Under consideration for publication in Formal Aspects of Computing

Atomicity Failure and the Retrenchment Atomicity Pattern

Richard Banach[1], Czesław Jeske[1], Anthony Hall[2], Susan Stepney[3]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
{banach,cjeske}@cs.man.ac.uk,
[2]Independent Consultant,
United Kingdom
anthony@anthonyhall.org,
[3]Dept. of Computer Science, University of York,
Heslington, York, YO10 5DD, U.K.
susan.stepney@cs.york.ac.uk

Abstract. The issues surrounding the question of atomicity, both in the past and nowadays, are briefly reviewed, and a picture of an ACID (atomic, consistent, isolated, durable) transaction as a refinement problem is presented. An example of a simple air traffic control system is introduced, and the discrepancies that can arise when read-only operations examine the state at atomic and finegrained levels are handled by retrenchment. Non-ACID timing aspects of the ATC example are also handled by retrenchment, and the treatment is generalised to yield the Retrenchment *Atomicity Pattern*. The utility of the pattern is confirmed against a number of different case studies. One is the Mondex Electronic Purse, its protocol treated as a conventional atomic transaction. Another is the recovery protocol of Mondex, viewed as a compensated transaction (leading to the view that compensated transactions in general fit the pattern). A final one comprises various unruly phenomena occurring in the implementations of software transactional memory systems, which can frequently display non-ACID behaviour. In all cases the *Atomicity Pattern* is seen to perform well.

Keywords: Atomic Actions, Refinement, Retrenchment, Air Traffic Control, Mondex, Compensated Transactions, Software Transactional Memory.

1. Introduction

Atomicity is by no means a new issue in the design of computer systems: insights about mutual exclusion primitives and their consequences have developed since the earliest days of the subject [BA82, Ray88, LMWF94, Lyn96]. The deepening understanding of atomicity mechanisms led to the development of efficient distributed operating systems [SBG05, HB03, CDK05], and bolstered with maturing knowledge about data representation, has led to the flowering

Correspondence and offprint requests to: Richard Banach, School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, U.K. email: banach@cs.man.ac.uk

of a database industry offering products which are sufficiently reliable, and sufficiently easily usable, that they now occupy mission critical positions in many organisations [BHG87, GMUW03, EN03, CB04, Lon04, ZBM03]. The vast information resources available on the web provide ever increasing opportunities for applications in all spheres to benefit from a distributed approach.

The watchword of the implementation of an atomic action in these pardigms is the ACID (atomic, consistent, isolated, durable) transaction [JK97]. This provides the default goal to which implementations now aspire, providing the maximum possible conceptual clarity for the transaction concept, convenient for higher levels of applications.

At the heart of the atomicity question is some notion of refinement. One has a picture of a task, performed atomically at the abstract level, but broken up into fragments, usually co-operating via a protocol, at a more concrete level. The two are supposed to achieve the same ends, so the concrete level ought to be some sort of refinement of the abstract level. Now it is not the job of this paper to get embroiled in a discussion of the variety of possibilities for, or optimal formulation(s) of refinement, for achieving the proposed tieup between abstract and concrete levels. Many formulations exist for capturing one notion of refinement or another, and each can have its merits weighed, when confronted with the present challenge — in fact many of the possibilities that exist are adequate to the task. In this paper we will, as needed, make use of the results of one particular detailed study of the problem [BS08b, BS08a], one whose precise results prove to be especially convenient for our purposes since they integrate so well with the problems studied here.

In the ideal ACID-refinement-based formulation, the protocol always either runs to a successful conclusion, or the whole attempt gets wiped, leaving no trace. Nowadays however, although the ACID ideal is still highly prized, the necessity to relax some or all of its precepts to avoid excessive performance penalties is widely recognised. This is prompted by scenarios such as web services [Pap07, WS], long-lived workflows [GG, EL97, SGMA89], and highly concurrent and highly distributed environments [JK97].

Our aim in this paper is to illustrate the capabilities of retrenchment [RET, BPJS07, BJP08, BP00, BP03, PB03] in dealing with the various kinds of circumstance which arise that can spoil the ideal ACID based refinement view of atomicity. On the one hand, the ‘atomic action refined to distributed algorithm’ perspective generates a strong pull to find common structure across many such situations. On the other, the necessity of departure from the ideal can arise in a myriad ways, leading to a proliferation of incompatible special cases if a common account is pitched at an inappropriate level. It turns out that retrenchment can provide a vehicle for capturing a useful degree of commonality across such situations, while leaving room for incompatibilities regarding specific details.

The commonality arises via the retrenchment *Atomicity Pattern*, which we introduce and explore in this paper. This is an arrangement of refinements and retrenchments that we show is common to atomicity situations. The idea is developed in the remainder of this paper as follows.

We start by outlining the formulation of refinement and retrenchment used in this paper in Section 2. In Section 3 we then introduce our main example, a pidgin air traffic control (ATC) display application, abstracted from the CDIS development [Hal96], in which critical pieces of information must reach a family of displays in the correct order (and ultimately in a timely fashion). An ‘ideal specification’ of this is atomic; a ‘more realistic specification’ captures some of the realities of asynchrony and of constraints on timing delays. The gap between them is bridged by a series of small retrenchments. (The small size of the model evolution steps is rather reminiscent of the Event-B approach [Abr03, ACM], and of many practical ASM refinements [BS03, Bör03].) In Section 4 we indicate how one might proceed towards an implementation from such a starting point. In Section 5 we abstract from the phenomena introduced thus far, to present a general structure for capturing a whole class of similar situations involving atomicity, its refinement, and its possible breakdown. This is the *Atomicity Pattern* itself. In Section 6 we confront the *Atomicity Pattern* with a different scenario, namely the atomicity issues arising in the refinement development of the Mondex Purse [SCW00, WSC+08], and we see that the proposed structure can account adequately for the phenomena in Mondex. In Section 7 we explore compensated transactions, which are (in principle long lived) transactions accompanied by ‘compensations’. The latter are invoked when the main transaction aborts, in order to not have to undo potentially large amounts of useful work accomplished already, and also to deal with the consequences of side effects in the environment that cannot be rolled back. A reinterpretation of Mondex failed transactions (which in the original Mondex picture are viewed as another kind of successful transaction), proves to be a nice vehicle for such compensated transaction notions, and we see that the *Atomicity Pattern* fits the bill here too. In Section 8 we briefly explore transactional memory models, and see that some of the phenomena encountered there also fit the *Atomicity Pattern* bill rather well. Section 9 recapitulates and concludes.

N. B. Aside from when it is occasionally necessary to stray into notational paradigms appropriate to other specialised application areas, we express all our models using the Z notation, for its relative conciseness (which we amplify by taking occasional notational liberties).

2. Refinements and Retrenchments

In this section we briefly review the notions of refinement and retrenchment used in the remainder of the paper. We give these in Z [ISO02, Spi92, WD96, DB01] since that is the vehicle for the majority of the examples we discuss. For refinement, we adapt slightly the formulation in [CSW02] as used in the Mondex development [SCW00]. The retrenchment rules are adapted to fit conveniently with the refinement ones. It will suffice to quote the forward rules for refinement and retrenchment.

The context of the rules for both refinement and retrenchment is a pair of (abstract and concrete) ADTs: $(A, AInit, \{AOp, AI_{Op}, AO_{Op} \mid Op \in \mathsf{Ops}_A\})$, and $(C, CInit, \{COp, CI_{Op}, CO_{Op} \mid Op \in \mathsf{Ops}_C\})$. Here A is the abstract state schema, $AInit$ is its initialisation, and for $Op \in \mathsf{Ops}_A$, we have the operation schemas and their input space and output space schemas: AOp, AI_{Op}, AO_{Op}. Similarly, for the concrete side, we have C the the concrete state schema, with $CInit$ its initialisation, and for $Op \in \mathsf{Ops}_C$, we have the operation schemas and their input space and output space schemas: COp, CI_{Op}, CO_{Op}.

For refinement, the two ADTs are related by the retrieve relation $R_{A,C}$ on states, and on a per operation basis, the input and output relations $RI_{A,C,Op}$ and $RO_{A,C,Op}$, which relate the abstract and concrete input spaces and output spaces respectively. For refinement we stipulate that $\mathsf{Ops}_A \subseteq \mathsf{Ops}_C$, but such that any $Op \in \mathsf{Ops}_C - \mathsf{Ops}_A$ is a refinement of a corresponding unstated abstract operation whose definition is skip on the abstract state schema A.

Forward refinement is given by three main proof obligations (POs), *initialisation*, *applicability* and *correctness*:

$$\forall C' \bullet CInit \Rightarrow \exists A' \bullet AInit \wedge R'_{A,C}$$

$$\forall A;\ AI_{Op};\ C;\ CI_{Op} \bullet R_{A,C} \wedge RI_{A,C,Op} \wedge \mathrm{pre}\, AOp \Rightarrow \mathrm{pre}\, COp$$

$$\forall A;\ AI_{Op};\ C;\ CI_{Op};\ C';\ CO_{Op} \bullet R_{A,C} \wedge RI_{A,C,Op} \wedge \mathrm{pre}\, AOp \wedge COp \Rightarrow \exists A';\ AO_{Op} \bullet AOp \wedge R'_{A,C} \wedge RO_{A,C,Op}$$

For purposes of intuition, we can say the following. The initialisation PO ensures that the starting conditions of the abstract system correspond appropriately to those of the concrete system. The applicability PO then demands that if an abstract operation may be called (from some abstract before-state and input), then the corresponding concrete operation can also be called from a matching concrete before-state and input. Finally, the correctness PO demands that when a call of a concrete operation completes, yielding a specific after-state and output, then there is an abstract after-state and output that matches the concrete ones, and this after-state and output pair could have been yielded by a call of the abstract operation from the matching before-state and input.

For retrenchment, the two ADTs are related by the retrieve relation $R_{A,C}$ on states as before, and on a per operation basis, the within, output, and concedes relations $W_{A,C,Op}$, $O_{A,C,Op}$, and $C_{A,C,Op}$. For retrenchment $\mathsf{Ops}_A \subseteq \mathsf{Ops}_C$, and there is no restriction on operations in $\mathsf{Ops}_C - \mathsf{Ops}_A$.

Two POs define a retrenchment between two models: *initialisation* and *correctness*:

$$\forall C' \bullet CInit \Rightarrow \exists A' \bullet AInit \wedge R'_{A,C}$$

$$\forall A;\ AI_{Op};\ C;\ CI_{Op};\ C';\ CO_{Op} \bullet R_{A,C} \wedge W_{A,C,Op} \wedge COp \Rightarrow \exists A';\ AO_{Op} \bullet AOp \wedge ((R'_{A,C} \wedge O_{A,C,Op}) \vee C_{A,C,Op})$$

Note that applicability issues are subsumed via the within relation, thus we assume also that the following *well-behavedness* PO holds:

$$\forall A;\ AI_{Op};\ C;\ CI_{Op} \bullet R_{A,C} \wedge W_{A,C,Op} \Rightarrow \mathrm{pre}\, AOp \wedge \mathrm{pre}\, COp$$

On an intuitive level, we can describe these POs as demanding the following, which is to be compared with the above remarks for refinement. Initialisation is identical. For correctness, the within relation, which is a(n otherwise unrestricted) relation over all before-state variables and inputs, gives the capability of narrowing the focus of what the PO speaks about; this may be appropriate in case the abstract and concrete systems are too different to allow a full comparison, or a full comparison is inappropriate for some other reason — in any event the use of a within relation such that $R_{A,C} \wedge W_{A,C,Op}$ is *properly* stronger than $\mathrm{pre}\, AOp \wedge \mathrm{pre}\, COp$ (that the former merely implies the latter being well-behavedness itself) ought to be fully validated against the external requirements. The output relation is over all variables of the abstract and concrete transitions. The permitted presence of *all* of the variables (i.e. outputs and after-states, and including before-states and inputs, if required) allows properties stronger than the mere conjunction of the retrieve relation on after-states with a relation on outputs to be expressed — such a possibility plays a vital role in the RE-Refs of the *Atomicity Pattern* below. Finally, the concession allows a description to be made (while still remaining within the remit of the formal framework) of the state of affairs that results when a pair of corresponding

transitions *is unable to* reestablish the retrieve relation on after-states — again, the allowed presence of all of the variables maximises the expressive power.

As well as the POs, which are useful for establishing the retrenchment, we will be interested in the associated simulation relation for the operation *Op*, which holds when the hypothesis and conclusion of the correctness PO hold, avoiding the 'don't care' discharge of the PO.

$$\Sigma_{A,C,Op} \equiv R_{A,C} \land W_{A,C,Op} \land COp \land AOp \land ((R'_{A,C} \land O_{A,C,Op}) \lor C_{A,C,Op})$$

The truth of $\Sigma_{A,C,Op}$ implies genuine *AOp* and *COp* transitions that make the retrenchment data true in a suitable way, unlike the PO itself. Obviously there is an analogous notion for refinement, but we will not need it.

3. The Abstract Pidgin ATC System

In an air traffic control system, there is (among other things) a family of workstations *WS*, at which the air traffic controllers sit and do their work. The workstations display a variety of items of information to the controllers, some of them more critical than others. For the most critical items, correct ordering and timeliness are important issues: the controllers must be made aware of changes in the critical items as they occur, within a tightly controlled *LATENCY*, so that safety in the aerodrome is not compromised.

For simplicity, we assume that there is just one critical item, the *QNH* value (modelled as a natural number say), and that this is the only item on the workstation display. For an air traffic control system this is, admittedly, a rather drastic simplification.

The system ideal is atomic update of all the displays, so we can build an abstract A model consisting of a single *QNH* value, updated by an *ANewQnh* operation, and observed by an *AShowWs* operation which outputs the *QNH* value displayed on each workstation:[1]

Aworld

$Aqnh : QNH$

ANewQnh

$\Delta Aworld$
$Aqnh? : QNH$

$Aqnh' = Aqnh?$

AShowWs

$Aworld$
$Adisp! : WS \to QNH$

$Adisp! = WS \times \{Aqnh\}$

The atomicity of the ideal model is reflected in the fact that *AShowWs* always outputs a constant function. However, the reality is that the updates to the system *QNH* value are broadcast to the individual workstations over a network. This generates transmission delays, and the various tolerances in the system cause these to be observable, within limits. These aspects require a more detailed model than the ideal A model, and we approach the construction of the appropriate 'realistic specification' in a number of steps.

First we build the AH model, which just includes history information:

AHworld

$AHhist : \text{seq}_1\ QNH$

AHNewQnh

$\Delta AHworld$
$AHqnh? : QNH$

$AHhist' = AHhist \frown \langle AHqnh? \rangle$

AHShowWs

$AHworld$
$AHdisp! : WS \to QNH$

$AHdisp! = WS \times \{last\ AHhist\}$

With suitable initialisations, it is not hard to see that the A and AH models are interrefinable under (equality output relations and) the retrieve relation:

[1] N.B. Each model-specific schema and variable is prefixed by a letter or two indicating the relevant model: A for abstract, AH for A with history, AA for A with asynchrony, AT for AA with time, C for concrete, and D is the alphabetic successor of C.

$R_{A,AH}$

$$Aworld$$
$$AHworld$$
$$Aqnh = last\ AHhist$$

Next, we build the AA model. This introduces some asynchrony into the system by allowing the *AAShowWs* operation (which corresponds to the *AHShowWs* operation in the previous model) to output a selection of values from the history. The only restriction on these is that displaying *QNH* values out of order is forbidden. This constraint is satisfied by keeping a record of the current *QNH* value for each workstation, and allowing it to advance using the new operation *AAWsUpdate*:[2]

AAworld

$$AAhist : \mathrm{seq}_1\ QNH$$
$$AAseq : WS \rightarrow \mathbb{N}$$
$$\mathrm{ran}\ AAseq \subseteq \mathrm{dom}\, AAhist$$

AAWsUpdate

$$\Delta AAworld$$
$$AAws? : WS$$
$$(AAseq'\ AAws?) \geq (AAseq\ AAws?)$$
RestSame.....

AANewQnh

$$\Delta AAWorld$$
$$AAqnh? : QNH$$
$$AAhist' = AAhist \frown \langle AAqnh? \rangle$$
RestSame.....

AAShowWs

$$AAworld$$
$$AAdisp! : WS \rightarrow QNH$$
$$AAdisp! = AAseq \fatsemi AAhist$$

Although we managed to relate the A model and the AH model using refinement (as formulated in Section 2), this time, our conventional refinement notion is too demanding to describe the relationship between the AH model and the AA model, since the outputs of the two *ShowWs* operations do not match up according to any plausible output relation that we could imagine drawing up.[3] We need the greater flexibility of retrenchment to capture what is going on. The retrenchment needed is given by the retrieve relation $R_{AH,AA}$ and the output relation for *ShowWs*, with all other retrenchment data trivial:[4]

$O_{AH,AA,ShowWs}$

$$AHworld'$$
$$AAworld'$$
$$AHdisp! : WS \rightarrow QNH$$
$$AAdisp! : WS \rightarrow QNH$$
$$AAdisp! = AAseq' \fatsemi AAhist'$$
$$AHdisp! = WS \times \{last\ AAhist'\}$$

$R_{AH,AA}$

$$AHworld$$
$$AAworld$$
$$AHhist = AAhist$$

A retrenchment output relation generalises a refinement one, in the sense that it can refer to the state variables in relating the observed outputs, whereas a refinement output relation cannot do so. This extra flexibility is needed here: the AH output is the constant value resulting from the latest atomic update, whereas the AA output is a selection of potentially older values from the system history, since there may be some workstations which are not completely up to date.

The retrenchment just introduced is of a special kind, which we call a retrenchment-enhanced refinement (RE-Ref) since it falls short of being a refinement by the smallest of margins, namely that its output relation needs to be more

[2] Henceforth, in Z schemas, the phrase RestSame..... means that any other variables in scope but not explicitly assigned to in the schema are to remain unchanged, something typically handled less tersely in legal Z by including a suitable Ξ schema for the unaffected variables inside the main schema to specify the lack of change of the unaffected variables.

[3] Beyond this, the AA operation *AAWsUpdate*, does not correspond to any AH model operation. However, we can allow this in a refinement if the new operation refines an (unstated) AH model operation whose body is skip. Now since *AAWsUpdate* only manipulates the *AA* variable *AAseq*, which is invisible (via the retrieve relation $R_{AH,AA}$) to the *AH* model, the *AAWsUpdate* operation will indeed refine skip. See Section 2 for more details.

[4] I.e. given by identities on inputs and outputs, false for concessions.

complicated than just a simple relation between the output spaces. Although it is more complicated, the additional complexity is itself highly constrained. We can define an RE-Ref more precisely thus:

Definition 3.1. A retrenchment-enhanced refinement (RE-Ref) is a retrenchment with data as follows. The retrieve relation is a relation between the state spaces only (as usual). For all operations *Op* the following hold. The within relation is a relation between input spaces only (i.e. it is given by a predicate that does not mention the before-states). The concession is trivial (i.e. it is given by the predicate false). The output relation is a relation between output spaces and after-states only, constrained in the following manner. There is a (partial) function $REf_{A,C,Op}$ from after-states to outputs, such that $REf_{A,C,Op}$ is implied by $\Sigma_{A,C,Op}$, and $REf_{A,C,Op}$ implies $O_{A,C,Op}$.

$$REf_{A,C,Op} : A';\ C' \pfun AO_{Op};\ CO_{Op} \qquad \Sigma_{A,C,Op} \Rightarrow REf_{A,C,Op}$$

$$O_{A,C,Op} : AO_{Op};\ CO_{Op};\ A';\ C' \qquad REf_{A,C,Op} \Rightarrow O_{A,C,Op}$$

In a nutshell, whatever the output relation says, must be derivable from a functional relationship from after-states to outputs.

We see that if we completely removed the outputs from the systems, we would have a perfectly good refinement, and the observed relationship between the outputs that we actually have in the two systems would be derivable from it. Putting it another way, if it is only such asynchrony that a change of model is introducing, then the true information contained in the state histories of the models is not being lost, and so any disagreement in the outputs of read-only operations on the states should be explicable from the state histories themselves; i.e. the change of model indeed ought to be capable of being described by an RE-Ref as we defined it above.

Of course, similar observations apply when it is the inputs at stake rather than outputs. In that case the asynchrony considerations mean that corresponding inputs arrive at different times in the two models. This can be handled in the within relation of the later occurring of the two operations, as follows. The functional relationship $REf_{A,C,Op} : A';\ C' \pfun AO_{Op};\ CO_{Op}$ of Definition 3.1 is removed (the output relation relation reverting to a relation on outputs alone, as befits a refinement), and the within relation acquires a corresponding functional relationship from before-states to inputs (which we will also refer to as $REf_{A,C,Op}$ for simplicity): $REf_{A,C,Op} : A;\ C \pfun AI_{Op};\ CI_{Op}$, with $\Sigma_{A,C,Op}$ implying $REf_{A,C,Op}$ as before, and $W_{A,C,Op}$ implying $REf_{A,C,Op}$. A case in point of such behaviour is to be found in Section 6.[5]

The utility of RE-Refs in handling issues of asynchrony is the first contribution that retrenchment makes to the atomicity arena in situations where atomicity in the very strictest sense does not quite hold up.

The next step in the development of the ATC system is to introduce the time aspect, so that we can bring the asynchrony under control. This leads to the AT model, containing an *ATtimenow* variable with values in *TIME* (which we model using the naturals in this paper, but which could be any totally ordered set), and where *ATtimenow* is updated by an *ATTick* operation. Also each update of the *QNH* value is timestamped, with the values being recorded in the *AThisttime* variable (which is thus also totally ordered):

$$TIME == \mathbb{N}$$

ATworld

$$\begin{array}{l} AThist : \mathrm{seq}_1\ QNH \\ ATseq : WS \rightarrow \mathbb{N} \\ ATtimenow : TIME \\ AThisttime : \mathrm{seq}_1\ TIME \\ \hline \mathrm{dom}\ AThist = \mathrm{dom}\ AThisttime \\ \mathrm{ran}\ ATseq \subseteq \mathrm{dom}\ AThist \\ \forall\, i, j : \mathrm{dom}\ AThisttime \bullet i \leq j \mid \\ \quad AThisttime(i) \leq AThisttime(j) \\ last\ AThisttime \leq ATtimenow \end{array}$$

ATTick

$$\begin{array}{l} \Delta ATworld \\ \hline ATtimenow' = ATtimenow + 1 \\ \mathsf{RestSame.....} \end{array}$$

ATWsUpdate

$$\begin{array}{l} \Delta ATworld \\ ATws? : WS \\ \hline (ATseq'\ ATws?) \geq (ATseq\ ATws?) \\ \mathsf{RestSame.....} \end{array}$$

[5] Although it may seem strange to have inputs (which are normally freely assigned) functionally dependent on state (which is normally invisible), we recall that the abstract system, the one synchronising late, is having its behaviour designed to simulate the behaviour of the concrete system, for which the corresponding 'freely assigned' input has occured earlier, and has been appropriately remembered in the state.

ATNewQnh

$\Delta ATworld$
$ATqnh? : QNH$

$AThist' = AThist \frown \langle ATqnh? \rangle$
$AThisttime' = AThisttime \frown \langle ATtimenow \rangle$
RestSame.....

ATShowWs

$ATworld$
$ATdisp! : WS \rightarrow QNH$

$ATdisp! = ATseq \fatsemi AThist$

Thus far we have a straightforward superposition refinement [BKS83, FF90, Kat93] of the AA model, since we have just added some new data and operations, and no new observations of the new data. All the old data and operations remained unchanged. A retrieve relation that simply forgets the time in abstracting from the AT model easily proves the refinement.

However this is not enough. We need to distinguish well behaved workstations from badly behaved ones. The former get their updates done within *LATENCY* timesteps, the others don't. A well behaved workstation satisfies:

ATWellBhWs

$ATworld$
$ws? : WS$

$\forall sq : \mathrm{dom}\, AThist \setminus (1 \,..\, ATseq\ ws?) \bullet ATtimenow - (AThisttime\ sq) \leq LATENCY$

i.e. all its unprocessed updates were introduced at most *LATENCY* ago. We want the refinement part of the eventual relationship between the AA and AT models to insist that all workstations are well behaved:[6]

$R_{AA,AT}$

$AAworld$
$ATworld$

$AAworld$ **" = "** $ATworld$
$\forall ws? : WS \bullet ATWellBhWs$

To deal with the (small but nonzero) possibility that network delays turn out to be greater than desirable, leading to the failure of the retrieve relation, we need more of the expressivity of retrenchment. It is actually the innocuous *ATtick* operation we need to focus on, since it is the passage of time which causes workstations to become badly behaved. At this point we stub our toe on a small retrenchment pebble.

Since there is no *Tick* operation in the AA model, normal retrenchment policy dictates that there will be no retrenchment data (i.e. within, output or concedes relations) associated with *Ttick*. The normal policy is justified by observing that genuinely new operations introduced during a model evolution step, will concern aspects absent from the prior model, and thus any attempt to relate them to the prior model are likely to appear artificial. However, the passage of time may reasonably be taken as a universal (if usually unstated) feature of models, so that viewing the present case as a retrenchment of an unstated skip is entirely justified. This understood, the retrenchment's within and output relations can be trivial, the concession being where the interest lies:

$C_{AA,AT,Tick}$

$\Delta AAworld$
$\Delta ATworld$

$\forall ws? : WS \bullet \exists sq : \mathrm{dom}\, AThist \setminus (1 \,..\, ATseq\ ws?) \bullet ATtimenow - (AThisttime\ sq) = LATENCY$
$\quad \Rightarrow \neg\ ATWellBhWs'$

This shows that any workstation with a *LATENCY*-old update outstanding, will become badly behaved at the next tick unless it is updated beforehand.

We now have a route from the utterly atomic A model, to model AT, which abstracts the inevitable asynchrony of an implementation, but which allows the quality of that asynchrony to be quantified via a retrenchment. The A model

[6] We use **" = "** between schemas to abbreviate a set of equalities between corresponding variables that differ only in the model-identifying prefix.

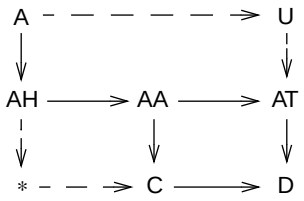

Fig. 1. Models, refinements (vertical arrows), retrenchments (horizontal arrows), making up a commuting diagram of the ATC specification development.

specifies an unattainable perfection, while the AT model represents a more complex but more realistic specification. Refinement alone can never reconcile these two widely separated viewpoints, but retrenchment can.

The retrenchment from A to AT itself is the composition of the A-AH refinement with the AH-AA and AA-AT retrenchments [BJP08]. We omit the details of the calculation, save to say that the situation is sufficiently straightforward, that the result is obtained by simply translating the variables occurring in the non-trivial bits of the earlier retrenchments to those of the A and AT models, in the obvious way.

The retrenchment utilises the *LATENCY* parameter and permits a stochastic analysis of the circumstances under which the relevant concession becomes valid. Such an analysis would consider a sample space of system runs, constructed by taking into account the many external factors that influence the behaviour of the system, but that lie outside of the formal model. The likelihood and severity of timing failures could then be derived. The output of such an analysis can provide a useful negotiating pivot between customer and supplier — the customer would be interested in a precise statement of what constituted timing failure and how often it occurred, but the details of what happened subsequently would be more a matter for the supplier, taking into account the higher level invariants demanded of the system. This scenario illustrates in minature the second contribution that retrenchment makes to the atomicity issue, namely the straightforward incorporation into a formal account, of matters that make implementations of atomic actions insufficiently ACIDic.

What we have so far is the solid arrows of the upper layer of Fig. 1, which is a commuting diagram of vertical refinements and horizontal retrenchments. These connect a family of models involved in our ATC development, abstracted from CDIS [Hal96]. Customarily, the refinement component of the A-AT retrenchment (i.e. A-AH) would enable the A-AT retrenchment to be lifted to generate a more abstract model U using results in [BJ09, Jes05]. However the fact that A and AH are interrefinable, means that nothing useful would be gained by doing this.

4. Towards an Implementation of the Pidgin ATC System

Considering the move towards an implementation of the ATC System, we refine our preceding models. We start with model AA, since that is the first along the A-AT path which incorporates asynchrony, which is unavoidable in any implementation. So as not to detract from the main focus of interest of the paper, our remarks will be merely indicative rather than comprehensive. We sketch model C, a refinement of AA, and then model D, a refinement of AT, connecting these two developments afterwards with a suitable retrenchment. Of course if we were doing implementation for real, we would not do both. The discussion of the various possibilities is intended to highlight the greater flexibility that the combination of refinement and retrenchment allows (compared with using refinement alone), in treating the requirements in an order dictated by appropriateness for development rather than theoretical constraint.

Model C's 'more realistic' description of the system contains a family of workstations, each containing its own portion of the system state; hence the map *Cwsqnh*.[7] The network is modelled as an 'ether' of messages containing *QNH* updates, to which individual workstations help themselves. To disambiguate and preserve order, we have a sequence number type *SQNO* (modelled as a positive natural number say), and the ether thus becomes a map *Cethqnh* from sequence numbers to *QNH* values. Each workstation keeps track of where it is up to with a local copy of the latest sequence number it has processed (via the map *Cwsseq*).

[7] This is best captured formally via Z promotion, though for brevity we will not use that here.

Cworld

$Cmaxseq : SQNO$
$Cwsseq : WS \rightarrow SQNO$
$Cwsqnh : WS \rightarrow QNH$
$Cethqnh : SQNO \nrightarrow QNH$

$\mathrm{dom}\, Cethqnh = 1 \,..\, Cmaxseq$
$Cwsqnh = Cwsseq \fatsemi Cethqnh$

CWsUpdate

$\Delta Cworld$
$Cws? : WS$

$(Cwsseq'\ Cws?) \geq (Cwsseq\ Cws?)$
RestSame.....

CNewQnh

$\Delta CWorld$
$Cqnh? : QNH$

$Cmaxseq' = Cmaxseq + 1$
$Cethqnh'\ Cmaxseq' = Cqnh?$
RestSame.....

CShowWs

$Cworld$
$Cdisp! : WS \rightarrow QNH$

$Cdisp! = Cwsseq \fatsemi Cethqnh$

Noting that $SQNO == \mathbb{N}_1$, the reader will quickly realise that the C model that we have just built is (mathematically) little more than a slightly verbose restatement of the AA model, with an additional dependent variable, *Cwsqnh*. Recognising this, we conclude that the C model will be interrefinable with AA. Obviously we could contemplate more dramatic refinements of the AA model, but what we have done will suffice for purposes of illustration.

Similarly, we can build a model D, refining AT. It follows the pattern established by the C model. Thus we use the same sequence number type, and use it to index both *QNH* values and their timestamps. Otherwise, the structure is as in the AT model.

Dworld

$Dmaxseq : SQNO$
$Dwsseq : WS \rightarrow SQNO$
$Dwsqnh : WS \rightarrow QNH$
$Dethqnh : SQNO \nrightarrow QNH$
$Dtimenow : TIME$
$Dethtime : SQNO \nrightarrow TIME$

$\mathrm{dom}\, Dethtime = \mathrm{dom}\, Dethqnh = 1 \,..\, Dmaxseq$
$Dwsqnh = Dwsseq \fatsemi Dethqnh$

DTick

$\Delta Dworld$

$Dtimenow' = Dtimenow + 1$
RestSame.....

DWsUpdate

$\Delta Dworld$
$Dws? : WS$

$(Dwsseq'\ Dws?) \geq (Dwsseq\ Dws?)$
RestSame.....

DNewQnh

$\Delta Dworld$
$Dqnh? : QNH$

$Dmaxseq' = Dmaxseq + 1$
$Dethqnh'\ Dmaxseq' = Dqnh?$
$Dethtime'\ Dmaxseq' = Dtimenow$
RestSame.....

DShowWs

$Dworld$
$Ddisp! : WS \rightarrow QNH$

$Ddisp! = Dwsseq \fatsemi Dethqnh$

The two refinements AA-C and AT-D will be related not only by the AA-AT retrenchment, but by a retrenchment C-D. This latter retrenchment will be the obvious counterpart of the AA-AT retrenchment at the lower level of abstraction of C and D. In detail, the retrenchment will depend on the C-D analogue of *ATWellBhWs*, which asserts that all the unprocessed updates of a workstation were introduced less than *LATENCY* ago:

DWellBhWs

$Dworld$
$ws? : WS$

$\forall sq : (Dwsseq\ ws?) + 1 \,..\, Dmaxseq \bullet Dtimenow - (Dethtime\ sq) \leq LATENCY$

As before, the refinement part of the C-D retrenchment insists that all workstations are well behaved:

$R_{C,D}$

$$Cworld$$
$$Dworld$$
$$\hline$$
$$Cworld \text{ “ = ” } Dworld$$
$$\forall\, ws? : WS \bullet DWellBhWs$$

The concession again polices the *DTick* event, singling out those workstations whose updates lag more than *LATENCY* behind the arrival of unprocessed *QNH* values:

$C_{C,D,Tick}$

$$\Delta Cworld$$
$$\Delta Dworld$$
$$\hline$$
$$\forall\, ws? : WS \bullet \exists\, sq : (Dwsseq\ ws?) + 1\ ..\ Dmaxseq \bullet Dtimenow - (Dethtime\ sq) = LATENCY$$
$$\Rightarrow \neg\, DWellBhWs'$$

We can complete the details of the retrenchment with trivial within and output relations. Using the techniques elaborated in [BJP08], we can then calculate the composition of the AA-AT retrenchment with the AT-D refinement, and compare it with the result of calculating the composition of the AA-C refinement with the C-D retrenchment. In the simple situation that we have here, these turn out to be the same. Thus we have the commuting square of retrenchments and refinements shown in the lower right half of Fig. 1.

In fact, commuting squares such as this can be built not only by hand, as we indicated above, but also using generic constructions such as are described in [BJ09, Jes05] — specifically we would need the Postjoin Theorem from one or other of these references. Typically, the relevant theorem constructs a system that ‘completes the square’ in a generic way, up to a notion of universality that invariably includes inter-refinability; in other words we can replace the generically constructed system by one inter-refinable with it without losing any of the properties of the construction, a useful property that helps keep the ‘square completing’ system looking close to applications level concerns. Obviously, one could develop the C and D models even further towards implementation by making the modelling increasingly realistic.

The above takes care of the lower layer of Fig. 1, aside from the model labelled ‘*’. Model ‘*’ refines AH and is retrenchable to C. It can be obtained via the lowering construction in [BJ09, Jes05] from AH, AA, C and their relationships, or independently, again yielding a commuting square. The fact that AA and C are interrefinable, means that ‘*’ contains nothing new beyond AH, and the fact that its workstation updates must be atomic, means that it is unrealistic.[8]

5. The Retrenchment Atomicity Pattern

The last few remarks indicate that a protocol implementation at the most abstract level possible[9] has to be refinable from the C model, not from the A model. And yet the A model captures the most transparent expression of what one would like the protocol to do, so it would be regrettable if we had to exclude it from a rigorous development. The way to reconcile these views, is to pursue the suggestion that an RE-Ref can indeed be usefully viewed as a kind of refinement, rather than as a retrenchment, which, strictly speaking, it is. Taking this view straightens out the composition along the path A-AH-AA-C into an RE-Ref, collapsing the left hand part of Fig. 1 as it does so. The resulting RE-Ref from A to C now expresses, as an almost-refinement, a useful change in modelling perspective, depicted in the vertical direction. Moreover, incorporating the ACIDity losing aspects of the retrenchment from model C to model D via the composition A-C-D, and then performing the lifting construction from [BJ09, Jes05], results in a model U entirely equivalent to the one constructed before, since the overall composition A-C-D yields the same composed retrenchment from model A to model D as obtained previously. This is a useful observation since it is often initially easier to express the ACIDity losing aspects of some development within a more concrete model than in a

[8] Unrealistic because of the intepretation of the model as a distributed system, rather than any mathematical difficulty.
[9] I.e. incorporating the fewest constraints while retaining implementability.

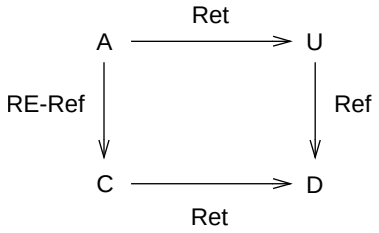

Fig. 2. The Atomicity Pattern.

more abstract one, since the ACIDity losing aspects are often centred on lower level details (relatively speaking), and the way that these are reflected at more abstract levels is not always entirely obvious. A collection of models A, C, D, U, and their interconnection via two retrenchments, a refinement and an RE-Ref, as above, constitutes the retrenchment *Atomicity Pattern*; see Fig. 2. We claim that this arrangement recurs frequently in situations featuring loss of atomicity, and thus deserves to be highlighted — a claim we support in the rest of the paper.

Referring to Fig. 1, we see that Fig. 1 is an instance of the *Tower Pattern* [BPJS]. This makes the *Atomicity Pattern* a special case of the *Tower*. However, a number of features make the *Atomicity Pattern* deserve to be singled out specially.

First and foremost, is the use of RE-Refs (specifically avoiding more general kinds of retrenchment) in the left hand side of the diagram — this collapses the zig-zag that would result if the retrenchment aspects were singled out as such there. The pure loss of atomicity implicit in the fact that we have an atomic action at one level of abstraction which is refined to a multi-step protocol at another, means that the abstract and concrete states will be adrift of the 'not in the middle of the protocol' ideal form when the protocol is actually running. This situation has been thoroughly studied in [BS08a, BS08b], and the precise relationships between possible abstract and concrete states during a protocol run are now well understood.[10] The fact that the concrete level is still a refinement (in the strict sense) of the abstract level in the absence of I/O, means that the states differ, but that they differ only in a very controlled way. This close, yet non-ideal relationship between the states, means that any observed operation inputs and outputs which are related to those state values will also be in a close, yet non-ideal relationship. For this reason, the very restricted RE-Refs are sufficient for this kind of situation.

Second, is the fact that the fairly large gap between the A and C models encourages us to take a broad perspective on how an atomic and a non-atomic model ought to be synchronised. Following the line developed in [BS08a, BS08b], the synchronisation mechanism is in fact captured in the retrieve relation between the two models. Considering our example, the finegrained path A-AH-AA-C strongly suggests an early synchronisation; i.e. in each protocol run, *ANewQnh* is synchronised (via the retrieve relation) with *CNewQnh*,[11] with the rest of the concrete protocol following behind. This gives a refinement from the A model to the C model with retrieve relation $R_{A,C_{early}}$ below. However, this is but one possibility.

In general, the single step of an atomic protocol can be mapped to practically any step of a concrete protocol which implements the atomic one, provided the various (in general nondeterministic) outcomes of the two descriptions match up via the retrieve relation (see [BS08a, BS08b] for details). Different choices merely lead to different retrieve relations between the two models.[12] As an example, consider a late synchronisation option in our CDIS example. This matches *ANewQnh* with the last *CWsUpdate* in a protocol run, identified via $CWsUpdate_{last}$ below, and implicitly requires that *CNewQnh* and all earlier occurrences of *CWsUpdate* become refinements of abstract skips. Such a synchronisation is given by retrieve relation $R_{A,C_{late}}$. (N.B. In our example, both early and late formulations of the refinement are *forward*

[10] In [BS08a, BS08b], there was no concept of loss of atomicity, so suitably designed notions of refinement could cope, without any need for retrenchment. In the present context, the different kinds of synchronisation that refinement permits us to have, inform the kinds of retrenchment one might need to handle loss of atomicity.

[11] Many conventional refinement notions demand that abstract operations are refined by operations *with the same name*. However this is just a technical convenience, and is easily generalised to the case where for each concrete step of interest, one can identify a step of *some* abstract operation of which it is a refinement; official Z refinement is like this. Our discussion presupposes this generalisation where necessary.

[12] Observe an interesting phenomenon. When a refinement preserves the atomicity, i.e. abstract and concrete steps match up 1–1 in related runs, it is usually the case that the retrieve relation is 'obvious' — there is essentially only one choice that makes sense. The situation changes dramatically when atomicity is *not* preserved. Then, the variety of possible synchronisations leads to a variety of accompanying retrieve relations. Moreover, rarely is any of them 'obvious', even though, according to the results of [BS08a, BS08b], they can all be mechanistically calculated from the details of the chosen synchronisation.

simulations, since the broadcast protocol is deterministic; i.e. all the workstations always get successfully updated (assuming weak fairness). In general, early synchronisation requires *backward* simulation to handle nondeterminism after the synchronisation point. See [BS08a, BS08b] again for the technical details.)

$R_{A,C_{early}}$

$$Aworld$$
$$Cworld$$
$$Aqnh = Cethqnh\ Cmaxseq$$

$R_{A,C_{late}}$

$$Aworld$$
$$Cworld$$
$$Aqnh = Cethqnh(min\,\mathrm{ran}\,Cwsseq)$$

$CWsUpdate_{last}$

$$CWsUpdate$$
$$\forall ws : WS \bullet ws \neq ws? \Rightarrow (Cwsseq\ ws) > (Cwsseq\ ws?)$$

Third, is a fact prompted by the preceding parenthetic remark. The detailed complexities of simulations in which the concrete state is matched to the abstract state after each concrete step of the protocol, can be largely avoided if we take a more coarse grained approach to refinement, à la ASM refinement [BS03, Bör03, Sch01, Sch05]. Here, the refinement becomes insensitive to state values in the middle of a concrete protocol run, and the retrieve relation is only required to match up abstract and concrete states at the beginning and end. This *en bloc* approach can yield considerable simplifications in the description of a single run of the protocol, but makes the description of interleaved concurrent protocol runs by independent agents rather more problematic.

In our example, *ANewQnh* together with a suitable collection of *AShowW*ss would be refined *en bloc* to an entire concrete protocol run with suitable *CShowW*ss interspersed. Done properly, this would make the previously observed discrepancies between abstract and concrete outputs disappear, since the coarser grain would enable us to schedule the abstract and concrete *ShowW*ss so that they matched up, the details of the scheduling being concealed in the interior of the coarse grained refinement. We do not give the details here, due to the technical complexity of dealing with the many interleavings of independent updates. This approach gives further encouragement to the view that an RE-Ref is after all a sensible species of refinement.

6. The Mondex Purse, as Atomic Action

Having developed the *Atomicity Pattern*, in this section we confront it with a different but nevertheless realistically grounded example, the Mondex Purse, to verify the genericity of the description of atomicity situations that it furnishes.

The Mondex Purse is a smartcard electronic purse for containing genuine money, and as such, is a security critical application. The 1990s development of Mondex was the among the first of such developments to achieve the highest possible ITSEC rating of E6 (see [WSC+08]), equivalent these days to a Common Criteria rating of EAL7 [Dep91]. The ITSEC E6 rating requires there to be an abstract model, a concrete model, and a proof of correspondence between them. For Mondex, the proof was a manual refinement proof between two Z models, an abstract model and a concrete model. The details of the Mondex project as a whole are commercially sensitive. However, in a rare departure from the usual practice regarding commercially sensitive developments, a desensitised public version of the less sensitive and intellectually more interesting parts of the development was produced in [SCW00]. The development in [SCW00] remains an impressive achievement, and a trailblazer for showing that fully formal techniques could be applied within realistic time and cost limitations on industrial scale applications.

More recently, the Mondex refinement proof was adopted as the first case study in the Verification Grand Challenge; see [JOW06, Woo06, WB07]. In this, the objective was to redo the previously hand-done proofs using state of the art verification tools. The first such attempt to be successfully carried through to completion was by the Augsburg group [SGHR06, SGH+07]. Reports of successful treatments by other groups soon followed, and the results of a variety of approaches to the task are reported in [JW08]. These not only attest to the viability of doing such developments in a fully mechanised manner, but also confirm the solidity of the original manual proof.

Despite the above, the exigiencies of refinement caused a number of issues to be treated in a less than ideal manner in Mondex. In the actual (commercially sensitive version of the) development, these were dealt with via informal arguments, which fully justified the positions taken on the issues in question. Nevertheless, a suitably formal treatment would obviously have been better, not least because a proper formalisation opens the way to mechanically checking the arguments made, with a consequent improvement in dependability. This being the case, as well as its amply

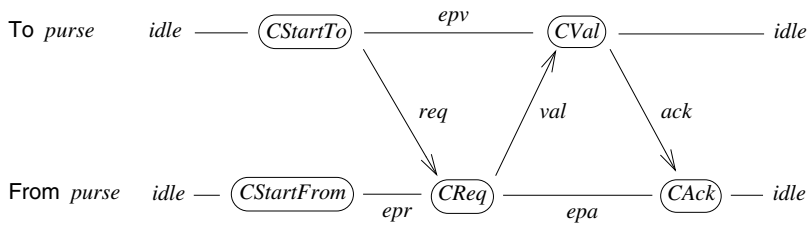

Fig. 3. The Mondex Concrete Protocol.

borne out prospects for mechanical verification, Mondex provides a superb platform for testing out the efficacy of the retrenchment approach to handling situations which turn out to be awkward for refinement — a number of existing case studies bear this aspect out equally amply [BPJS, BPJS06a, BPJS06b, BJPS07].

Turning to the Mondex development itself, at the top level, there is an abstract A model, which is a model of atomic funds transfer between purses. According to this, a transaction can do one of the following: (a) complete successfully (lodging the funds transfered instantaneously in the destination purse), *AbTransferOkay*; (b) atomically 'lose' the funds (placing them in a special 'lost' component of the state, implying that *it is known* (by the system state) that the funds are lost, permiting offline recovery later), *AbTransferLost*; (c) nothing at all happens (at this level of abstraction), *AbIgnore*. One absolute requirement that the A model embodies, is the prohibition of *failing but non-recoverable* transfers. Considering that real money is involved, this is as we would wish.

Here are the essentials of Mondex in a cut down world of only two purses, the To purse and the From purse, which are hardwired into the state. Two purses are in fact sufficient for our purposes for the following reason. In a realistic Mondex world, there are many purses, but the community of purses can be partitioned as follows. There are (non-intersecting) pairs of purses involved in transactions (and that is where the atomicity issues of interest to us lie); and there are the remaining purses, which are either idle or are involved —with another purse— in the setting up of a new non-intersecting pair — and no purse of this kind exhibits any atomicity issue of concern to us. Therefore, looking at a single pair is adequate.

Abworld

$$Afrombal : \mathbb{N}$$
$$Afromlost : \mathbb{N}$$
$$Atobal : \mathbb{N}$$
$$Atolost : \mathbb{N}$$

AbIgnore

$$\Xi Abworld$$

AbTransferOkay

$$\Delta Abworld$$
$$Avalue? : \mathbb{N}$$

$$0 < Avalue? \leq Afrombal$$
$$Afrombal' = Afrombal - Avalue?$$
$$Atobal' = Atobal + Avalue?$$
$$Afromlost' = Afromlost$$
$$Atolost' = Atolost$$

AbTransferLost

$$\Delta Abworld$$
$$Avalue? : \mathbb{N}$$

$$0 < Avalue? \leq Afrombal$$
$$Afrombal' = Afrombal - Avalue?$$
$$Atobal' = Atobal$$
$$Afromlost' = Afromlost + Avalue?$$
$$Atolost' = Atolost$$

In reality of course, some procedure involving a lot of low level activity takes place. Two purse owners wishing to participate in a funds transfer insert their purses, the From purse and the To purse, into an interface device, and type in the instructions. The device then initiates the funds transfer process by informing the two purses of the details of the required transaction. A protocol, described by the C model (which refines the A model), is then enacted. Fig. 3 shows how it works.

The protocol starts its run by calling the *CStartFrom* and *CStartTo* events in the respective purses, assuming that both purses are in an *idle* state. These two events prime both purses with the information needed to execute the protocol, including in particular, for each purse, the information directly pertaining to its counterpart. This puts both purses in a position to check the extent to which (to the best of the available local knowledge) the playout of a running protocol instance conforms to what is expected at that point. This reconciling of actual against expected is what gives the Mondex protocol its recoverability properties in the face of protocol failure and interruption.

Once the *CStart* events take place, the protocol proper commences. As part of *CStartTo*, the To purse issues a (cryptographically protected) *req* (request) message to the From purse and enters the *epv* (expecting payment value) state. On receipt of the *req* message, the From purse, which has been in the *epr* (expecting payment request) state since its *CStartFrom*, executes the *CReq* (Request) event, decrements its balance appropriately, and sends the amount requested in a (cryptographically protected) *val* (value) message to the To purse, itself going into the *epa* (expecting payment acknowledgement) state. On arrival of the *val* message, the To purse, executes the *CVal* (Value) event, increments its balance appropriately, becomes *idle* again, and sends a (cryptographically protected) *ack* (acknowledgement) message back to the From purse. When this finally arrives, the From purse, executes the *CAck* (Acknowledgement) event, and the protocol completes with both purses becoming *idle* once more. In addition to this just described ideal protocol run outline, are numerous cases corresponding to failed protocol runs. In reality, any of the messages we mentioned may get lost in transit, and any of the events that produce and/or consume them may fail to take place. Tieing off the loose ends in the protocol generated thereby, is the *CAbort* event, which unconditionally cleans up any partially completed protocol run, logging any information needed for recovery, and resets the relevant purse to the *idle* state. (*CAbort* is called in each purse at the start of every protocol run, as a precaution in case the purse is still waiting for some previous transfer to complete.) The recoverability properties of Mondex are in fact attributable to the rather subtle properties of the *CAbort* event.

Above we described an unproblematic run of the protocol. Of course much can go wrong in practice. The protocol may get interrupted by accident or by design, and a purse may be subjected to deliberate attack in order to attempt to subvert its integrity (and in the ideal case, to increase the balance it contains beyond what is legitimate). The protocol must be robust against all this. Part of the protection built into the protocol is the fact that any time a purse feels like it, it has the option of doing nothing or of aborting the current transaction: this means that a purse will always respond to any request to perform any of its actions, but the response will be null or aborting if the purse does not consider the request to be appropriate in the context of its current state. This creates a large number of additional playouts of the protocol which are not illustrated in Fig. 3. The non-trivial ones end in *CAbort*s by the participating purses. We do not define the *CAbort* event in this paper, primarily because it does not impact on the atomicity issues of interest to us in this section, but also because it is the most complex element of the Mondex protocol, and taking the time and space to describe it properly would entail a considerable detour from our intended aims. Nevertheless it can be shown that all the possible playouts do indeed do the right thing, because the concrete protocol can be proved to be a refinement of the abstract A model, both for successful runs (which refine *AbTransferOkay*) and for aborting runs (which refine *AbTransferLost*). See [SCW00] for the original account, and also [BJPS07], which discusses the properties of the protocol in detail in a manner compatible with the present discussion.

How does the preceding fit the *Atomicity Pattern*? Well, we have an A model and a C model, and C refines A. Since A is atomic and C is not, any operation that reads and outputs the state values will exhibit a discrepancy if abstract and concrete versions are invoked at an inopportune moment. A balance enquiry operation is just such an operation, and one we would not unreasonably expect to find among the operations that are offered to the user in a banking application.[13]

A balance enquiry operation can cause various kinds of trouble, depending on the synchronisation embodied in the refinement used, and how this interacts with other technical details in the relationship between abstract and concrete models. In [SCW00] the abstract transfer is synchronised with the concrete *CReq* operation, so the discrepancy shows up in a To purse enquiry if invoked while the value is in transit, i.e. between the departure and subsequent arrival of the *val* message in Fig. 3. In [BJPS07], a different refinement is given which synchronises the abstract transfer with the concrete *CVal* operation, which puts the discrepancy on the From purse side. (The pros and cons of dealing with balance enquiries in various different ways are studied in depth in [BJPS07].) Either way, there is scope for an AA model, introducing asynchrony at the most abstract level possible.

In the original development, [SCW00], in between the A and C models there is a B model. Its purpose is to capture axiomatically various properties of the state of the C model that are useful in discharging the (backwards) refinement proof between the A and B models — these properties being subsequently proved to be inductive invariants of all C model runs during the (forwards) refinement proof between the B and C models. In any event, the B model turns out to be technically very close to the C model, so it could not play the role that we have in mind for the AA model here. However it is very easy to construct a suitable AA model from scratch. For simplicity, we encode a transaction in progress via $AAval > 0$ in the AA model below.

[13] The fact that accounting for outputs which are incompatible for atomicity reasons cannot be convincingly done using refinement alone, led, along with a whole host of other technical issues, to the complete omission of balance enquiry operations from [SCW00].

AAbworld

$AAfrombal : \mathbb{N}$
$AAfromlost : \mathbb{N}$
$AAtobal : \mathbb{N}$
$AAtolost : \mathbb{N}$
$AAval : \mathbb{N}$

AAbTransferStart

$\Delta AAbworld$
$AAvalue? : \mathbb{N}$

$AAval = 0$
$0 < AAvalue? \leq AAfrombal$
$AAval' = AAvalue?$
$AAfrombal' = AAfrombal - AAvalue?$
RestSame.....

AAbTransferOkay

$\Delta AAbworld$

$AAval > 0$
$AAval' = 0$
$AAtobal' = AAtobal + AAval$
RestSame.....

AAbTransferLost

$\Delta AAbworld$

$AAval > 0$
$AAval' = 0$
$AAfromlost' = AAfromlost + AAval$
RestSame.....

Note that the above model just separates out the beginning and end of a transaction, by introducing the *AAbTransferStart* event; the beginning and end were combined in the A model. The relationship between the A and AA models is evidently an RE-Ref, but, assuming we choose to synchronise the A model transaction late (i.e. the A model operation is synchronised with *AAbTransferOkay* or *AAbTransferLost*), the RE-Ref needs to relate discrepancies in inputs to the states via within relations, since late synchronisation implies that the AA model input occurs earlier than the A model input. In this scenario, we can use the simple retrieve relation $R_{A,AA}$ (where RestEqual..... has the obvious meaning), and within relations $W_{A,AA,TransferOkay}$ and $W_{A,AA,TransferLost}$ (whose bodies are identical). Note that doing it this way makes essential use of the ability of retrenchment within relations to combine not only input information but also state information into a single relationship, just as the corresponding ability of retrenchment output relations to combine not only output information but also state information into a single relationship was exploited earlier in the relation $O_{AH,AA,ShowWs}$.

$W_{A,AA,TransferOkay/Lost}$

Abworld
AAbworld
Avalue?

$Avalue? = AAval$

$R_{A,AA}$

Abworld
AAbworld

$AAval = Afrombal - AAfrombal$
RestEqual.....

(N. B. If we synchronised early rather than late, though the inputs would coincide, we would need a more complex, nondeterministic, retrieve relation to intercede in what would need to be a backward simulation refinement as in the original development [SCW00]. See [BS08b, BS08a] for a general treatment.)

Further down the modelling hierarchy, one can relatively straightforwardly synchronise *AAbTransferStart* with the *CReq* operation, *AAbTransferOkay* can be synchronised with *CVal*, and with a little further manipulation of the concrete state, *AAbTransferLost* can be synchronised with a suitable *CAbort*. (N. B. This general approach, of dealing with (non-)atomicity issues right away, and then pursuing a relatively conventional refinement approach towards the lowest level models, is broadly similar to the strategy followed in the RAISE approach to the mechanisation of the Mondex proofs. See [HGS06], and [GH08] in [JW08] — except that RAISE cannot deal directly with the change of operation signatures implicit in a splitting of an atomic action into a protocol, and so the splitting step must be performed informally at the outset of the development.)

One can delve further into the refinement possibilities permitted by the approach in [BS08b, BS08a]. For example, with a more complex retrieve relation, one could synchronise *AAbTransferStart* with the first concrete *CStart*, *AAbTransferOkay* with *CAck*, and *AAbTransferLost* with a suitable *CAbort*. In the end, there are many choices, discussed at length in [BS08a]. In a nutshell, the Mondex development is, in the authors' view, a superb example of the *Atomicity Pattern* at work. Moreover, it is a pre-existing example, not one invented specially to put the pattern in a good light, and for that it is the more convincing.

7. The Mondex Purse, as Compensated Transaction

In the previous section we viewed Mondex as an ACID transaction problem, and discussed various refinements and RE-Refs in that light — we note that Mondex is squarely in the financial world, where the merest whiff of 'alkalinity' in financial transactions is utterly intolerable. In fact, the Mondex protocol maintains a sufficiently copious (electronic) papertrail, that aborted transactions, even though they do not achieve their original objective, can be traced, and the whereabouts of the funds they involve can ultimately be reconciled with the original intentions of the participants. In this manner, in the financial world, protocol failure is recategorised as a different kind of success, and ACIDity emerges as a matter of careful definition.

Of course, there is nothing to stop us using the non-ACID potential of the *Atomicity Pattern* to quantify some aspects of interest of the protocol, such as the proportion of transactions that might abort under some given set of assumptions or other, but this is a case of using the possibilities of retrenchment (perfectly reasonably), as a technical convenience, rather than a pronouncement about a lack of integrity of the protocol. Further possibilities could entail examining the performance of the protocol under the assumption that one or more of the security hypotheses it rests on is weakened by some specified amount, etc.

However, there exists another approach to the question of possible alternative matches of the *Atomicity Pattern* to the Mondex protocol, via a redefinition of success and failure, and we look at this now. It permits the exercise of the horizontal aspects of Fig. 2, which describe possible lack of ACIDity of the asynchronous protocol.

When a Mondex protocol run fails (by being a refinement of *AbTransferLost*, as described above), the following facts hold, and the sequence of events to be described takes place.

Firstly, a protocol run refines *AbTransferLost* if and only if both purses involved in the transaction have the (*auth*enticated) transaction *p*ayment *d*etails '*pdauth*', logged in their local (i.e. on-purse) exception log *CXexlog* (where $X \in \{from, to\}$), these details having been put there by suitable *CAbort* events, which executed on each of the From and To purses.[14] If this is the case, then both purses need to get in contact with the bank underwriting the Mondex system in order that both log contents can be uploaded to the bank's central Mondex exception archive, whereupon the presence of a matching pair of records from the From and To purses of the transaction confirms the loss of the *val* message in transit. Since it is only when the *val* message is in flight that the From and To purse balances do not add up to the original amount, it follows that only when the the *val* message is lost in transit, is there any nontrivial action to be taken to recover lost funds. In this situation the bank can restore the missing funds to the From or To purse according to the purse owners' wishes.[15]

According to our original ACID picture, the preceding can be viewed as another kind of transaction present within the overall Mondex system, and guaranteed to 'succeed' in the same way that the core protocol is guaranteed to succeed, i.e. by definition. Alternatively, we can choose to view the *AbTransferLost* outcomes of the core protocol as *failures* of the transaction, followed in time by *compensating actions* engaged in at the bank, which serve to remedy the preceding protocol failures.

In such a view, the overall process is still guaranteed to be successful, as one would demand of a financial application, but its ingredients can consist of an initial failure (implying temporary loss of ACID properties) followed by an ACIDity-restoring compensation.

Compensation mechanisms for transactions have been studied with interest in recent years [BFH$^+$02, BFN05, BHF04, BBF$^+$05] as a structured way of avoiding having to abandon the whole of a long-lived transaction (with all the attendant impact on performance) when some particular part of it fails — especially if that part comes just before the end, when a *bona fide* abort would entail the loss of a huge amount of useful work, and especially if also, side effects have been performed in the environment which cannot be undone (such as the sending of irrevocable communications that may already have been received by their addressees). A compensation undertakes such measures as are needed to recover 'ACIDity in the real world sense' in these situations (insofar as such recovery is indeed possible).

Conventionally, transactions [BHG87, GR93, BN97, WV02] connect with their environment (specifically the transaction manager residing in the operating system) via stylised interfaces. For the most simply structured transactions these amount to *TransStart*, *TransCommit*, *TransAbort*. The transaction starts with *TransStart* which informs the transaction manager that the transaction has started and its activities are to be policed via the transaction mechanism. After

[14] As hinted above, all of the subtlety of the Mondex protocol resides in the somewhat complicated properties of the *CAbort* events. We do not need to plunge into these in detail to make our point in this paper; see [BJPS07] and other cited references for a full discussion. However, we point up the fact that *both* purses need to contain matching entries in their logs for there to be a protocol failure — it turns out that a single entry in one of the two purses does not constitute a protocol failure by itself.

[15] We eschew discussion of situations in which one or other of the parties to the transaction neglects, or is uncooperative regarding, the stated interaction with the bank. Obviously, in reality, pragmatic measures must be in place to deal with such eventualities.

that, the transaction acquires resources and does its work (ideally, entirely in private, so that it can easily be rolled back without visible trace if necessary). The work either succeeds or fails. If it succeeds, the transaction manager is asked to run the corresponding *TransCommit* operation, which commits the changes effected by the transaction, making them visible to the wider environment. If it fails, the transaction manager is asked to run the *TransAbort* operation, which undoes the work provisionally attempted by the transaction, and restores all the resources accessed by the transaction to their previous state. The use of the *TransStart*, *TransCommit*, *TransAbort* interfaces (and their more complex analogues in more sophisticated transaction models) is what enables the transaction manager to ensure that the four ACID attributes are maintained across the community of executing transactions, and is also what enables individual transactions to be written in isolation from one another.

Compensated transactions elaborate the fixed range of rollback mechanisms implicit in a transaction-manager-controlled *TransAbort* operation, and place the responsiblity for rolling back more complex transactions in the lap of the transaction writer. This is especially useful when rollback, in the strict system state sense, is impossible for reasons such as were already noted above. The previously cited references [BFH$^+$02, BFN05, BHF04, BBF$^+$05] illustrate well the wide range of issues that one has to take design decisions about when the structuring and scheduling of compensations interacts with a whole host of familiar semantic issues in an integrated linguistic framework for compensated transactions.

For the purpose of illustrating the fit between compensated transactions (CTs) and the *Atomicity Pattern*, we introduce a tiny CT language. Although some of its combinators resemble those of Z schema calculus, and despite the fact we actually use Z schemas to define the transactions at the lowest level, the semantics of complex expressions is closer to the operational nature of process algebras, as we sketch below, and as is the case for the majority of compensated transaction languages. The productions of our tiny CT language are:

$$CT := \epsilon \mid (CT_1 \; ; \; CT_2) \mid (CT_1 \lor CT_2) \mid (CT_1 \;\; || \;\; CT_2) \mid (CT_1 \div CT_2) \mid \checkmark \mid \times$$

In the preceding, ϵ is the null (compensated) transaction, and $(CT_1 \; ; \; CT_2)$, $(CT_1 \lor CT_2)$, $(CT_1 \;\; || \;\; CT_2)$ are respectively the sequential composition, nondeterministic choice, and parallel composition of compensated transactions. $(CT_1 \div CT_2)$ is the basic compensated transaction pair, with CT_1 being the transaction's primary task and CT_2 being its compensation. We can abbreviate $(CT_1 \div \epsilon)$ to just CT_1. When $(CT_1 \div CT_2)$ is executed, the primary task CT_1 is actually performed, and the compensation CT_2 (in effect a kind of continuation) is pushed onto the compensation stack. The normal execution of transactions is manipulated by encountering ✓ and ✗ in the control flow. The symbol ✓ acts as a kind of commit, forcing the emptying of the compensation stack, while ✗ interrupts the execution, and passes control to the compensations stacked on the compensation stack, which are executed in last-in first-out order. Obviously, a proper semantics would clarify the many questions left unanswered by this outline description, but what we have said will do for this paper.

Let us now return to the Mondex atomic level. We introduce the *Ideal Transaction Abstract* model (prefix *ITAb*), defined as:

$$ITAbTansfer = (ITAbIgnore \;\lor ITAbTransferOkay)$$

where *ITAbIgnore* **"="** *AbIgnore* and *ITAbTransferOkay* **"="** *AbTransferOkay*. This is a model which does not include any notion of transaction failure in the sense discussed earlier, i.e. there is no vestige of *AbTransferLost*, and it is derived from the Mondex abstract model of the preceding section by simply omitting the *AbTransferLost* possibility.

To cope with the transaction failure implicit in *AbTransferLost*, we build another model, the *Compensated Transaction Abstract* model (prefix *CTAb*), defined as:

$$CTAbTansfer = (CTAbIgnore \lor CTAbTransferOkay \lor ((CTAbTransferLost \div CTAbRestoreLost) \; ; \times)) \; ; \checkmark$$

where *CTAbIgnore* **"="** *AbIgnore*, *CTAbTransferOkay* **"="** *AbTransferOkay*, *CTAbTransferLost* **"="** *AbTransferLost*, and *CTAbRestoreLost* is given by the schema:

CTAbRestoreLost

$$CTAbRestoreLost2To \lor CTAbRestoreLost2From$$

where:

CTAbRestoreLost2To

$\Delta CTAbworld$
$CTAval? : \mathbb{N}$

$CTAtobal' = CTAtobal + CTAval?$
RestSame.....

CTAbRestoreLost2From

$\Delta CTAbworld$
$CTAval? : \mathbb{N}$

$CTAfrombal' = CTAfrombal + CTAval?$
RestSame.....

Clearly, the *CTAbTansfer* model allows for transaction failure, and includes a suitable compensation for it in the operation *CTAbRestoreLost*. If we now define $CTAbTansfer_T$ to be $(CTAbIgnore \vee CTAbTransferOkay \vee CTAbTransferLost)$, i.e. it is the portion of *CTAbTansfer* that will actually be done as a single atomic action, then we will have a retrenchment from *ITAbTansfer* to $CTAbTansfer_T$. Most of this is predictably trivial, the only part of interest being the concession:

$C_{ITAbTansfer, CTAbTansfer_T}$

$\Delta ITAbworld$
$\Delta CTAbworld$
$CTAval? : \mathbb{N}$

$ITAfrombal' + ITAtobal' = ITAfrombal + ITAtobal = CTAfrombal + CTAtobal$
$= CTAfrombal' + CTAtobal' + CTAval?$

This now expresses the possibility that the $CTAbTansfer_T$ transaction may lose funds whereas the *ITAbTansfer* transaction cannot. Putting *ITAbTansfer* in the place of model A in Fig. 2 and putting $CTAbTansfer_T$ in the place of model U, we see that the atomic actions of our compensated transaction model fit the *Atomicity Pattern* very well. Evidently, the models in this top row of the pattern could now be refined to lower level non-atomic Mondex models in the way that we have already seen.

Taking a more coarse grained view, we could look at the relationship between *ITAbTansfer* and the complete *CTAbTansfer* model, allowing the latter to do two steps (*CTAbTransferLost* and its compensation *CTAbRestoreLost*) before we insisted on reconciling its behaviour with that of *ITAbTansfer*. In such a case we would find nothing amiss, since *CTAbRestoreLost* is able to compensate completely for the inappropriate consequences of *CTAbTransferLost*, and thus *CTAbTansfer* would be a refinement of *ITAbTansfer*. In such a case the top row of the *Atomicity Pattern* in effect collapses to a single model, the A model of Fig. 2, since the retrenchment from model A to model U reduces to a refinement, and this can be composed with the refinement from model U to model D, sidestepping model U completely if desired.

In more messy scenarios, such as we have hinted at above, the relevant compensations would not necessarily be able to completely reverse the effects of their ACIDity-losing primary tasks. In such situations, the top row of the *Atomicity Pattern* would be genuinely required, in that it would not be possible to elide model U regardless of the granularity of the viewpoint taken. Our next case study features precisely this kind of behaviour.

8. Transactional Memory

Compared with the previous section, in this section we go the opposite extreme as regards the size of the basic atomic actions that we deal with, since now, these often reside at the level of individual machine instructions. These however, are pefectly capable of achieving the messiness alluded to in the previous paragraph.

The contemporary CPU chip scene is dominated by the twin facts that while minaturisation continues apace for the time being, allowing many processors to be placed on a single chip, the increase in speed of an individual processor, so notable from generation to generation in the past, has more or less slowed to a standstill. Tomorrow's processors are likely to be no faster than today's, though we will have many more of them at our disposal. This has spawned a desire to make greater and more convenient use of the potential power of multi-core CPUs, all in the face of many years' experience of the difficulty in making concurrency a true 'mass' programming paradigm.

One potentially promising approach to achieving greater usability for concurrent computation is via transactional memory (TM) [LR06]. The idea is that rather than providing programmers with explicit low level devices (such as locks etc.) for controlling concurrency, with the well known consequence that most programs making non-trivial use of these will be replete with bugs of rather exquisite obscurity (and will thus, in effect, be almost useless),[16]

[16] Typical sources on concurrent programming, eg. [BA82, Ray88, LMWF94, Lyn96] stress the fact that the transient aspects of concurrent program

programmers would instead be provided with an encapsulation mechanism, a transaction-like atomic action syntactic primitive (whose optimum semantics were to be determined by future research), and the implementation would have the responsibility of ensuring its correct operation.

TM comes in broadly two flavours, hardware and software. Predictably, the hardware flavour supports the atomicity primitive via the hardware memory management system [HM93, RG02, HWC+04, YBM+07, RRP+07, KHR+08], while the software flavour looks to do the same thing via a software layer sitting just above the memory management system [HF03, HMPJH05, SMAT+07, IB07, ABHI08, MBS+08, DS09]. Recently a hybrid scheme has been considered, exploiting off-the-shelf memory management hardware in a novel way to implement TM [AHM09]. In this section, we will, without striving to be in any way comprehensive, consider three typical examples (paraphrased from [ABHI08]) that arise in this fertile and intensively studied area, and how they relate to the *Atomicity Pattern*.

Given the amount of research into transaction notions that has been done in recent decades (amply demonstrated in the references cited in the previous paragraph), TM would not be such an intriguing issue were it not for the fact that 'properly transacted' code is expected to run alongside, and to co-operate sensibly with, 'non-transacted' legacy code. Combining the potential pitfalls of this with the kinds of tricks perpetrated by optimising compilers, such as code movement and speculative computation, gives rise to copious bug-spawning phenomena that are easy enough to imagine, and that constitute exactly the kind of scenarios that the *Atomicity Pattern* was designed to capture. In contrast to the applications of the pattern in preceding sections, which could all plausibly be carried out in a top-down manner, in TM, the action is all at the instruction set level, and the challenge is to not only to keep the low level behaviour under reasonable control, but to find abstractions that are useful to describe it at a higher level.

Values out of thin air. We examine our first example, zombie transactions that produce 'values out of thin air'. Below, we see a model A, specified in Z, (and intended to occupy the A position in Fig. 2). It consists of three natural-valued variables, such that initially, two of them are equal and away from zero, the other being zero. Three operations are given, *AAtomic*1, *AAtomic*2 and *AUnprotected*1. The first two have transaction semantics, in that either a non-trivial effect, or no visible change takes place — and furthermore, this is all accomplished atomically (on the understanding that Z operations specify instantaneous changes of state). In addition, we make use of outputs to model the writing of information to machine registers. The last, *AUnprotected*1, is not given a roll-back option — it represents legacy non-transaction code, and consists of the value assignment to a single variable, which may be seen as corresponding to a single machine instruction running alongside the others. The fact that it corresponds to just one instruction, means that there is no conflict between the implicit atomicity of Z and the normal execution of machine instructions.

Aworld

$Au, Av, Ax : \mathbb{N}$

*AAtomic*1

Δ*Aworld*

$((Au \neq Av \wedge Ax' = 42 \vee Au = Av \wedge Ax' = Ax)$
RestSame.....
$) \vee \Xi$*Aworld*

*AAtomic*2

Δ*Aworld*

$(Au' = Au + 1$
$Av' = Av + 1$
RestSame.....
$) \vee \Xi$*Aworld*

*AUnprotected*1

Δ*Aworld*
$Ar1! : \mathbb{N}$

$Ar1! = Ax$
RestSame.....

AInitially

Aworld

$Au = Av \neq 0$
$Ax = 0$

Following [ABHI08], we can ask whether, when all three operations are executed, $Ar1!$ can ever acquire the value 42. Clearly, with the semantics we have described, the answer is no. Provided *AAtomic*1 and *AAtomic*2 are atomic, the values of *Au* and *Av*, as seen by *AAtomic*1, can never differ, so the assignment of *Ax* to 42 can never take place, and *AUnprotected*1 always sees 0 as the value of *Ax*, regardless of when it runs. Now, provided that we have a reliable implementation of *bona fide* transactions to use for refining *AAtomic*1 and *AAtomic*2, the A model can be refined towards an implementation, represented by eg. model C in the *Atomicity Pattern*.

Next, we see a description of the same scenario in terms more appropriate to a Software Transactional Memory (STM) system such as Bartok-STM [HPST06]. As regards the *Atomicity Pattern*, it is intended for the U position of Fig. 2.

execution, namely the details of the low level instruction schedule, being irreproducible under normal circumstances, make the identification of errors in faulty concurrent code extraordinarily hard compared with the situation for sequential code.

	Initially: `u == v != 0 , r1 == 0 , r2 == 0 , x == 0`		
	Thread 1	Thread 2	Thread 3
1	`// Atomic1`	`// Atomic2`	`// Unprotected1`
2	`atomic{`	`atomic{`	`// non-atomic`
3	`  r1 = u;`	`  u++;`	
4	`  r2 = v;`	`  v++;`	`r1 = x;`
5	`  if (r1 != r2)`	`}÷{`	
6	`  {x = 42;}`	`  u = 'u;`	
7	`}÷{`	`  v = 'v;`	
8	`  x = 'x;`	`}`	
9	`}`		

Three threads execute instruction level versions of the operations above. The essentials of the semantics are as follows. Individual instructions (such as `r1 = u`) execute atomically. Sequential composition '`;`', is porous in the sense that it allows the interleaving of the activities of other threads. The `atomic` construct is less reliable than a fully ACID transaction, in that work is done in-place, allowing it to be seen by other threads, and the usual R/W conflicts between different `atomic` blocks are detected lazily (i.e. perhaps after another thread has seen some effect of the block), and are rolled back asynchronously. We indicate the latter by using our compensation mechanism from the previous section, since that too is not intended to execute atomically with the primary task. The compensations feature pre-primed variables to remember variables' values at the beginning of the transaction for roll-back purposes.

If we now ask, in this new situation, whether `r1` can ever acquire the value 42, the answer becomes yes. A possible scenario is as follows (cf. [ABHI08]). Thread 2 runs and its `atomic` block starts. In between the `u++` and the `v++` of Thread 2, Thread 1 starts and runs to completion. Since `u` and `v` are different at this point, `x` acquires the value 42. Of course, Thread 1 is in conflict with Thread2, and since it started later, it is the thread whose atomic action must be aborted. However, before Thread 1 actions the compensation for its `atomic` block, Thread 3 runs, picking up the 42 from `x` and depositing it in `r1`. Once the conflicting transaction in Thread 1 has been rolled back, no trace of it remains, and `r1` has acquired the value 42 out of thin air. The possibility of 'values out of thin air', created by zombie transactions that *de facto* access more data than would be permitted in any serial execution, is one of the classic ills that STM systems offering less than ACID semantics for their atomic blocks are heir to.

It is clear now why the instruction level model has to go in the U position of the *Atomicity Pattern* rather than in a C-like position. While ideally, an implementation remains faithful to the abstraction it is supposed to incarnate, this one obviously doesn't. And rather than a refinement relationship from abstraction to code, as we would prefer, we must be content with a retrenchment. Of course, the model we presented is still an abstraction, but it is an abstraction of a different kind of behaviour than in the A model. Refining the U model, for instance to make the details of the STM implementation more explicit, would take us in the direction of the D model of the *Atomicity Pattern*.

We turn to the retrenchment from A to U. Disregarding, for simplicity, the fact that Z naturals are unbounded whereas machine level ones are bounded (something that can be routinely incorporated into the retrenchment without upsetting our story, so is omitted for simplicity), we will just concentrate on the functional discrepancy between the A and U models, which amounts to focusing on the errant behaviour of `r1` under `Unprotected1`. The only item in the retrenchment data that is relevant to this is the concession $C_{AUnprotected1,\texttt{Unprotected1}}$, a relatively minimal choice for which could be as follows, which clearly captures what transpires in our simple scenario:

$C_{AUnprotected1,\texttt{Unprotected1}}$

$\Delta Aworld$
$\Delta \texttt{Uworld}$
$Ar1! : \mathbb{N}$
$\texttt{r1}! : \mathbb{N}$

$\texttt{r1}! - Ar1! = 42$
RestSame.....

A wider question that arises, is of course the extent to which one can reasonably expect to capture more far-reaching consequences of such lapses in atomicity as we have seen here, but purely on the basis of knowing that the implementation of atomic actions in an STM system is imperfectly ACIDic.

Unfortunately, it is not hard to see that in general, one could say almost nothing at all beyond the trivial true, since as is well known, a single unexpected value could have the most far-reaching consequences in a discrete transition

system such as those we are dealing with, since the occurrence of such a single unexpected value could be coupled to any behaviour whatsoever. Without reasonably incisive knowledge of what threads the system supported, the inter-dependencies between them, and the higher level invariants that the system was expected to uphold, we could predict almost nothing.

The privatisation problem. We examine our second example, the 'privatisation problem' in which a piece of data is accessed, sometimes from within atomic blocks, sometimes directly. Below we see another A model, this time working on a natural and a boolean. The disposition of the Z schemas is intended to indicate that *AUnprotected*1 must always be executed after *AAtomic*1. Besides this, there are no constraints at this level of abstraction, so that *AAtomic*2 is free to be interleaved arbitrarily with respect to the other two operations. Again we ask whether *Ax* can ever be left with the value 42. As previously, the answer is no. If *AAtomic*2 runs first, then *Ax* becomes 42, but it is then guaranteed that *Ax* will be incremented by *AUnprotected*1. On the other hand, if *AAtomic*2 does not run first (and *AAtomic*1 only ever executes the Ξ*Aworld* option if it conflicts with an earlier transaction, i.e. not ever in this case since *there is no earlier transaction* to potentially conflict with), then *AAtomic*2 will see *Ax_shared* as false and so will not attempt to assign *Ax* to 42, and there will be no possible race between that assignment and the increment in *AUnprotected*1.

Aworld

$Ax\_shared : \mathbb{B}$
$Ax : \mathbb{N}$

*AAtomic*1

Δ*Aworld*

$(\ Ax\_shared' = \text{false}$
RestSame.....
$) \lor \Xi$*Aworld*

*AAtomic*2

Δ*Aworld*

$((Ax\_shared \land Ax' = 42$
$\lor \neg\, Ax\_shared \land Ax' = Ax)$
RestSame.....
$) \lor \Xi$*Aworld*

AInitially

Aworld

$Ax\_shared = \text{true}$
$Ax = 0$

*AUnprotected*1

Δ*Aworld*

$Ax' = Ax + 1$
RestSame.....

Following [ABHI08] once more, we now look at a code level description of the situation, in the context of Bartok-STM. Below, we see two threads, the first of which executes `Atomic1` followed by `Unprotected1`, and the other of which executes `Atomic2`.

Initially: `x_shared == true , x == 0`

	Thread 1	Thread 2
1	`// Atomic1`	`// Atomic2`
2	`atomic{`	`atomic{`
3	`  x_shared =`	`  if (x_shared)`
4	`  false;`	`  {x = 42;}`
5	`}÷{`	`}÷{`
6	`  x_shared =`	`  x = 'x;`
7	`  'x_shared;`	`}`
8	`} ;`	
9	`// Unprotected1`	
10	`// non-atomic`	
11	`x++;`	

As before, we can ask whether in this lower level world `x` can ever acquire the value 42. Again the answer is now yes, and the scenario that shows this runs as follows. `Atomic2` starts and runs. In between `Atomic2`'s read from `x_shared` and its write to `x`, `Atomic1` runs in its entirety. Of course, it will be detected that `Atomic1` is in conflict with the earlier `Atomic2`, so it will have to be rolled back, after which `Unprotected1` can run. `Unprotected1` now has the opportunity to complete its increment on `x` before `Atomic2` gets round to its write of 42 to `x`. (N. B.

As [ABHI08] point out, the preceding is not the only possible scenario. If writes are buffered, then the write of 42 could get delayed, causing a similar outcome, even without conflicting transactions.) All of this constitutes our U model.

As before, both the A and the U model can be refined, each using a faithful implementation of their own notion of atomicity, but the two models can only be related to each other using a weaker notion than refinement, such as retrenchment. In the current context, although we know that the incompatible behaviour becomes visible in the *Ax*/`x` variables, it is harder to know 'who to blame' for this, since both *AAtomic*2/`Atomic2` and *AUnprotected*1/`Unprotected1` access them. For the sake of having a simple retrenchment, we will 'blame' *AUnprotected*1/`Unprotected1` in this paper, though a better treatment would utilise a more coarse-grained approach to retrenchment [Ban09] to allow all the different possibilities to be represented more fairly.[17] With this proviso, we can again restrict to just giving a simple concession for the relevant operation.

$C_{AUnprotected1,\texttt{Unprotected1}}$

$\Delta Aworld$
$\Delta \texttt{Uworld}$

$Ax \neq \texttt{x} = 42$
RestSame.....

The publication problem. We move on to our third example, again taken from [ABHI08], the 'publication problem' in which a piece of data is initially private to a thread, and then becomes shared. From a serialisability perspective, this is the most innocuous looking of all our examples. A piece of data is manipulated privately. When the private manipulation is finished, the owning thread atomically sets a public flag to signal the public availability of the data, after which, other transactions access it in a disciplined way.

Again, the ideal A model is given in Z below. Once more, the disposition of the *AUnprotected*1 and *AAtomic*1 schemas indicates that the latter is to be executed after the former. The other atomic block, *AAtomic*2, runs interleaved with these two. We see that from the vantage point of atomic semantics, nothing can go wrong. Either the atomic block *AAtomic*2 is executed after *AAtomic*1, whereupon it sees the updated *Ax_shared* variable, and knows that it is permitted to access *Ax* and thence to update *Ar*1! accordingly. Or *AAtomic*2 is executed before *AAtomic*1, whereupon it does not see the new value of *Ax_shared*, and must relinquish its desire to read *Ax*. Either way, *Ar*1! ends up as -1 or as 42.

Aworld

$Ax\_shared : \mathbb{B}$
$Ax : \mathbb{N}$

*AUnprotected*1

$\Delta Aworld$

$Ax' = 42$
RestSame.....

*AAtomic*1

$\Delta Aworld$

$(\ Ax\_shared' = \text{true}$
RestSame.....
$) \lor \Xi Aworld$

*AAtomic*2

$\Delta Aworld$
$Ar1! : \mathbb{Z}$

$((Ax\_shared \land Ar1! = Ax$
$\lor \neg\, Ax\_shared \land Ar1! = -1)$
RestSame.....
$) \lor \Xi Aworld$

AInitially

$Aworld$

$Ax\_shared = \text{false}$
$Ax = 0$

Next, we see the code level description, in the context of Bartok-STM, constituting our U model. We again see two threads, the first of which executes `Unprotected1` followed by `Atomic1`, and the other of which executes `Atomic2`. Note that the second of these, `Atomic2`, has an empty compensation in view of the fact that it only writes to a register, which is regarded as an output variable, and thus there are no memory writes to roll back.

[17] This also illustrates well a universal truth about retrenchment, namely that what one wishes to *achieve* in a retrenchment has a crucial effect on what one *puts into* the retrenchment.

Initially: `x_shared == false , x == 0`

	Thread 1	Thread 2
1	`// Unprotected1`	`// Atomic2`
2	`// non-atomic`	`atomic{`
3	`x = 42;`	`  r1 = -1;`
4	`// Atomic1`	`  if (x_shared)`
5	`atomic{`	`    {r1 = x;}`
6	`  x_shared =`	`}÷{ }`
7	`    true;`	
8	`}÷{`	
9	`  x_shared =`	
10	`    false;`	
11	`}`	

Even given the problems we have seen earlier, there appears to be no problem with this example. The only possible source of conflict between `Atomic1` and `Atomic2` is the shared variable `x_shared`, and since each atomic block only accesses it during one instruction, either execution order (for these instructions) defines an acceptable serialisation. As [ABHI08] point out though, the problem goes deeper. There is no inkling, from the perspective of Thread 2, that the order of reading `x` and `x_shared` is significant. An optimising compiler may therefore schedule the read of `x` early, before `Unprotected1` has done its update to `x`. As a result, the value 0 may be the one used by `Atomic2`, if the subsequent execution schedules `Atomic2` after `Atomic1`.

Again, the only way of reconciling the ideal behaviour of the A model with the errant behaviour of the U model is via a retrenchment. In this case, it is clear who we have to 'blame'. It is the *AAtomic*2/`Atomic2` pair. Here is the straightforward concession which results.

$C_{AAtomic2,\texttt{Atomic2}}$

$\Delta Aworld$
$\Delta \texttt{Uworld}$
$Ar1! : \mathbb{N}$
$\texttt{r1}! : \mathbb{N}$

$Ar1! \in \{-1, 42\}$
$\texttt{r1} = 0$
RestSame.....

9. Conclusions

In the preceding sections, we took a particularly simple example, based on an ATC application [Hal96], and via a series of simple models and small model evolution steps (reminiscent of the Event-B approach [Abr03, ACM], and of many practical ASM refinements [BS03, Bör03]), teased out how issues arising from non-atomicity of the real system interacted with the remainder of the development. The step from an atomic to a non-atomic model meant that inevitably, if one examined the states of the abstract and concrete models at an inopportune moment, some discrepancy would be observed, which went beyond what traditional substitutivity-based notions of refinement could cope with. Quite where the discrepancy might be observed, depended on how one chose to synchronise the atomic abstract action with one of the constituent non-atomic concrete actions which implemented it. (The very non-atomicity of the concrete model guarantees that there will be more than one such choice.) We showed that the greater flexibility of retrenchment could account for what was going on in a rather straightforward manner, one moreover, that readily lends itself to generalisation as a retrenchment *Atomicity Pattern*.

We then tested the pattern against a different, and if anything more challenging example, based on the Mondex Purse [SCW00, BJPS07], and found that it could cope extremely well with all the various possibilities when we viewed the fundamental Mondex transaction model as an atomic action according to our way of looking at it.

One particularly useful aspect of the *Atomicity Pattern* was its potential for coping with situations that fell short of perfect ACIDity in the concrete protocol. We showed this in the relatively simple context of the timing aspects of

the ATC application. Admittedly this is an extremely simple scenario, but it showed something of the power of the *Atomicity Pattern* to generalise across a wide variety of phenomena.

We explored the potential of the *Atomicity Pattern* to capture non-ACID aspects of transactions further, by reinterpreting the recovery aspects of Mondex transactions as long-lived compensated transactions. We saw that here too, the *Atomicity Pattern* was flexible enough and generic enough to encompass the requisite behaviour.

All of this makes us confident that the pattern will be applicable in much larger application domains where loss of ACIDity is an issue. Why? Well, all models of the kind we are considering are defined by using a collection of events or operations — a large complicated model merely has more of them, they may be more complex, and they may be organised into more layers. Provided though, that the model is sufficiently comprehensive, and captures enough of the environment if the environment is implicated in ACIDity losing behaviour, loss of ACIDity will arise through specific events. Provided we model these events appropriately, the changing properties of interest can be captured in suitable retrenchment data (i.e. the within, output, and concedes relations) attached to relevant event or operation descriptions, as we pass from an idealised model to a more realistic and imperfect one.

In this manner we anticipate that ACIDity losing phenomena in other long-lived workflows and their compensated transactions, and highly concurrent and highly distributed environments, can also be subsumed by the pattern. Let us briefly sketch how such an example might go. Consider a long-lived workflow containing online purchasing transactions. In an ideal world, items are only sold if they are in stock. However, in the real world, it may occasionally happen that the system could sell an item that did not exist due to poor stocktaking. Thus the real world could be described using two stock variables: the 'nominal stock level' used by the system, and the 'true stock level', accurate, but unknown to the system. While the latter remained positive all would be well. But as soon as the system (unknowingly) made a sale that pushed it below zero, the ACID properties would be compromised, since the sale would commit before it became known that it was invalid. In due course, further events would ensue, that (say) resulted in the purchaser being refunded. Although the *system* would not know it had transacted a rogue sale at the moment it happened, there is nothing to stop a *model of the system* having such knowledge, and thus being able to identify loss of ACIDity through appropriate retrenchment data.

Finally, we leapt to the opposite extreme as regards the size of the basic atomic action in our system model, viewing various unruly phenomena in the code-level implementations of imperfectly ACIDic software transactional memory systems, as yet other manifestations of the general phenomena captured in the retrenchment *Atomicity Pattern*. Here again we saw that the pattern was able to express the dissonances between the ideal ACIDic view of what the code was supposed to do and the unpleasant reality of a realistic implementation. Putting it all together, we regard the evidence accumulated in this paper as a solid vindication of the utility of the retrenchment *Atomicity Pattern*.

References

[ABHI08] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proc. POPL 2008*, 2008.

[Abr03] J.-R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In Araki et al. [AGM03], pages 51–74.

[ACM] J.-R. Abrial, D. Cansell, and D. Méry. Refinement and Reachability in Event-B. In *Proc. ZB 2005*, volume 3455 of *LNCS*, pages 222–241.

[AGM03] K. Araki, S. Gnesi, and D. Mandrioli, editors. *International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, Pisa, Italy, September 2003. Springer.

[AHM09] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory protection Hardware. In *Proc. PPoPP 2009*, 2009.

[BA82] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall, 1982.

[Ban09] R. Banach. Coarse Grained Retrenchment and the Mondex Denial of Service Attacks. In *Proc. IEEE TASE-09*. IEEE Computer Society Press, 2009.

[BBF+05] R. Bruni, M. Butler, C. Ferreira, T. Hoare, H. Melgratti, and U. Montanari. Comparing two Approaches to Compensable Flow Composition. In *Proc. CONCUR 2005*, 2005.

[BFH+02] M. Butler, C. Ferreira, P. Henderson, M. Chessell, C. Griffin, and D. Vines. Extending the Concept of Transaction Compensation. *IBM Systems Journal*, 47:743–758, 2002.

[BFN05] M. Butler, C. Ferreira, and M. Ng. Precise Modelling of Compensating Business Transactions and its Application to BPEL. *J.UCS*, 11:712–743, 2005.

[BHF04] M. Butler, T. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In *25 Years of CSP*, Jul. 2004.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ09] R. Banach and C. Jeske. Retrenchment and Refinement Interworking: the Tower Theorems. 2009. Available at [RET].

[BJP08] R. Banach, C. Jeske, and M. Poppleton. Composition Mechanisms for Retrenchment. *J. Log. Alg. Prog.*, 75:209–229, 2008.

[BJPS07] R. Banach, C. Jeske, M. Poppleton, and S. Stepney. Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1,1) Forward Refinements. *Fund. Inf.*, 77:29–69, 2007.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralisation of Process Nets with Centralised Control. In *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[BN97] P. A. Bernstein and E. Newcomer. *Transaction Processing*. Morgan Kaufmann, 1997.

[Bör03] E. Börger. The ASM Refinement Method. *Formal Aspects of Computing*, 15:237–275, 2003.

[BP00] R. Banach and M. Poppleton. Fragmented Retrenchment, Concurrency and Fairness. In *Proc. IEEE ICFEM2000*, pages 143–151, York, 2000. IEEE Computer Society Press.

[BP03] R. Banach and M. Poppleton. Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study. *Requirements Engineering Journal*, 8:266–288, 2003.

[BPJS] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Finite Sequence Numbers and the Tower Pattern. In *FM 2005*, volume 3582 of *LNCS*, pages 382–398. Springer.

[BPJS06a] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Finite Exception Logs, and Validating the Small. In *IEEE/NASA Software Engineering Workshop 30, 2006*, pages 234–245. IEEE Computer Society Press, 2006.

[BPJS06b] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: Hashing Injective CLEAR Codes, and Security Properties. In *2nd IEEE International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, 2006*, pages 82–90. IEEE Computer Society Press, 2006.

[BPJS07] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and Theoretical Underpinnings of Retrenchment. *Science of Computer Programing*, 67:301–329, 2007.

[BS03] E. Börger and R.F. Stärk. *Abstract State Machines. A Method for High Level System Design and Analysis*. Springer, 2003.

[BS08a] R. Banach and G. Schellhorn. Atomic Actions and their Refinement to Isolated Protocols. *Form. Asp. Comp.*, 22:33–61, 2010.

[BS08b] R. Banach and G. Schellhorn. On the Refinement of Atomic Actions. *ENTCS*, 201:3–30, 2008. Also in: University of Kent Computing Laboratory Technical Report 4-07, 168-191.

[CB04] T. Connolly and C. Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison Wesley, 2004.

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 2005.

[CSW02] D. Cooper, S. Stepney, and J. Woodcock. Derivation of Z Refinement Proof Rules. Technical Report YCS-2002-347, University of York, 2002.

[DB01] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. FACIT. Springer, 2001.

[Dep91] Department of Trade and Industry. Information Technology Security Evaluation Criteria, 1991. http://www.cesg.gov.uk/site/iacs/itsec/media/formal-docs/Itsec.pdf.

[DS09] L. Dalessandro and M. Scott. Strong Isolation is a Weak Idea. In *Proc. Transact 2009*, 2009.

[EL97] J. Eder and W. Liebhart. Workflow transactions. In *Workflow Handbook*, pages 157–163. John Wiley, 1997.

[EN03] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2003.

[FF90] N. Francez and I. Forman. Superimposition for Interactive Processes. In *Proc. CONCUR 1990*, volume 458 of *LNCS*, pages 230–245. Springer, 1990.

[GG] M. Gore and R. Ghosh. Recovery in Distributed Extended Long-lived Transaction Models. In *Proc. Sixth International Conference on Database Systems for Advanced Applications*, pages 313–320. IEEE Computer Society.

[GH08] C. George and A. Haxthausen. Specification, Proof, and Model Checking of the Mondex Electronic Purse using RAISE. *Form. Asp. Comp.*, 20:101–116, 2008.

[GMUW03] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2003.

[GR93] J. Gray and A. Reuter. *Transaction Processing*. Morgan Kaufmann, 1993.

[Hal96] A. Hall. Using Formal Methods to Develop an ATC Information System. *IEEE Software*, 13:66–76, 1996.

[HB03] T. Harris and J. Bacon. *Operating Systems: Concurrent and Distributed Software Design*. Addison Wesley, 2003.

[HF03] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. OOPSLA 2003*, 2003.

[HGS06] A. E. Haxthausen, C. George, and M. Schütz. Specification and Proof of the Mondex Electronic Purse. In *Proceedings of 1st Asian Working Conference on Verified Software, AWCVS'06, UNU-IIST Reports 348, Macau*, 2006.

[HM93] M. Herlihy and E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. 20th ISCA*, pages 289–300. 1993.

[HMPJH05] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. PPoPP 2005*, 2005.

[HPST06] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. PLDI 2006*, pages 14–25, 2006.

[HWC+04] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, Ma. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. 31st ISCA*, page 102. 2004.

[IB07] M. Isard and A. Birrell. Automatic Mutual Exclusion. In *Proc. Workshop on Hot Topics in Operating Systems 2007*, 2007.

[ISO02] ISO/IEC 13568. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics: International Standard*, 2002. http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip.

[Jes05] C. Jeske. *Algebraic Integration of Retrenchment and Refinement*. PhD thesis, University of Manchester, 2005.

[JK97] S. Jajodia and L. Kerschberg. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[JOW06] C.B. Jones, P. O'Hearne, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, 2006.

[JW08] C. Jones and J. Woodcock, (eds.). Special Issue on the Mondex Verification. *Form. Asp. Comp.*, 20(1):1–139, January 2008.

[Kat93] S. Katz. A Superimposition Control Construct for Distributed Systems. *ACM TPLAN*, 15(2):337–356, 1993.

[KHR+08] B. Khan, M. Horsnell, I. Rogers, M. Luján, A. Dinn, and I. Watson. An Object-Aware Hardware Transactional Memory. In *Proc. ICHPCC*, pages 51–58, 2008.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, 1994.

[Lon04] K. Loney. *Oracle Database 10g: The Complete Reference*. McGraw-Hill, 2004.

[LR06] J. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MBS+08] V. Menon, S. Balensiefer, T. Shpeisman, A-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. SPAA 2008*, 2008.

[Pap07] M. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2007.
[PB03] M. Poppleton and R. Banach. Structuring Retrenchments in B by Decomposition. In Araki et al. [AGM03], pages 814–833.
[Ray88] M. Raynal. *Distributed Algorithms and Protocols*. Wiley, 1988.
[RET] Retrenchment Homepage. http://www.cs.man.ac.uk/retrenchment.
[RG02] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. 10th SASPLO*, pages 5–17. 2002.
[RRP+07] H. Ramadan, C. Rossbach, D. Porter, Ow. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory for an Operating System. In *Proc. 34th ISCA*, pages 92–103, 2007.
[SBG05] A. Silberschatz, P. Baer, and G. Gagne. *Operating System Concepts*. Wiley, 2005.
[Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *JUCS*, 7:952–979, 2001.
[Sch05] G. Schellhorn. ASM Refinement and Generalisations of Forward Simulation in Data Refinement: A Comparison. *Theoretical Computer Science*, 336:403–435, 2005.
[SCW00] S. Stepney, D. Cooper, and J. Woodcock. An Electronic Purse: Specification, Refinement and Proof. Technical Report PRG-126, Oxford University Computing Laboratory, 2000.
[SGH+07] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In *Proc. Dagstuhl Workshop on Rigorous Methods for Software Construction and Analysis 2007*, LNCS. Springer, 2007.
[SGHR06] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In *Proc. FM 2006*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.
[SGMA89] K. Salem, H. Garcia-Molina, and R. Alonso. Altruistic Locking: A Strategy for Coping with Long Lived Transactions. In *Proc. Second International Workshop on High Performance Transaction Systems*, volume 359 of *LNCS*, pages 175–199. Springer, 1989.
[SMAT+07] T. Shpeisman, V. Menon, A-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proc. PLDI 2007*, 2007.
[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, second edition, 1992.
[WB07] J. Woodcock and R. Banach. The Verification Grand Challenge. *JUCS*, 13(5):661–668, May 2007.
[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, 1996.
[Woo06] J.C.P. Woodcock. First Steps in the Verified Software Grand Challenge. *IEEE Computer*, 39(10):57–64, 2006.
[WS] Web Services Org. http://www.webservices.org/.
[WSC+08] J. Woodcock, S. Stepney, D. Cooper, J. Clark, and J. Jacob. The Certification of the Mondex Electronic Purse to ITSEC Level E6. *Form. Asp. Comp.*, 20:5–19, 2008.
[WV02] G. Weikum and G. Vossen. *Transaction Processing*. Morgan Kaufmann, 2002.
[YBM+07] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *Proc. 13th ISHPCA*. 2007.
[ZBM03] P. Zikopoulos, G. Baklarz, and R. Melnyk. *Official Guide to DB2 Version 8*. Prentice Hall, 2003.