that is expressed by the relation $O_{ab,ToBalanceEnquiry}$, when extended pointwise across all pairs of occurrences of *AtoBalanceEnquiry* and *BtoBalanceEnquiry* outputs in a run, is appropriate as an output finalisation policy, although since it involves the states, conventional restrictions on the variables that can occur in an output finalisation, prevent it being adopted as such in the framework of [27]. (The preceding recognises that no other abstract operation produces any output in our models, thus precluding any non-trivial $ROut_{ab,Op}$ for the other *Op*s in them.) Thus it is the restrictions of the formalism of [27] itself which are partly responsible for the awkwardness of the Balance Enquiry Quandary. We return to this point in the next section.

Let us comment briefly on the forward direction. Since from the state point of view, the operations *AtoBalanceEnquiry* and *BtoBalanceEnquiry* are skips, for every possible way of satisfying the backward PO (9), there will be a corresponding way of satisfying the forward PO (8), given by interchanging primed and unprimed state components. Therefore, given a suitable augmentation of the signature of $O_{ab,ToBalanceEnquiry}$, the same predicate part for the schema will do duty in (8), as sufficed for the backward PO (9). We return to this in Section 7.

6. A Generalised Forward Refinement Account of the Balance Enquiry Quandary

Generalised forward refinement was first introduced in the context of the ASM system development technique; see [17, 18, 25, 19, 37, 38]. Of course the underlying ideas of such a refinement technique are not tied to the specific details of the ASM syntax, and can be readily applied to any model based formal development technique utilising states and operations. In this section we apply it to our Mondex models, to see what insights it can contribute to the Balance Enquiry Quandary.

Generalised forward refinement works by relating a sequence of zero or more steps of an abstract model of a system *en bloc*, to a sequence of zero or more steps of a concrete model *en bloc*, provided, obviously, that there is at least one step in total. The objective is that a retrieve relation on the states is preserved between the ends of the two sequences, even if it is not necessarily preserved at points internal to the two sequences. For obvious reasons, such pairs of sequences are called (m,n) pairs.

To build up the usual kind of inductive argument over traces, (m,n) pairs are required to witness the generalisation of the forward correctness PO (6), to the case where the *BOp* and *AOp* appearing in (6) refer to several transitions (or none) instead of just one. To achieve a successful generalised forward refinement, one must therefore identify enough (m,n) pairs to enable any concrete run to be simulated. Specifically, one must be able to cut up an arbitrary concrete run into suitable pieces, and for each concrete piece one must find a corresponding (m,n) pair. The pair must be such that the concrete piece forms the concrete component of the pair, and moreover, the abstract component of the pair must have an (abstract) initial state that matches the final (abstract) state of the induction over traces established thus far, so that the new (m,n) pair abuts successfully to the simulation so far. In this manner, one constructs an abstract run that simulates the concrete run by re-establishing the retrieve relation periodically. The fact that the retrieve relation only needs to be re-established periodically, and not at every step (and that the numbers of abstract and concrete steps need not match) can lead to a simpler overall relationship between abstract and concrete models. This is all brought out in our detailed calculations below.

Here is the retrieve relation we work with in this section:

$R_{GFR,ab}$

$$
\begin{array}{l}
AbWorld \\
BetweenWorld \\
\hline
Abfromlost = \Sigma Bdefinitelylost + \\
\quad (\textsf{if } Btostatus = Bepv \land Bfromstatus = Bepa \land Bfrompaydetails = Btopaydetails \\
\quad \textsf{then } Bfrompaydetails \textsf{ else } 0) \\
Abtolost = 0 \\
Abfrombalance = Bfrombalance \\
Abtobalance = Btobalance
\end{array}
$$

We note that this is simpler than R_{ab} due to the greater flexibility that generalised forward refinement affords us. Essentially, in the critical $Btostatus = Bepv \land Bfromstatus = Bepa \land Bfrompaydetails = Btopaydetails$ state, we can assert that the payment (which is in flight, and may or may not eventually arrive) has definitively been lost. This is because if the payment eventually arrives, this intermediate critical state becomes internal to the relevant (m,n) sequences, and the retrieve relation $R_{GFR,ab}$ is no longer obliged to make an accurate pronouncement about that state of affairs. Note how this contrasts with the backward refinement, in which the retrieve relation R_{ab} has to make an accurate statement about all intermediate points of the protocol, within a $(1,1)$ refinement structure. The price to be paid for the simplicity gained, is a careful analysis of what are the needed (m,n) pairs, so that every possible concrete run is catered for.

To get a handle on the (m,n) pairs, we will firstly disregard all *BIgnore*s since they merely skip. Next, we argue as follows. A *BfromAckPurseOkay* operation has to have been preceded by a *BtoValPurseOkay* operation by nonforgeability arguments. Similarly a *BtoValPurseOkay* operation has to have been preceded by a *BfromReqPurseOkay* operation, and a *BfromReqPurseOkay* operation has to have been pre-

ceded by a *BStartFromPurseEafromOkay* and a *BStartToPurseEafromOkay* in either order. This gives two total orders for a run through to acknowledgement for the protocol. Thus any trial of the protocol contains some maximal prefix of one or other order as a subsequence. We call the relevant prefix the *core prefix*.

The *BStartFromPurseEafromOkay* and *BStartToPurseEafromOkay* themselves have to be preceded by a *BfromAbort* and a *BtoAbort* respectively. For technical convenience we will regard the *BfromAbort* and *BtoAbort* as terminating the *preceding* protocol trial, rather than initiating the current one. If we do this we must make special arrangements for the first and last trials in the system run.

For the first trial, by initialisation, the initiating *BfromAbort* and *BtoAbort* are equivalent to skips and may be ignored. For the last trial, one or other of *BfromAbort* or *BtoAbort* may simply not be present after the core prefix; we deal with this later.

The picture so far of a concrete system run is causally equivalent to a sequence of core prefixes for protocol trials terminated by two aborts, followed by a last core prefix not necessarily terminated by any aborts. (N.B. The caveat rejoining causal equivalence is nontrivial, since an indolent *BfromPurse* may still be waiting for an *ack* for one transaction, while an impatient *BtoPurse* may have already started on the next. The *Brfrompaydetails* $=$ *Btopaydetails* simplified restriction in $R_{GFR,ab}$ (and the sequence number check in the real protocol) enables overlapping transactions to be accommodated if desired.)

In between two successive core prefixes, we can have any finite number of aborts; since *BfromAbort* and *BtoAbort* are both idempotent operations, several aborts can be reduced to just one of each. Into this structure, we can interleave *BtoBalanceEnquiry*s at will.

We can now list the concrete execution fragments we need to consider to ensure every concrete run is covered. They are just the core prefixes, either abort-terminated or not, with arbitrary numbers of occurrences of *BtoBalanceEnquiry* interleaved into them. For each we give the abstract execution fragments that simulate them in making up the (m, n) pairs needed for a successful generalised forward refinement.

(0) Empty core prefix: This is a generalised forward refinement of as many *AbToBalanceEnquiry*s, as *BtoBalanceEnquiry*s in the concrete sequence.

(1) *BStartFromPurseEafromOkay*; or *BStartToPurseEafromOkay*; or both *Start* operations (in each case terminated by zero, one or two aborts): All these sequences are generalised forward refinements of a number of *AbToBalanceEnquiry*s, equal to the number of *BtoBalanceEnquiry*s interleaved into the concrete sequence.

(2) Both *Start* operations followed by *BfromReqPurseOkay* (terminated by zero, one or two aborts): This is a generalised forward refinement of as many *AbToBalanceEnquiry*s interleaved around *AbTransferLost*, as there were *BtoBalanceEnquiry*s interleaved into the concrete sequence.

(3) Both *Start* operations followed by *BfromReqPurseOkay* followed by *BtoValPurseOkay* (terminated by zero, one or two aborts): This is a generalised forward refinement of *k AbToBalanceEnquiry*s followed by *AbTransferOkay* followed by *h AbToBalanceEnquiry*s, where *k BtoBalanceEnquiry*s were interleaved before *BtoValPurseOkay*, and *h BtoBalanceEnquiry*s were interleaved after it, in the concrete sequence.

(4) Both *Start* operations followed by *BfromReqPurseOkay* followed by *BtoValPurseOkay* followed by *BfromAckPurseOkay* (terminated by zero, one or two aborts): This is a generalised forward refinement of *k AbToBalanceEnquiry*s followed by *AbTransferOkay* followed by *h AbToBalanceEnquiry*s, where *k BtoBalanceEnquiry*s were interleaved before *BtoValPurseOkay*, and *h BtoBalanceEnquiry*s were inter-

leaved after it, in the concrete sequence.

Reflection upon the above makes two things clear. Firstly, that if we were also observing the *from* balance as well as the *to* balance, we would have to partition *from* balance enquiries in cases (2)-(4) according to which enquiries preceded or followed *BfromReqPurseOkay*, just as we needed to partition cases (3)-(4) according to *BtoValPurseOkay*. Secondly, the if clause in $R_{GFR,ab}$ is only needed to cater for a last core prefix not terminated by suitable aborts; it is not used for abort-terminated ones.

These points highlight the fact that when the concrete runs of the system are under the control of an environment which is able to drastically curtail the number of concrete fragments to take into account for constructing (m, n) pairs, generalised forward refinement can yield a real bonus in simplifying the work needed to show refinement. However in a case like ours, in which the environment must be assumed uncooperative, nay hostile, a combinatorial explosion of possibilities can easily arise, when the application can execute in a genuinely concurrent manner. The explosion can arise not only due to the many causally equivalent interleavings of the activity of the system of interest, but also due to the interleavings of this activity with the activity of parts of the system that are independent of it but running concurrently. All this requires careful analysis to keep the refinement sound, but is completely avoided within a $(1, 1)$ refinement perspective.[23]

A related issue arises from engineering considerations. As noted near the beginning of this paper, refinements can leave out certain aspects of the real system, and system designs themselves can evolve. In the light of this, the $(1, 1)$ and (m, n) approaches can exhibit different risks regarding the robustness of the refinement in the face of changes in the system model, whether conceptual and arising from implementation, or more consciously via a design change. On the one hand, a $(1, 1)$ approach can be more robust in the face of a system change that generates more runs involving the considered operations, since the refinement of each operation is a self contained problem. On the other hand, an (m, n) approach can be more robust in the face of a system change that complicates the system state, since its invariant is likely to be simpler, and is likely to involve less of the state, thus being more likely to be decoupled from any subsequent state enrichment; compare $R_{GFR,ab}$ (in the ideal case, without the if clause) with R_{ab}. The whole question of the relative merits of the two approaches in the face of evolving engineering risk deserves further consideration.

7. Nondeterminism Resolution, and $(1, 1)$ Forward Refinement

In Section 4.3 we outlined the Mondex backward refinement, and in Section 6 we gave an alternative treatment of the protocol in terms of generalised forward refinement. Would it not be possible to get the best of both worlds, getting thereby a $(1, 1)$ forward refinement, by moving the point of resolution of abstract nondeterminism in the refinement so that it coincided with the corresponding point of resolution of concrete nondeterminism? The simplicity of the retrenchment resolution of the balance enquiry quandary encourages us to suppose this can be done, but to achieve it necessitates appreciating a number of issues.

Firstly, it is not just the point of resolution of nondeterminism that is of interest; the point of introduction of nondeterminism is just as important. Here we see a crucial distinction between abstract

[23] In our case, we did not actually see such an explosion, owing to the essentially sequential nature of the protocol, and to the restriction to only two purses.

and concrete. In the abstract world, nondeterminism is introduced and resolved within the same event,[24] the *AbTransfer* operation, which has two incompatible outcomes. However concrete nondeterminism is more concerned with which operation is elected to happen next, and in fact concrete operations themselves are almost completely deterministic. (Thus we observe that whenever there are overlapping guards in any operation, only one of them does anything nontrivial; and whenever there is a disjunction in the body of a bottom level schema, the guards are disjoint, reducing the disjunction to mere packaging rather than an expression of uncertainty in principle.[25])

The determinism is driven by the desire in Mondex to have 'All value accounted', which is underpinned by a largely unstated invariant permeating both abstract and concrete models: namely that 'money can be neither created nor destroyed' within our models.[26] So if all money is conserved, then if one maintains enough state in the models to support a provably correct protocol, the whereabouts of all the money can be tracked. The Mondex concrete model does indeed maintain enough state to do this.

In a sense, concrete nondeterminism is introduced in the two *Start* events, since if the next event is *BtoAbort*, then the transaction is already doomed, even though, as if in some Wagnerian tragedy, several further acts have to ensue[27] before everyone is lying dead on the stage.

Concrete nondeterminism (as regards the transaction as a whole) is resolved either quite early (as just described), or late (when the transaction succeeds), or at an intermediate point of failure. Beyond this fundamental nondeterminism, there is nondeterminism regarding how many unproductive events occur for a transaction that has already failed in principle, and the nondeterminism of interleaving of independent operations.

The reduction of concrete nondeterminism to the choice of next event, each one of which is essentially deterministic, has a salutary effect on the prospects for forward refinement. This is because the remit of the correctness PO, (6), is just a single operation. Thus one can match the choice of next operation to perform during a run, with the choice of operation PO for refining the resulting execution step, and can hope that this will be enough to account for all the nondeterminism in the models.

Encouraged by the above considerations regarding forward refinement, deciding how to treat successful transactions is easy: we just have to make *AbTransferOkay* refine to *BtoValPurseOkay*, as that is the moment that the value successfully arrives. What to do about failing ones is a little trickier, as the care over double counting in the *from* and *to* purse logs in Section 4.2 demonstrated. Thus any abort event (and it has to be a *logging* abort event) is either the start of the abortion process, or its completion. Since two such aborts are required to properly confound a transaction, we use the 'Last man out switch off the lights!' principle, and make the second abort event a refinement of *AbTransferLost*. Unfortunately neither agent in the transaction knows which part of the abortion process any particular abort event that it performs is, or indeed whether it is an isolated, and therefore harmless abort. What distinguishes these cases is state information in the *other* agent. Accordingly, is is sufficient to decompose abort operations into *..Benign*, *..Start*, and *..Complete* suboperations, on the basis of this further state information (since it is in principle available to an all-knowing global refinement relationship), even though all that

[24] In this section, an event is an occurrence of an operation within a run. Where the operation is nondeterministic, this amounts to a choice of one of its possible outcomes.

[25] This view is supported by the fact that in [46], the reasoning that establishes the refinement is done using (the counterpart of) R_{abCl}, exposing *chosenlost*, rather than R_{ab} itself.

[26] The behaviour of central banks lies outside our discourse.

[27] At worst: the *req* message arrives; a *BfromReqOkay* event sends a *val* message; it arrives; it is discarded by a *to* purse *BIgnore*; eventually the *from* purse *BfromAbort*s.

any individual agent knows, is that he is executing one or other of the cases, without knowing which it is. As ever, disjunction does the job of providing the more subtle packaging required here.[28] We thus decompose *BfromAbort* and *BtoAbort* as follows.

BfromAbort

$$BfromAbortBenign \lor BfromAbortStart \lor BfromAbortComplete$$

BfromAbortBenign

$$\Delta BetweenWorld$$
$$Bfromstatus \neq Bepa$$
$$Bfromstatus' = BeaFrom$$
RestSame.....

BfromAbortStart

$$\Delta BetweenWorld$$
$$Bfromstatus = Bepa$$
$$Bfromstatus' = BeaFrom$$
$$Bfrompaydetails \notin Btoexlog$$
$$Bfromexlog' = Bfromexlog \cup \{Bfrompaydetails\}$$
RestSame.....

BfromAbortComplete

$$\Delta BetweenWorld$$
$$Bfromstatus = Bepa$$
$$Bfromstatus' = BeaFrom$$
$$Bfrompaydetails \in Btoexlog$$
$$Bfromexlog' = Bfromexlog \cup \{Bfrompaydetails\}$$
RestSame.....

and

BtoAbort

$$BtoAbortBenign \lor BtoAbortStart \lor BtoAbortComplete$$

BtoAbortBenign

$$\Delta BetweenWorld$$
$$Btostatus \neq Bepv$$
$$Btostatus' = BeaFrom$$
RestSame.....

BtoAbortStart

$$\Delta BetweenWorld$$
$$Btostatus = Bepv$$
$$Btostatus' = BeaFrom$$
$$Btopaydetails \notin Bfromexlog$$
$$Btoexlog' = Btoexlog \cup \{Btopaydetails\}$$
RestSame.....

BtoAbortComplete

$$\Delta BetweenWorld$$
$$Btostatus = Bepv$$
$$Btostatus' = BeaFrom$$
$$Btopaydetails \in Bfromexlog$$
$$Btoexlog' = Btoexlog \cup \{Btopaydetails\}$$
RestSame.....

Note that as this is no more than a partitioning into disjointly guarded suboperations, no fresh non-determinism is introduced.[29] Note also that, although the preconditions in the separate suboperations *B..AbortStart* and *B..AbortComplete* refer to conditions across two different purses, the actual state

[28] Note that as far as any agent is concerned, it is nonconstructive, classical, disjunction.

[29] It is in fact just a continuation of the process started in Section 4, since in [46], the equivalent operation would correspond to $BAbort = BfromAbort \lor BtoAbort$.

changes are identical, and so can be implemented locally in individual purses without any need for (unavailable) global information.

The preceding permits us to achieve a $(1,1)$ forward refinement by carefully matching the points of resolution of nondeterminism in the two models. The price to be paid, is that the retrieve relation $R_{NR,ab}$ becomes more complicated, though not significantly more so than in the backward case if one were to unpack the *Bdefinitelylost* and *Bmaybelost* variables in the latter. $R_{NR,ab}$ itself is a simple default $R_{NR,ab,def}$, which is overridden by four special cases ($R_{NR,ab,(a,\overline{v})}$, $R_{NR,ab,(\overline{a},v)}$, $R_{NR,ab,(a=v)}$, $R_{NR,ab,(a\neq v)}$) that cover particular points in the protocol when the value is in flight.

Thus $R_{NR,ab,(a,\overline{v})}$ covers the case where the *from* purse is still waiting for its *ack*, while the *to* purse has finished, and is not in any transaction. $R_{NR,ab,(\overline{a},v)}$ covers the case where the *from* purse is not in any transaction, while the *to* purse has started a new transaction and is waiting for the *val* to arrive. $R_{NR,ab,(a=v)}$ covers the case where the two purses are in the middle of a given transaction, and the value is actually in flight. $R_{NR,ab,(a\neq v)}$ covers the case where the two purses are in their critical states, but the value has arrived and is safely in the *to* purse's balance; the *from* purse is still waiting for the *ack* to arrive, while the *to* purse has started a new transaction and is waiting for the new *val* to arrive; this case is distinguished from the preceding one by the different paydetails of the two transactions.

The good news is that despite the protocol having many more possible states, only these four need to be singled out in the retrieve relation. To clarify the interpretation of the overriding, the concrete state is to be regarded as the domain of $R_{NR,ab}$ and the abstract state as the codomain in the following:

$$R_{NR,ab} \mathrel{\widehat{=}} R_{NR,ab,def} \oplus \left(R_{NR,ab,(a,\overline{v})} \vee R_{NR,ab,(\overline{a},v)} \vee R_{NR,ab,(a=v)} \vee R_{NR,ab,(a\neq v)}\right)$$

where

$R_{NR,ab,core}$

$$\begin{array}{l} AbWorld \\ BetweenWorld \\ \hline Abfromlost = \Sigma\,(Bfromexlog \cap Btoexlog) \\ Abtolost = 0 \\ Abtobalance = Btobalance \end{array}$$

and

$R_{NR,ab,def}$

$$\begin{array}{l} R_{NR,ab,core} \\ \hline Abfrombalance = Bfrombalance \end{array}$$

$R_{NR,ab,(a,\overline{v})}$

$$\begin{array}{l} R_{NR,ab,core} \\ \hline Bfromstatus = Bepa \\ Btostatus \neq Bepv \\ Bfrompaydetails \in Btoexlog \\ Abfrombalance = Bfrombalance + \\ \qquad\qquad Bfrompaydetails \end{array}$$

$R_{NR,ab,(\overline{a},v)}$

$$\begin{array}{l} R_{NR,ab,core} \\ \hline Bfromstatus \neq Bepa \\ Btostatus = Bepv \\ Btopaydetails \in Bfromexlog \\ Abfrombalance = Bfrombalance + \\ \qquad\qquad Btopaydetails \end{array}$$

$R_{NR,ab,(a=v)}$

$R_{NR,ab,core}$
$Bfromstatus = Bepa$
$Btostatus = Bepv$
$Bfrompaydetails = Btopaydetails$
$Abfrombalance = Bfrombalance + Bfrompaydetails$

$R_{NR,ab,(a\neq v)}$

$R_{NR,ab,core}$
$Bfromstatus = Bepa$
$Btostatus = Bepv$
$Bfrompaydetails \neq Btopaydetails$
$Bfrompaydetails \in Btoexlog$
$Abfrombalance = Bfrombalance + Bfrompaydetails$

Note that $R_{NR,ab}$ is a completely deterministic relation, indeed a function from concrete to abstract, unlike R_{ab} in the backward refinement. We can now briefly discuss the refinements of the various (sub)operations.

Initialisation: Trivial as before. $R_{NR,ab,def}$ is established.

BIgnore, ***BfromAbortBenign***, ***BtoAbortBenign*** all refine ***AbIgnore***: *BIgnore* skips, as does *AbIgnore*. For *BfromAbortBenign*, only $R_{NR,ab,def}$, or $R_{NR,ab,(\overline{a},v)}$ can hold in the before state. Now *BfromAbortBenign* modifies *Bfromstatus* in a way that does not affect the truth of either of these. For *BtoAbortBenign*, only $R_{NR,ab,def}$, or $R_{NR,ab,(a,\overline{v})}$ can hold in the before state. Now *BtoAbortBenign* modifies *Btostatus* in a way that does not affect the truth of either of these.

BStartFromPurseEafromOkay and ***BStartToPurseEafromOkay*** both refine ***AbIgnore***: For operation *BStartFromPurseEafromOkay*, only $R_{NR,ab,def}$, or $R_{NR,ab,(\overline{a},v)}$ can hold in the before state. Noting that the variables changed by *BStartFromPurseEafromOkay* are *Bfromstatus* and *Bfrompaydetails*, for $R_{NR,ab,(\overline{a},v)}$, the change in *Bfromstatus* leaves clause $Bfromstatus \neq Bepa$ invariant, and *Bfrompaydetails* is not even mentioned, so $R_{NR,ab,(\overline{a},v)}$ is invariant under the operation. If $R_{NR,ab,def}$ holds in the before state (and is not overridden by $R_{NR,ab,(\overline{a},v)}$), then since the variables changed are not mentioned in $R_{NR,ab,def}$, it remains invariant under the operation, and is also not overridden, by the invariance of $R_{NR,ab,(\overline{a},v)}$.

For *BStartToPurseEafromOkay*, only $R_{NR,ab,def}$, or $R_{NR,ab,(a,\overline{v})}$ can hold in the before state. Consider the $R_{NR,ab,(a,\overline{v})}$ case. Noting that the variables changed by *BStartToPurseEafromOkay* are *Btostatus* and *Btopaydetails*, for $R_{NR,ab,(a,\overline{v})}$, the change in *Btostatus* causes one of $R_{NR,ab,(a=v)}$ or $R_{NR,ab,(a\neq v)}$ to become true. In fact $R_{NR,ab,(a=v)}$ is excluded because $Btopaydetails'$ refers to a new transaction, and $Bfromstatus = Bepa$ shows that the *from* purse must still be engaged in an earlier transaction, since the *from* purse only goes into the *Bepa* state for the same transaction as the *to* purse upon receipt of the *req* message, which is only just being dispatched, and so cannot yet have arrived at the *from* purse. This forces $Bfrompaydetails' \neq Btopaydetails'$. Alternatively, suppose $R_{NR,ab,def}$ holds in the before state (and is not overridden by $R_{NR,ab,(a,\overline{v})}$). Then since the variables changed are not mentioned in $R_{NR,ab,def}$, it remains invariant under the operation, but we must check that it is not overridden by either of $R_{NR,ab,(a=v)}$ or $R_{NR,ab,(a\neq v)}$ in the after state. Now $R'_{NR,ab,(a=v)}$ is excluded since for $Bfromstatus' = Bepa$ and $Bfrompaydetails' = Btopaydetails'$ to be true, the *from* purse must have already received and processed the *req* message that the *to* purse is only just sending out in the current operation. $R'_{NR,ab,(a\neq v)}$ is excluded since it would imply that $R_{NR,ab,(a,\overline{v})}$ held in the before state, overriding $R_{NR,ab,def}$, which we assumed not to be the case.

BfromReqPurseOkay refines ***AbIgnore***: By unforgeability reasoning, we know that a *req* message containing the value in *Bfrompaydetails* will have been received by *BfromPurse*. Ostensibly, $R_{NR,ab,(\overline{a},v)}$

might hold in the before state. However, this would require $Btopaydetails \in Bfromexlog$. This in turn would require *BfromPurse* to have aborted *BtoPurse*'s current transaction, which is the same as or later than *BfromPurse*'s current transaction (since the *req* message identifies a single transaction in both purses, and *BtoPurse* sent the message earlier than *BfromPurse* received it). But *BfromPurse* can only have aborted some earlier transaction, so this state of affairs is impossible. So only $R_{NR,ab,def}$ may hold in the before state. The change in *Bfromstatus* and in *Bfrombalance* caused by the operation then makes $R_{NR,ab,(a\neq v)}$ true (or $R_{NR,ab,(a=v)}$ true), depending on whether since the start of *BfromPurse*'s current transaction, *BtoPurse* has (or has not, respectively) aborted and started a fresh transaction.

BtoValPurseOkay refines ***AbTransferOkay***: By unforgeability reasoning, a *val* message containing the value in *Btopaydetails* will have been received by *BtoPurse*. In the before state, either $R_{NR,ab,(\overline{a},v)}$ or $R_{NR,ab,(a=v)}$ can hold, depending on whether since sending the *val* message, *BfromPurse* has (or has not, respectively) aborted and started a fresh transaction. In either case the changes in *Btostatus* and in *Btobalance* match the changes in *Abtobalance* to establish $R_{NR,ab,def}$. (In the $R_{NR,ab,(a=v)}$ case, since the *to* purse is not aborting, $R_{NR,ab,(a,\overline{v})}$ cannot be established in the after state.)

BfromAckPurseOkay refines ***AbIgnore***: By unforgeability reasoning, an *ack* message containing the value in *Bfrompaydetails* will have been received by *BfromPurse*. Ostensibly, $R_{NR,ab,(a\neq v)}$ or $R_{NR,ab,(a,\overline{v})}$ might hold in the before state. However, either of these would require *BtoPurse* to have both aborted-and-logged *BfromPurse*'s current transaction, and also sent the *ack* message, which is a contradiction. Therefore only $R_{NR,ab,def}$ can hold in the before state, is preserved by the operation, and cannot be overridden in the after state, since only a benign change in *Bfromstatus* took place.

BfromAbortStart and ***BtoAbortStart*** refine ***AbIgnore***: For *BfromAbortStart*, only either $R_{NR,ab,def}$ or $R_{NR,ab,(a=v)}$ can hold in the before state. (The precondition $Bfrompaydetails \notin Btoexlog$ precludes $R_{NR,ab,(a,\overline{v})}$ and $R_{NR,ab,(a\neq v)}$ in the before state. $R_{NR,ab,(\overline{a},v)}$ is obviously precluded.) Now $R_{NR,ab,(a=v)}$ is transformed into $R_{NR,ab,(\overline{a},v)}$ by the operation. Suppose $R_{NR,ab,def}$ holds in the before state and is not overridden. Then it is invariant, but we much check that it is not overridden by $R_{NR,ab,(\overline{a},v)}$ in the after state. However the latter would imply that if $Bfrompaydetails = Btopaydetails$ held in the before state, then $R_{NR,ab,(a=v)}$ would also have to hold in the before state, which we assumed false. And if $Bfrompaydetails \neq Btopaydetails$ in the before state, the only candidates are excluded by the precondition mentioned above.

For *BtoAbortStart*, only $R_{NR,ab,def}$, or $R_{NR,ab,(a=v)}$, or $R_{NR,ab,(a\neq v)}$ can hold in the before state. (The precondition $Btopaydetails \notin Bfromexlog$ precludes $R_{NR,ab,(\overline{a},v)}$ in the before state. $R_{NR,ab,(a,\overline{v})}$ is obviously precluded.) Now $R_{NR,ab,(a=v)}$ and $R_{NR,ab,(a\neq v)}$ are both transformed into $R_{NR,ab,(a,\overline{v})}$ by the operation. Suppose $R_{NR,ab,def}$ holds in the before state and is not overridden. Then it is invariant, but as ever, we much check that it is not overridden by $R_{NR,ab,(a,\overline{v})}$ (the only candidate) in the after state. Suppose it was. Then we would have $Bfrompaydetails \in Btoexlog'$, and so $Bfrompaydetails \in Btoexlog$ would hold for the before state. Since $Btostatus = Bepv$ in the before state, either $R_{NR,ab,(a\neq v)}$ would have held in the before state (if $Bfrompaydetails \neq Btopaydetails$), contradicting our assumption, or $R_{NR,ab,(a=v)}$ would have held in the before state (if $Bfrompaydetails = Btopaydetails$), also contradicting our assumption.

BfromAbortComplete and ***BtoAbortComplete*** refine ***AbTransferLost***: For *BfromAbortComplete*, only $R_{NR,ab,(a,\overline{v})}$ or $R_{NR,ab,(a\neq v)}$ can hold in the before state. ($R_{NR,ab,(a=v)}$ is precluded, since the precondition $Bfrompaydetails \in Btoexlog$ means that the *to* purse has aborted the *from* purse's current transaction, and the *to* purse's *Bepv* state must refer to a newer transaction.) The operation transforms

$R_{NR,ab,(a,\overline{v})}$ into $R_{NR,ab,def}$ (given the effect of *AbTransferLost*), which is easily seen to not be overridden because $Bfromstatus' \neq Bepa$ and $Btostatus' \neq Bepv$. The operation also transforms $R_{NR,ab,(a\neq v)}$ into $R_{NR,ab,def}$ (given the effect of *AbTransferLost*). This is also not overridden because $Bfrompaydetails \neq Btopaydetails$, and the fact that *Btopaydetails* refers to a newer transaction as we argued above, means that *Btopaydetails* cannot be in $Bfromexlog'$, ruling out $R_{NR,ab,(\overline{a},v)}$, the only conceivable possibility, in the after state.

For *BtoAbortComplete*, only $R_{NR,ab,(\overline{a},v)}$ can hold in the before state. ($R_{NR,ab,(a=v)}$ and $R_{NR,ab,(a\neq v)}$ are precluded, since the precondition $Btopaydetails \in Bfromexlog$ means that the *from* purse has aborted the *to* purse's current transaction, exiting the *Bepa* state for the *to* purse's current transaction, and cannot have reached the *Bepa* state for a new transaction without a *req* message from the *to* purse, which evidently has not been sent.) The operation transforms $R_{NR,ab,(\overline{a},v)}$ into $R_{NR,ab,def}$ (given the effect of *AbTransferLost*), which is easily seen to not be overridden because $Bfromstatus' \neq Bepa$ and $Btostatus' \neq Bepv$.

We see that all the concrete suboperations of the protocol are refinements of abstract operations. It is now easy to package them up into the original concrete operations of the protocol, and thus to arrive at a $(1,1)$ forward refinement of the abstract system, via disjunctions of the above suboperation refinements. The technical details are uninteresting so we omit them.

Beyond this, one can ask how a balance enquiry fares in the above scenario. Evidently, abstract and concrete balance enquiries for the *to* purse will always give answers in agreement, since the two *to* balances agree in all components of $R_{NR,ab}$, some case of which always holds. This is a win for the forward refinement formulation, but of course, it comes at a price. Since we still fundamentally have an atomic operation refined to a non-atomic one, there will always be a discrepancy in balances while the value is in flight. This time the discrepancy shows up in the *from* purse's abstract and concrete balances any time $R_{NR,ab,def}$ fails to hold, as the details of the non-$R_{NR,ab,def}$ cases of the retrieve relation make plain. Thus the corresponding enquiry operations for the *from* purse will disagree if they are invoked at the relevant moments. Furthermore, the breakdown of the refinement that this signals is in a way more acute than in the backward case, since it is the operation PO (6) itself that fails, via a the failure of a nontrivial $ROut_{ab,Op}$ in the consequent, rather than the mere failure of an environmental assumption such as output finalisation.

Nevertheless, it is clear from all that has gone before that a retrenchment output relation very similar to $O_{ab,ToBalanceEnquiry}$ (but this time constructed for a forward retrenchment, as indicated in Section 5), will be able to handle the situation, with exactly the same benefits as in the backward case case:

$O_{NR,ab,FromBalanceEnquiry}$

$$\Delta AbWorld$$
$$\Delta BetweenWorld$$
$$Abal! : \mathbb{N}$$
$$Bbal! : \mathbb{N}$$

$$(\exists\, Btostatus', Btopaydetails' \bullet$$
$$(\neg(Btostatus' = Bepv \land Bfromstatus' = Bepa \land Bfrompaydetails' = Btopaydetails')$$
$$\land\, Abal! = Bbal!) \quad \lor$$
$$((Btostatus' = Bepv \land Bfromstatus' = Bepa \land Bfrompaydetails' = Btopaydetails')$$
$$\land\, Abal! - Bbal! = Afrombalance' - Bfrombalance' = Bfrompaydetails'))$$

We close this section with a brief comparison of the forward and backward $(1,1)$ refinements. The backward refinement had the undoubted virtue of conceptual simplicity. A single concrete operation accounted for all the nondeterminism latent in the abstract model. On the other hand, the fact that the relevant concrete operation was itself deterministic, led to some subtlety and nondeterminism in the retrieve relation, to account for the differing possible eventual outcomes. The forward refinement did not exhibit the latter aspects, but in contrast, the different abstract outcomes were distributed around different concrete operations and suboperations in quite a subtle way. Then again, the retrieve relation was pleasingly functional.

8. Conclusions

In the preceding sections, we started by surveying some of the systems engineering motivations for the introduction of retrenchment. We pointed out how systems engineering in the real world is often constrained by considerations that a pure research based environment can choose to neglect, and that this can drive a wedge between technical approaches that work in the research environment and what can be achieved in real engineering situations.

We then moved on to consider the Mondex development in detail. We surveyed the 'retrenchment opportunities' that Mondex offers as a result of having taken pragmatic decisions on how certain aspects of the development were to be handled, given the demands of refinement. We outlined how the *Tower Pattern* of retrenchment could address many of these. We then turned our attention to the 'Balance Enquiry Quandary', a retrenchment opportunity that requires the consideration of the model B protocol in its entirety, and thus falls outside the scope of the *Tower Pattern* as applied in the other cases. We went into fair detail to describe how the refinement in [46] fared as regards the Balance Enquiry operation, and the rather subtle way that it failed, the subtlety being mainly attributable to the backward nature of the refinement of [46].

The Balance Enquiry Quandary could in fact be solved simply by pre-aborting during balance enquiry operations. This would require any running transaction to be terminated before the enquiry takes place, thus resolving the non-determinism in a known way (to *lost*). This is an example of how changing the specification could remove the need for retrenchment. But, in the Mondex project, this particular change request from the formal methods team was rejected by the customer, on separate commercial grounds, the reasons for which were outlined at the end of Section 4.3. This is therefore also an example of how changing the specification to remove the need for retrenchment is not always an available option in real developments, as noted in the Introduction.

We went on to show how a formal account of the quandary could nevertheless be given via retrenchment. This retrenchment is just like the corresponding refinement, except for the one rogue balance enquiry operation, which can be smoothly handled via a suitable retrenchment output relation.

We then reconsidered the whole situation using generalised forward refinement, which yielded a relatively straightforward solution by hiding the awkward details inside the existential quantifications of suitable (m,n) pairs. The simplicity of both the retrenchment and generalised forward refinement approaches was the spur to the development of a genuine $(1,1)$ forward refinement for Mondex, something long thought impossible.

We remark that in [39], the authors perform their mechanised proof using a single refinement step, mimicking the broad strategy of [46] in an ASM framework, and using some (0,1) diagrams to remove

the need for skips. The rapid completion of the mechanisation, performed from scratch in a month or so, strongly suggests that the proofs of the novel refinements and retrenchments for Mondex outlined in this paper could be mechanised in an even shorter time by a team with the accumulated expertise of [39]. Perhaps the only less than straightforward part of such an undertaking might concern what we refered to as 'unforgeability reasoning', where, in order to save space and reduce the complexity of the main arguments, we made use of properties of the protocol whose truth we argued in what was really an extremely informal manner.

The fact that a balance enquiry quandary shows up with both forward and backward $(1,1)$ refinements, highlights the fact that it is a fundamental consequence of the refinement of an atomic action to a non-atomic protocol. Such a scenario is always going to contain moments in which the relationship between abstract and concrete states is temporarily adrift of the ideal, regardless of the variant of refinement used. The question becomes, to what extent is this state of affairs visible, and to what extent is it viewed as being problematic. Obviously if the portfolio of operations contains ones that can make visible the non-coherence of the states at such times, the expected refinement relationship will break down. At such moments the greater flexibility of retrenchment comes into its own as a means of providing a straightforward formal justification for the observations via suitable retrenchment output relations. The fact that concessions are not needed in such retrenchments is a clear signal that the refinement fold has not in fact been departed from, and we can anticipate a host of similar applications of concession-free retrenchments for treating atomic-versus-finegrained situations. This foreshadows a retrenchment *Atomicity Pattern* that provides a common framework for dealing with them. See [3].

Generalised forward refinement can also address these more general atomicity situations, and potentially more conveniently, since it can arrange for any inconvenient phenomena to remain concealed within the interior of appropriate (m,n) sequences. However the price to be paid is, that one needs the assurance that all relevant (m,n) sequences have been taken into account, something that can become a demanding obstacle if operation scheduling is under the control of the environment rather than the system. The engineering pros and cons of these various approaches deserve further consideration.

The plethora of refinements considered in this paper, for the same abstract transaction model and concrete protocol, lead one to suspect that in any such situation, the precise way in which a refinement between abstract and concrete can be set up, can be chosen more or less at will — if one is ingenious enough.[30] This wider question also merits further consideration.

Despite this paper's title, we see that retrenchment played a relatively small part in its contents. In many ways, this is as it should be since refinement very often does 'almost all' that one would like. Retrenchment possesses a weaker theory than refinement, so it is preferable to use refinement when it can properly address the job at hand. Retrenchment's value comes in plugging the gaps that refinement leaves, in relating different refinement strands, and in formulating model evolution descriptions that can act as a spur for further refinement work — in other words, in acting as the mortar between solid refinement bricks. In this paper retrenchment smoothly overcame the infelicities of the $(1,1)$ refinement treatments, and acted as an enabler for the construction of a $(1,1)$ forward refinement long believed not to exist; these constitute a fine testament to its value when appropriately used.

[30] The ingenuity primarily involves the invention of suitable retrieve relations between abstract and concrete states —in this paper we have exhibited a varied selection of retrieve relations for the various refinements discussed.

References

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

[2] Department of Trade and Industry: Information Technology Security Evaluation Criteria, 1991, Http://www.cesg.gov.uk/site/iacs/itsec/media/formal-docs/Itsec.pdf.

[3] Banach, R., Jeske, C., Hall, A., Stepney, S.: Retrenchment and the Atomicity Pattern, Submitted.

[4] Banach, R., Poppleton, M.: Retrenchment: An Engineering Variation on Refinement, *2nd International B Conference* (D. Bert, Ed.), 1393, Springer, Montpellier, France, April 1998.

[5] Banach, R., Poppleton, M.: Sharp Retrenchment, Modulated Refinement and Simulation, *Formal Aspects of Computing*, **11**, 1999, 498–540.

[6] Banach, R., Poppleton, M.: Engineering and Theoretical Underpinnings of Retrenchment, 2002, Submitted, http://www.cs.man.ac.uk/˜banach/some.pubs/Retrench.Underpin.pdf.

[7] Banach, R., Poppleton, M.: Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study, *Requirements Engineering Journal*, **8**(2), 2003, 266–288.

[8] Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Retrenching the Purse: Finite Sequence Numbers and the Tower Pattern, *Formal Methods 2005* (J. Fitzgerald, I. Hayes, T. A., Eds.), LNCS 3582, Springer, Newcastle, UK, 2005.

[9] Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Retrenching the Purse: Finite Exception Logs, and Validating the Small, *Software Engineering Workshop 30* (M. Hinchey, Ed.), IEEE, Layola College Graduate Center, Columbia, MD, 2006, To appear.

[10] Banach, R., Poppleton, M., Jeske, C., Stepney, S.: Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties, *2nd International Symposium on Leveraging Applications of Formal Methods, Verifi cation and Validation* (B. Steffen, T. Margaria, Eds.), IEEE, Paphos, Cyprus, 2006, To appear.

[11] Barthe, G., Courtieu, P., Dufay, P., de Sousa S., M.: Tool-assisted Specification and Verification of the JavaCard Platform, *AMAST 2002* (H. Kirchner, C. Ringeissen, Eds.), 2422, Springer, 2002.

[12] Behm, P., Desforges, P., J-M., M.: MÉTÉOR: An Industrial Success in Formal Development, in: Bowen et al. [26], 374–393.

[13] Beierle, C., Börger, E.: Refinement of a typed WAM extension by polymorphic order-sorted types, *Formal Aspects of Computing*, **8**(5), 1996, 539–564.

[14] Beierle, C., Börger, E.: Specification and correctness proof of a WAM extension with abstract type constraints, *Formal Aspects of Computing*, **8**(4), 1996, 428–462.

[15] Bert, D., Bowen, J., King, S., Wald´en, M., Eds.: *Proc. ZB2003: Formal Specifi cation and Development in Z and B*, vol. 2651 of *LNCS*, Springer, Turku, Finland, June 2000.

[16] Bicarregui, J., Ritchie, B.: Invariants, Frames and Postconditions: a Comparison of the VDM and B Notations, 670, Springer, 1993, ISBN 3-540-56662-7.

[17] Börger, E.: A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control, *CSL'89. 3rd Workshop on Computer Science Logic* (E. Börger, H. Kleine Büning, M. M. Richter, W. Schönfeld, Eds.), 440, Springer-Verlag, 1990.

[18] Börger, E.: A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation, in: *Mathematical Foundations of Computer Science* (B. Rovan, Ed.), vol. 452 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990, 1–14.

[19] Börger, E.: The ASM Refinement Method, *Formal Aspects of Computing*, **15**, 2003, 237–275.

[20] Börger, E., Del Castillo, G.: A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study), *Proc. 1st IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'95)* (B. Werner, Ed.), November 1995.

[21] Börger, E., Durdanovi´c, I.: Correctness of compiling Occam to Transputer code, *Computer Journal*, **39**(1), 1996, 52–92.

[22] Börger, E., Mazzanti, S.: A Practical Method for Rigorously Controllable Hardware Design, in: *ZUM'97: The Z Formal Specifi cation Notation* (J. P. Bowen, M. B. Hinchey, D. Till, Eds.), vol. 1212 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, 151–187.

[23] Börger, E., Rosenzweig, D.: The WAM – Definition and Compiler Correctness, in: *Logic Programming: Formal Methods and Practical Applications* (C. Beierle, L. Plümer, Eds.), North-Holland, 1994, 20–90.

[24] Börger, E., Salamone, R.: CLAM Specification for Provably Correct Compilation of CLP($\mathcal{R}$) Programs, in: *Specifi cation and Validation Methods* (E. Börger, Ed.), Oxford University Press, 1995, 97–130.

[25] Börger, E., Stärk, R.: *Abstract State Machines. A Method for High Level System Design and Analysis*, Springer, 2003.

[26] Bowen, J., Dunne, S., Galloway, A., King, S., Eds.: *Proc. ZB2000: Formal Specifi cation and Development in Z and B*, vol. 1878 of *LNCS*, Springer, York, UK, August 2000.

[27] Cooper, D., Stepney, S., Woodcock, J.: *Derivation of Z Refi nement Proof Rules*, Technical Report YCS-2002-347, University of York, 2002.

[28] Dawes, J.: *The VDM-SL Reference Guide*, UCL Press/ Pitman Publishing, London, 1991.

[29] Derrick, J., Boiten, E.: *Refi nement in Z and Object-Z*, FACIT, Springer, 2001.

[30] Hall, A.: Using Formal Methods to Develop an ATC Information System, *IEEE Software*, **13**, 1996, 66–76.

[31] ISO 15408, v. 3.0 rev. 2: *Common Criteria for Information Security Evaluation*, 2005.

[32] ISO/IEC 13568: *Information Technology – Z Formal Specifi cation Notation – Syntax, Type System and Semantics: International Standard*, 2002,
http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip.

[33] Jones, C.: *Systematic Software Development using VDM*, Second edition, Prentice-Hall, 1990.

[34] Kleene, S.: *Mathematical Logic*, Wiley, 1967, *also* Dover 2002.

[35] Lano, K., Haughton, H.: *Specifi cation in B: An Introduction Using the B-Toolkit*, Imperial College Press, 1996.

[36] RAISE Method Group: *The RAISE Method Manual*, Prentice Hall, 1995.

[37] Schellhorn, G.: Verification of ASM Refinements Using Generalized Forward Simulation, *JUCS*, **7**, 2001, 952–979.

[38] Schellhorn, G.: ASM Refinement and Generalisations of Forward Simulation in Data Refinement: A Comparison, *Theoretical Computer Science*, **336**, 2005, 403–435.

[39] Schellhorn, G., Grandy, H., Haneberg, D., Reif, W.: *The Mondex Challenge: Machine Checked Proofs for an Electronic Purse*, Technical Report 2006-02, Institut fur Informatik Universitat Augsburg, February 2006.

[40] Schneider, S.: *The B-Method*, Palgrave Press, 2001.

[41] Spivey, J.: *The Z Notation: A Reference Manual*, Second edition, Prentice-Hall, 1992.

[42] Stärk, R., Schmidt, J., Börger, E.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer, 2000.

[43] Stepney, S.: New Horizons in Formal Methods, *The Computer Bulletin*, 2001, 24–26.

[44] Stepney, S., Cooper, D.: Formal Methods for Industrial Products, in: Bowen et al. [26], 374–393.

[45] Stepney, S., Cooper, D., Woodcock, J.: More Powerful Z Data Refinement: Pushing the State of the Art in Industrial Refinement, *11th International Conference of Z Users* (J. Bowen, A. Fett, M. Hinchey, Eds.), 1493, Springer, Berlin, Germany, September 1998.

[46] Stepney, S., Cooper, D., Woodcock, J.: *An Electronic Purse: Specification, Refinement and Proof*, Technical Report PRG-126, Oxford University Computing Laboratory, 2000.

[47] Stepney, S., Polack, F., Toyn, I.: Patterns to Guide Practical Refactoring: examples targetting promotion in Z, in: Bert et al. [15], 20–39.

[48] Stringer-Calvert, D. W. J., Stepney, S., Wand, I.: Using PVS to Prove a Z Refinement: a case study, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations, Graz, Austria, September 1997* (C. Jones, J. Fitzgerald, Eds.), 1313, Springer, 1997.

[49] Teich, J., Kutter, P., Weper, R.: Description and Simulation of Microprocessor Instruction Sets Using ASMs, *Abstract State Machines: Theory and Applications* (Y. Gurevich, P. Kutter, M. Odersky, L. Thiele, Eds.), 1912, Springer-Verlag, 2000.

[50] Van, H., George, C., Janowski, T., Moore, R.: *Specification Case Studies in RAISE*, FACIT, Springer, 2002.

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