Under consideration for publication in Formal Aspects of Computing

Atomic Actions, and their Refinements to Isolated and Not-So-Isolated Protocols

Richard Banach[1] and Gerhard Schellhorn[2]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
[2]Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg, Germany

Abstract. Inspired by the properties of the refinement development of the Mondex Electronic Purse, we view an isolated atomic action as a family of transitions with a common before-state, and different after-states corresponding to different possible outcomes when the action is attempted. We view a protocol for an atomic action as a computation DAG, each branch of which achieves in several steps, one of the outcomes of the atomic action. We show that in this picture, the protocol can be viewed as a relational refinement of the atomic action in a number of ways. Firstly, it yields a 'big diagram' simulation à la ASM. Secondly, it yields a 'small diagram' simulation, in which the atomic action is synchronised with an individual step along each path through the protocol, and all the other steps of the path simulate skip. We show that provided each path through the protocol contains one step synchronised with the atomic action, the choice of synchronisation point can be made freely. We describe the relationship between such synchronisations and forward and backward simulations. We relate this theory to serialisations of system runs containing interleaved multiple transactions, and show how existing Mondex refinements embody the ideas developed.

We then generalise the picture to encompass not-so-isolated atomic actions, exemplified by another motivating example, the lock-free stack, in which arbitrary numbers of agents may collaborate and/or interfere as the protocol runs. The working of the lock-free stack (and its enhancement to the elimination stack) are described using event structures. We then give an elaboration of the preceding framework, enriched with additional detail concerning agents, and aimed at capturing the new phenomena in not-so-isolated atomic actions. We revisit the earlier results in the new setting. Not only do the generalisations adequately describe the lock-free and elimination stack scenarios, but they also cope well with certain non-2-phase aspects of Mondex.

Keywords: Atomic Actions, Protocols, Synchronisation, Serialisation, 2-Phase Protocols, Non-2-Phase Protocols, Forward and Backward Simulation, Refinement, Mondex, Lock-Free Stack, Elimination Stack.

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

1. Introduction

The Mondex Electronic Purse was developed formally in the mid-1990s using Z refinement. It was one of the first developments to achieve an ITSEC E6 security rating [DoTaI91].[1] Rather unusually for a commercial product, a sanitised version of the core of the formal development was made publicly available [SCW00]. Since then it has been a fertile ground for formal methods researchers — the original, human-built proofs of the security properties have been subjected to re-examination by contemporary techniques, and have stood up extremely well to the fiercest tool-based scrutiny achievable today, the first such mechanical verification being [SGHR06b].

The Mondex formal development featured a refinement proof from an atomic abstract model to a multi-step protocol at the concrete level. The principal component of this refinement proof was a backward simulation from abstract to concrete. At the time of the original development, the development team did try to construct a forward simulation, but were not successful — for a long time it was believed that a forward simulation refinement was impossible. It is nowadays known that a forward simulation is entirely possible, and more than one of them is now available in the literature [BPJS07, SGH+07, HGS06].

In this paper we explore the wider question regarding possible kinds of simulation for the refinement of an atomic action into a multi-step protocol, in order to settle the matter in the general case. We do this in the simplest possible relational framework in order to avoid complications that would distract from the main point.

In Mondex, the original refinement was done in a $(1,1)$ manner, i.e. single concrete steps were made to refine single abstract ones. Consequently, since overall, there are more concrete steps than abstract ones, many concrete steps had to refine skip. Of course, one advantage of the $(1,1)$ strategy is that, in the face of malevolent users or an unpredictable environment, the concrete protocol can be proved to refine the abstract atomic action, no matter how such a user might interrupt the intended playing out of the protocol — since every possible sequence of concrete steps that can be executed, corresponds to *some* abstract execution, even if it is one consisting entirely of skips.

In this, the original framework, the backward simulation correlated with an *early* synchronisation, i.e. the single non-trivial abstract step was $(1,1)$ matched with a step that occurred early in protocol runs. By contrast, the more recently discovered forward simulations correlate with a *late* synchronisation, namely, the various possible non-trivial abstract steps are $(1,1)$ matched with steps that occur late in protocol runs. Given the past uncertainty regarding forward and backward simulations in such contexts, one of our aims in this paper is to give a general treatment.

Mondex has the *isolated protocol* property. In other words, once the protocol has started, it can be viewed as running to its conclusion in a manner free from outside interference. But these are not the only protocols in town. Another of our aims in this paper is to extend the theoretical approach we develop for Mondex, to also deal with *not-so-isolated protocols*. In a not-so-isolated protocol, various agents, other than the ones engaged in achieving the protocol's goals, can interfere in the running of the protocol, because they too wish to achieve similar goals and the protocol's design itself permits such interference. Very often such permissive designs arise because the steps of the protocol are at the granularity of individual machine instructions (or very small runs of machine instructions), so that the overhead of installing proper locking mechanisms to ensure true isolation of such instructions from one another is completely inappropriate.

One aspect of not-so-isolated protocols that makes them different from isolated protocols is that we have to take more note of the agents performing various actions. If a protocol is isolated, it is not so urgent to know which agent is doing what — after all, they are all co-operating towards the same goal. However with not-so-isolated protocols, while some agents are working towards the protocol's goals, others may be detracting from them (albeit quite innocently and in a well understood manner), so knowing what is going on at any point is more pressing, and the theory needs to be more sensitive to these aspects.

Another issue that gains a different slant in the not-so-isolated protocol arena is serialisability. Thus if a protocol is isolated, the same mechanisms that ensure isolation can usually be deployed to ensure that any interleavings that a system run permits, have good serialisability properties. However, if there is potential for interference in a protocol, there exists the potential for more subtle issues surrounding serial semantics too, since the locking mechanisms that are available are, almost by definition, weaker.

We base our work on not-so-isolated protocols on another example from the literature, the lock-free stack, and its enhacement the elimination stack, specifically as formulated in [CG07, CG06]. In these protocols, interference can occur, but it happens in a controlled way, and we develop our extended theory with an eye to giving a good account of them. Interestingly enough, the serialisability properties of the stack examples are stronger than those of (certain

[1] Nowadays, national standards like ITSEC have been superseded by the ISO Common Criteria standard [Int05]. The highest ITSEC level, E6, corresponds to the highest Common Criteria level, EAL7.

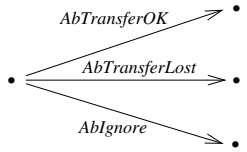

Fig. 1. The Mondex atomic actions.

parts of) the Mondex protocol. This is because in the days that Mondex was developed, the computational power of smartcards was much less than now, and it was judged acceptable for certain null parts of the protocol to fall short of the conventional 2-phase locking discipline, for efficiency's sake. Thus the most extreme extensions of our theory are brought to bear on Mondex once more.

The rest of this paper is as follows. In Section 2 we outline the operation of our motivating example, the Mondex Purse. In Section 3 we develop a theory of the refinement of a non-deterministic atomic action to a multi-step protocol in terms of computation DAGs. This explores the way that the single atomic action can be synchronised with an individual step of the protocol in a $(1,1)$ refinement, and we see that there are a large number of possibilities for this which we call synchronisation assignments (SAs). We see that SAs are related to the possible choices of forward or backward simulations, according to the manner in which abstract outcomes are related to the details of the SA. In Section 4 we relate the rather abstract computation DAG view of protocols to a more conventional one, using event structures, and show that the histories generated by event structures yield computation trees in a natural way. In Section 5 we relate the preceding theory of an isolated protocol run to the more global picture needed to embed protocol runs into system runs, and we explore serialisability and the 2-phase property. In the following Section 6 we apply the theory developed to the various refinements of Mondex available today, noting finally that there are in fact some non-2-phase corners of the original Mondex protocol (though none of them achieve anything observable at the abstract level, and are thus tolerated).

In Section 7 we start the work of extending our theory by describing the lock-free and elimination stacks. The event structure formulation introduced in Section 4 is extended, and provides the most compact way of describing these examples without descending to the level of actual code. Section 8 reflects on the properties of such protocols, and comes up with a generalisation of the formulation of Section 3 which encompasses the new features. As noted above, this is a much more agent-aware formulation in which 'template' serialisation properties of the protocol can be formulated. The serialisation of actual system runs is considered in Section 9, being an elaboration of the 2-phase version in Section 5. In Section 10 we apply the extended theory to the lock-free and elimination stacks, and to the non-2-phase Mondex fragments.

A portion of the preceding theory has been mechanically verified using KIV, and in Section 11 we review what has been achieved here. The final section concludes.

2. Mondex: A Motivating Example

Fundamentally, Mondex is a *smartcard purse*. Since it is a *purse*, it contains real money, and since it is a *smartcard*, it contains the money in digital form. This money is designed to be transferable from purse to purse. As for real money, the intention is that such transfers are normally performed in exchange for some desired purpose such as the purchase of goods or services, but equally —just as for real money— it is not the responsibility of the money itself to ensure that the transfer in which it engages is of a genuine nature. The only concern of money in general and of Mondex money in particular, is that it should be *unforgeable*.

The major objective of the original Mondex development was to develop a protocol for money transfer that ensured that:

1. Mondex money was unforgeable, even in the face of incomplete execution of the protocol or of malicious behaviour of the environment;
2. any full or partial run of the protocol is equivalent to either a successful money transfer, or a traceably (and thus recoverably) lost-in-transit money transfer, or a null action.

These two properties are what make Mondex credible in the face of customer requirements: the first property, unforgeability, gives confidence in the value of Mondex money; while the second property, atomicity, gives comprehensibility

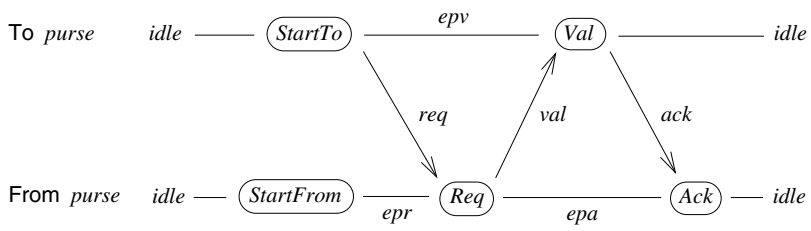

Fig. 2. The Mondex concrete protocol.

when compared with the behaviour of conventional financial transactions. Fig. 1 shows the atomic abstraction that the Mondex protocol ensures, reflecting the three possibilities given in (ii) above. In Fig. 1 the nodes are states, and the arrows are the different atomic actions that the concrete protocol refines.

The essence of the Mondex concrete protocol is illustrated in Fig. 2 in activity diagram style. The source purse is the From purse while the destination purse is the To purse. The protocol begins with the two *Start* events (initiated from the environment as a result of the purses' owners typing in appropriate instructions at the interface device (the wallet) into which the two purses have been inserted). These are the *StartTo* event, performed by the To purse, and *StartFrom* event, performed by the From purse, both of which take their respective purse from the idle state to a 'busy' state: the *epr* state (expecting payment request) for the From purse, and the *epv* state (expecting payment value) for the To purse. The *StartTo* event sends a *req* message to the From purse. Upon arrival of the *req* message, the From purse performs a *Req* event and dispatches the money in a *val* message to the To purse, itself passing into the *epa* (expecting payment acknowledgement) state. Upon arrival of the *val* message, the To purse performs a *Val* event and sends an *ack* message to the From purse, itself passing back into the idle state. Receipt of the *ack* message in the *Ack* event by the From purse completes the protocol, and the From purse too passes back into the idle state. Note that in Fig. 2, the nodes are now events, edges are states, and arrows are messages.

The preceding described the workings of a successful run of the protocol. Beyond that, all events after the *Start* events can be replaced by *Abort* events, corresponding to runs of the protocol that are unsuccessful for whatever reason. The fact that despite *Abort* events, the protocol still enjoys the unforgeability and atomicity properties, is what makes Mondex non-trivial theoretically. However, the details of how this comes about do not concern us in this paper.

A further issue is that the Mondex protocol is *isolated*, i.e. once the protocol has commenced, the two purses take note only of the arrival of the next message expected in the playout of the protocol, and of calls to *Abort*, ignoring all other messages or calls from the environment and reserving the option of responding to such unexpected events by performing a self-initiated *Abort* whenever appropriate.

In this paper, rather than being concerned with *proving* that the atomicity and isolatedness properties are enjoyed by the protocol, we take properties such as these for granted, and instead, take an interest in simulation-theoretic properties —in a general sense, and for their own sake— of the refinement of an atomic action to a protocol with characteristics such as Mondex's. The isolated property makes these simulation-theoretic properties particularly convenient to study.

3. Isolated Atomic Actions and their Protocols

For both protocols and atomic actions, we will specify the transitions involved using a relational approach. The following statements summarise the assumptions we make about this setup.

Assumptions 3.1.

1. Relations are represented by predicates whose variables take values in suitable types.
2. Each relation used is deterministic, i.e. for each collection of values for the domain variables of the predicate representing the relation, there is a unique collection of values for the codomain variables that makes the relation true.
3. For each relation, for all values of domain and codomain variables that make the relation true, the domain values are reachable from an initial state.
4. Where nondeterminism (whether at the atomic or the protocol level) is needed, it is handled by having different relations for different outcomes. We assume nondeterminism is always finite.
5. Both atomic actions and protocols are represented by transition systems. At the atomic level, atomic actions are

given by a collection of predicates whose interpretations are restricted to shallow computation forests (i.e. all maximal paths of length 1). At the protocol level, protocols are given by a collection of predicates whose interpretations are restricted to DAGs, all of whose paths are finite. A choice of initial state for a root of the interpreting forest of an atomic action picks out a unique tree, called the valid tree. A choice of an initial state for a root of the protocol DAG picks out a unique (maximal reachable) subDAG of the interpreting DAG, called the valid DAG.

Thus an atomic action will be specified by a finite collection of deterministic predicates $At_k(u,i,o,u') \;\; k = 1\ldots$, in which u and u' are (variables denoting) the before- and after- states of the atomic action, i and o are the input and output of the action (these may in fact denote sequences (or more complex structures) of input and output values corresponding to the finer grained events in the protocol, if convenient), and the label k distinguishes the different deterministic outcomes for the same starting conditions. All together, the complete atomic specification of the protocol becomes:

$$Atomic(u,i,o,u') \equiv At_1(u,i,o,u') \vee At_2(u,i,o,u') \vee \ldots \tag{1}$$

where

$$(\forall\, u,i \bullet At_k(u,i,o_1,u'_1) \wedge At_k(u,i,o_2,u'_2) \Rightarrow o_1 = o_2 \wedge u'_1 = u'_2) \tag{2}$$

(and where it turns out that (2) is not actually needed in the ensuing mathematics, but helps for a convenient mental picture).

At the protocol level, the individual steps are described by a collection of deterministic predicates $St_\rho(v,j,p,v')$ where v and v' are the before- and after- states of the step, and j and p are the input and output of the step. The label ρ is an identifier which discriminates between different nondeterministic outcomes from the same before-state and input, and is required to be different for each step along a path through the protocol DAG,[2] but is otherwise available to conveniently label steps from an applications perspective.

(Forward) paths through the protocol computation DAG are described by compound predicates:

$$\begin{aligned} &FPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I,j_1,p_1,v_1,j_2,p_2,v_2,\ldots,v_{t-1},j_t,p_t,v_t) \equiv \\ &\quad St_\alpha(v_I,j_1,p_1,v_1) \wedge St_\beta(v_1,j_2,p_2,v_2) \wedge \ldots \wedge St_\gamma(v_{t-1},j_t,p_t,v_t) \end{aligned} \tag{3}$$

in which v_I is a possible initial state of the protocol, α labels a possible first step of the protocol, β labels a possible successor step of the α step of the protocol, and so on. As (3) indicates, if a step has a successor, the before-state of the successor must match the after-state of its predecessor. The length of the sequence of labels in the subscript of $FPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ must match both the number of inputs and outputs, and be one less than the number of states, in the argument list.

Maximal paths arise in the obvious way:

$$\begin{aligned} &MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(\ldots) \equiv \\ &\quad FPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(\ldots) \wedge (\langle\alpha,\beta,\ldots,\gamma\rangle \text{ has no proper extension in the computation graph}) \end{aligned} \tag{4}$$

From maximal and non-maximal paths, we can implicitly define a predicate *BPath* (backward paths) that describes extensions of non-maximal forward paths:

$$\begin{aligned} &MPath_{\langle\alpha,\beta,\ldots,\gamma,\delta,\epsilon\ldots,\zeta\rangle}(v_I,j_1,p_1,v_1,\ldots,j_t,p_t,v_t,j_{t+1},p_{t+1},v_{t+1}\ldots,v_F) \equiv \\ &\quad FPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I,j_1,p_1,v_1,\ldots,j_t,p_t,v_t) \wedge BPath_{\langle\delta,\epsilon\ldots,\zeta\rangle}(v_t,j_{t+1},p_{t+1},v_{t+1}\ldots,v_F) \end{aligned} \tag{5}$$

In (5), v_F is a possible final state of the protocol.

Finally, maximal paths give rise to the predicate $Protocol(v_I,js,ps,v_F)$, where v_F is a possible final state of the protocol,[3] given by taking the disjunction over all maximal paths, existentially quantifying all intermediate states, and repackaging the inputs and outputs into sequences:

$$\begin{aligned} &Protocol(v_I,js,ps,v_F) \equiv \\ &\bigvee_{\left\{\substack{\text{maximal} \\ \langle\alpha,\beta,\ldots,\gamma\rangle}\right\}} \left(\begin{array}{l} (\exists\; j_1,p_1,v_1,j_2,p_2,v_2,\ldots,v_{t-1},j_t,p_t \bullet \\ \quad MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I,j_1,p_1,v_1,j_2,p_2,v_2,\ldots,v_{t-1},j_t,p_t,v_F) \\ \quad \wedge\; js = \langle j_1,j_2,\ldots,j_t\rangle \wedge ps = \langle p_1,p_2,\ldots,p_t\rangle) \end{array} \right) \end{aligned} \tag{6}$$

[2] As for (2), determinism and path-uniqueness are not strictly necessary for ρ, but are conceptually convenient.

[3] Initial and final states of the protocol coincide exactly with the root and leaf states of the protocol computation graph.

The fact that the protocol implements the atomic action is captured by relating the two via a retrieve relation R, input and output relations *Input* and *Output*, and demanding that an ASM-style [BS03] 'big-step' proof obligation holds. The retrieve relation is required to satisfy:

Assumptions 3.2.

1. $R(u,v)$ is a function from protocol states v to atomic states u. (7)
2. If v is a protocol state and v_{I1} and v_{I2} are initial protocol states, then
   $$FPath_{\langle\ldots\rangle}(v_{I1}\ldots v) \wedge FPath_{\langle\ldots\rangle}(v_{I2}\ldots v) \Rightarrow (\exists\ u_I \bullet R(u_I,v_{I1}) \wedge R(u_I,v_{I2})) \quad (8)$$
   (where u_I is obviously unique because of (7)).
3. $R(u,v)$ is 'not too big,' i.e. it concerns just the 'states of interest' for the overall protocol, i.e. the initial and final states:
   $$R(u,v) \Rightarrow (\exists\ js,ps,\tilde{v} \bullet Protocol(v,js,ps,\tilde{v}) \vee Protocol(\tilde{v},js,ps,v)) \quad (9)$$

(As for (2), it turns out that (9) is not needed later, but helps for a convenient mental picture.) The big-step PO is now:

$$Protocol(v_I,js,ps,v_F) \Rightarrow (\exists\ u_I,i,o,u_F \bullet R(u_I,v_I) \wedge Input(i,js) \wedge Atomic(u_I,i,o,u_F) \wedge Output(o,ps) \wedge R(u_F,v_F)) \quad (10)$$

Conditions (9) and (10) ensure that the hypotheses and conclusions of the big-step PO are valid exactly when the simulation predicate Σ:

$$\Sigma(u_I,i,o,u_F,v_I,js,ps,v_F) \equiv Atomic(u_I,i,o,u_F) \wedge Protocol(v_I,js,ps,v_F) \wedge R(u_I,v_I) \wedge Input(i,js) \wedge Output(o,ps) \wedge R(u_F,v_F) \quad (11)$$

is true in the given types.

Now that we have connected together the atomic and finegrained descriptions of the protocol, our aim is to develop a general way of seeing how *some individual step* of a maximal path may be viewed as refining the atomic action, and the consequences of such a view. First we develop some technical machinery in the shape of past and future oriented retrieve relations. Then we introduce synchronisation assignments, which delimit exactly how the choices of individual step within the protocol computation graph may be made. Finally we explore the consequences of these choices for proving the refinement via forward and backward simulation.

First we get the 'past oriented' retrieve relation R^P:

$$R^P(u_I,v_t) \equiv (\exists\ v_I,j_1,p_1,v_1,\ldots,j_t,p_t,\langle\alpha,\beta,\ldots,\gamma\rangle \bullet R(u_I,v_I) \wedge FPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I,j_1,p_1,\ldots,j_t,p_t,v_t)) \quad (12)$$

and the 'future oriented' retrieve relation R^F:

$$R^F(u_F,v_t) \equiv (\exists\ j_{t+1},p_{t+1},v_{t+1}\ldots,v_F,\langle\delta,\epsilon,\ldots,\zeta\rangle \bullet BPath_{\langle\delta,\epsilon\ldots,\zeta\rangle}(v_t,j_{t+1},p_{t+1},v_{t+1}\ldots,v_F) \wedge R(u_F,v_F)) \quad (13)$$

It is now easy to show the following:

Proposition 3.3.

$$R^P(u_I,v_t) \wedge R^F(u_F,v_t) \Rightarrow (\exists\ i,o \bullet Atomic(u_I,i,o,u_F)) \quad (14)$$
$$R^P(u_I,v_t) \Rightarrow (\exists\ i,o,u_F \bullet Atomic(u_I,i,o,u_F) \wedge R^F(u_F,v_t)) \quad (15)$$
$$R^F(u_F,v_t) \Rightarrow (\exists\ u_I,i,o \bullet R^P(u_I,v_t) \wedge Atomic(u_I,i,o,u_F)) \quad (16)$$

The proofs are similar to the proofs of the more interesting following result:

Theorem 3.4.

$$R^P(u_I,v_{t-1}) \wedge St_\rho(v_{t-1},j_t,p_t,v_t) \wedge R^F(u_F,v_t) \Rightarrow (\exists\ i,o,js^P,js^F,ps^P,ps^F \bullet Input(i,js^P{::}\langle j_t\rangle{::}js^F) \wedge Atomic(u_I,i,o,u_F) \wedge Output(o,ps^P{::}\langle p_t\rangle{::}ps^F)) \quad (17)$$
$$R^P(u_I,v_{t-1}) \wedge St_\rho(v_{t-1},j_t,p_t,v_t) \Rightarrow (\exists\ i,o,u_F,js^P,js^F,ps^P,ps^F \bullet \wedge Input(i,js^P{::}\langle j_t\rangle{::}js^F) \wedge Atomic(u_I,i,o,u_F) \wedge Output(o,ps^P{::}\langle p_t\rangle{::}ps^F) \wedge R^F(u_F,v_t)) \quad (18)$$
$$St_\rho(v_{t-1},j_t,p_t,v_t) \wedge R^F(u_F,v_t) \Rightarrow (\exists\ u_I,i,o,js^P,js^F,ps^P,ps^F \bullet R^P(u_I,v_t) \wedge Input(i,js^P{::}\langle j_t\rangle{::}js^F) \wedge Atomic(u_I,i,o,u_F) \wedge Output(o,ps^P{::}\langle p_t\rangle{::}ps^F)) \quad (19)$$

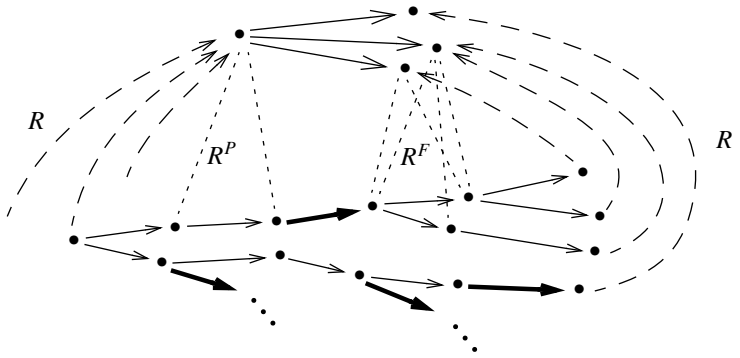

Fig. 3. A synchronisation assignment for a computation tree. The elements of the synchronisation assignment are shown bold.

Proof. For (17), from $R^P(u_I, v_{t-1})$ we know that there is a path through the computation tree from an initial v_I to v_{t-1}, satisfying (3), and such that $R(u_I, v_I)$ holds. Evidently $St_\rho(v_{t-1}, j_t, p_t, v_t)$ extends that path. From $R^F(u_F, v_t)$ we know that there is a completion of this path to a maximal path from v_I to some final v_F. This maximal path enables us to derive $R(u_F, v_F)$, and provides the witnessing js^P, js^F, ps^P, ps^F so that with j_t, p_t we can assemble $js = js^P ::\langle j_t\rangle :: js^F$ and $ps = ps^P ::\langle p_t\rangle :: ps^F$, and then assert $Protocol(v_I, js, ps, v_F)$.

Since we have $Protocol(v_I, js, ps, v_F)$, we can apply (10). The conclusions of (10) yield $R(\tilde{u}, v_I)$ for some $\tilde{u}$; and since R is functional (7), we must have $u_I = \tilde{u}$. The conclusions of (10) also yield $Atomic(u_I, i, o, \tilde{u}')$ and $R(\tilde{u}', v_F)$ for some $\tilde{u}'$. Again, since R is functional, we must have $u_F = \tilde{u}'$. From $Protocol(v_I, js, ps, v_F)$ we can also deduce $Input(i, js)$ and $Output(o, ps)$.

For (18), the argument is similar except that we do not have to use the functional nature of R to argue $u_F = \tilde{u}'$, since u_F is existentially quantified in the conclusion.

For (19), we note first that by Assumptions 3.1.3, v_t is reachable from some initial v_I. We use this to assert a u_I such that $R^P(u_I, v_t)$ holds, after which we argue as for case (17). We are done. □

Proposition 3.4 is a crucial observation, since it enables an arbitrary protocol step $St_\rho(v_{t-1}, j_t, p_t, v_t)$ to be singled out and made to correspond with a suitable abstract one $Atomic(u_I, i, o, u_F)$. For such a $St_\rho(v_{t-1}, j_t, p_t, v_t)$ step, let $\text{Outcomes}(St_\rho, u_I)$ (with v_{t-1}, j_t, p_t, v_t understood) be given by:

$$\text{Outcomes}(St_\rho, u_I) = \{u_F \mid (\exists\ v_F \bullet R^P(u_I, v_{t-1}) \wedge St_\rho(v_{t-1}, j_t, p_t, v_t) \wedge R^F(u_F, v_t))\} \tag{20}$$

and $\text{OD}(St_\rho, u_I)$ (outcome determinism of St_ρ, given u_I) be given by:

$$\text{OD}(St_\rho, u_I) = \mid \text{Outcomes}(St_\rho, u_I) \mid \tag{21}$$

If $\text{OD}(St_\rho, u_I) = 1$ we say that St_ρ is outcome deterministic at u_I (St_ρ is OD at u_I), whereas if $\text{OD}(St_\rho, u_I) > 1$ we say that St_ρ is outcome nondeterministic at u_I (St_ρ is ON at u_I).

Definition 3.5. Given an initial v_I, a synchronisation assignment ($\text{SA}(v_I)$) for the relevant valid DAG of a protocol computation DAG is a subset of its steps, such that for each maximal path through the valid DAG from v_I, exactly one of its steps is in $\text{SA}(v_I)$. Steps in $\text{SA}(v_I)$ are called SA steps.

Fig. 3 shows a synchronisation assignment. The many-level computation graph at the bottom (which happens to be a tree) has thickened arrows which are the elements of the SA. The atomic action is at the top and plays no specific part in the SA itself. Dashed arrows show the functional big-step retrieve relation R, while the dotted lines show some pieces from the R^P and R^F relations, for convenience below.

Definition 3.6. Given a protocol computation graph, an intial state v_I for the protocol, the atomic intial state u_I such that $R(u_I, v_I)$ holds, and a synchronisation assignment for the valid DAG determined by v_I, the steps of the valid DAG are classified as follows:

1. If a step is in the SA and is OD at u_I, it is called an outcome deterministic forward synchronisation (ODFS) step.
2. If a step is in the SA and is ON at u_I, it is called an outcome nondeterministic forward synchronisation (ONFS) step.
3. If a step is an immediate or later successor of an ONFS step, it is called a backward skip (BS) step.

4. Every step not covered by 1-3 is called a forward skip (FS) step.

This definition shows that every path through the protocol computation tree can be described by the following regular expression:

$$FS* \; ; \; (\; ODFS \; ; \; FS* \; + ONFS \;) \; ; \; BS* \tag{22}$$

Our aim is to show that when given a big-diagram refinement of an atomic action to a protocol of the kind we have described, if we wish to break the big-diagram refinement down into a collection of small-diagram refinements of zero or one atomic action steps to individual steps of the protocol, one can always use forward simulation reasoning, except for the BS steps. In fact one can use forward simulation reasoning for all steps except *branching BS steps* (a term explained below), though it comes at a price. Likewise, we have the option of using backward simulation reasoning for *all* steps if we so wish. We discuss these points later.

Definition 3.7. Assume given an abstract operation $AOp(u,i,o,u')$, a concrete operation $COp(v,j,p,v')$, and retrieve, input and output relations, $R^1(u,v)$, $In^1(i,j)$ and $Out^1(o,p)$. Then *AOp* forward simulates *COp* iff:

$$R^1(u,v) \wedge COp(v,j,p,v') \Rightarrow (\exists \; i,o,u' \; \bullet \; In^1(i,j) \wedge AOp(u,i,o,u') \wedge Out^1(o,p) \wedge R^1(u',v')) \tag{23}$$

And *AOp* backward simulates *COp* iff:

$$COp(v,j,p,v') \wedge R^1(u',v') \Rightarrow (\exists \; u,i,o \; \bullet \; R^1(u,v) \wedge In^1(i,j) \wedge AOp(u,i,o,u') \wedge Out^1(o,p)) \tag{24}$$

In both cases, $In^1(i,j)$ and/or $Out^1(o,p)$ can be omitted where there is no input and/or output from *AOp* and/or *COp*, as applicable.

Theorem 3.8. Let there be a big-step refinement of an atomic action *Atomic* to a protocol *Protocol*, given by a retrieve relation *R* and input and output relations *Input* and *Output*, so that (10) holds. Let v_I be a fixed initial state such that $R(u_I,v_I)$ holds, and let $\mathrm{SA}(v_I)$ be a synchronisation assignment for the valid DAG rooted at v_I. Then the refinement of *Atomic* to *Protocol* can be decomposed into single step simulations such that:

1. If an FS step occurs before an SA step, it is forward simulated by the identity operation on u_I.
2. If an FS step occurs after an SA step, it is forward simulated by the identity operation on u_F, where u_F is some outcome of *Atomic*.
3. If St_ρ is an SA step, it is forward simulated by $Atomic(u_I,i,o,u_F)$ for every u_F in $\mathrm{Outcomes}(St_\rho,u_I)$.
4. Every BS step is backward simulated by the identity operation on some u_F.

Proof. We start by defining R^1, which is:

$$\begin{aligned} R^1(u,v) \equiv (&\exists \text{ a maximal path from some initial } \tilde{v}_I, \text{ and} \\ &((v \text{ precedes an SA step along this path, and } R^P(u,v) \text{ holds}), \vee \\ &(v \text{ follows an SA step along this path, and } R^F(u,v) \text{ holds}))) \end{aligned} \tag{25}$$

Also we must define the single step input and output relations In^1 and Out^1; these however are only needed for the SA steps themselves.

$$In^1(i,j) \equiv (\exists \text{ an SA step } St_\rho(v_{t-1},j,p_t,v_t), js^B, js^F \bullet Input(i, js^P{::}\langle j\rangle{::}js^F)) \tag{26}$$

$$Out^1(o,p) \equiv (\exists \text{ an SA step } St_\rho(v_{t-1},j_t,p,v_t), ps^B, ps^F \bullet Output(o, ps^P{::}\langle p\rangle{::}ps^F)) \tag{27}$$

In fact we prove slightly more than we strictly need.

For 1, let $St_\rho(v_{t-1},j_t,p_t,v_t)$ be the FS step in question. Since the SA is defined with respect to paths reachable from v_I, and FS steps are defined with respect to the SA, v_{t-1} must be reachable from v_I. To prove forward simulation, assume $R^1(u,v_{t-1})$ holds. Then there is a maximal path from some initial $\tilde{v}_I$ that reaches v_{t-1} such that $R^P(u,v_{t-1})$ holds. From (12) there is a path from some initial $\tilde{\tilde{v}}_I$ that reaches v_{t-1} such that $R(\tilde{u}_I,\tilde{\tilde{v}}_I)$ holds for some initial $\tilde{u}_I$. By (7) and (8), $\tilde{u}_I = u = u_I$. So in fact $R^1(u_I,v_{t-1})$ and $R^P(u_I,v_{t-1})$ both hold. Since $St_\rho(v_{t-1},j_t,p_t,v_t)$ obviously extends the paths that witness $R^P(u_I,v_{t-1})$, the extensions witness $R^P(u_I,v_t)$ and $R^1(u_I,v_t)$ too, which is what is required for forward simulation of the identity on u_I.

For 2, let $St_\rho(v_{t-1},j_t,p_t,v_t)$ be the FS step in question. Since it occurs after an SA step, it must again be reachable from v_I. To prove forward simulation, assume $R^1(u,v_{t-1})$ holds. Then there is a maximal path from some initial $\tilde{v}_I$ that reaches v_{t-1} such that $R^F(u,v_{t-1})$ holds. From (13) there is a path from v_{t-1} to some final v_F such that $R(u_F,v_F)$ holds,

where u_F is the unique abstract outcome, that witnesses that the SA step that $St_\rho(v_{t-1}, j_t, p_t, v_t)$ follows, is outcome deterministic. By (7), $u = u_F$, so that $R^F(u_F, v_{t-1})$ holds, whereby $R^1(u_F, v_{t-1})$ holds too. Truncating the first step of the path from v_{t-1} to v_F that witnesses $R(u_F, v_F)$, gives a path that witnesses $R^F(u_F, v_t)$ and hence $R^1(u_F, v_t)$, which is what is required for forward simulation of the identity on u_F.

For 3, let $St_\rho(v_{t-1}, j_t, p_t, v_t)$ be the SA step in question. Obviously it is reachable from v_I. To prove forward simulation, assume $R^1(u, v_{t-1})$. Then we can deduce $R^1(u_I, v_{t-1})$ and $R^P(u_I, v_{t-1})$ exactly as in case 1. For any u_F in Outcomes(St_ρ, u_I), we know that $Atomic(u_I, i, o, u_F)$ holds. Also, we can deduce $R^F(u_F, v_t)$ and hence $R^1(u_F, v_t)$ exactly as in case 2. Since $St_\rho(v_{t-1}, j_t, p_t, v_t)$ occurs on a maximal path from v_I to v_F, the totality of inputs along the path, both js^P before j_t, and js^F after j_t, will witness that $Input(i, js^P :: \langle j_t \rangle :: js^F)$ holds, giving $In^1(i, j_t)$ as required. The reasoning for outputs is similar. So we have all the conclusions of (23), which is what is required for forward simulation of $Atomic(u_I, i, o, u_F)$.

For 4, let $St_\rho(v_{t-1}, j_t, p_t, v_t)$ be the BS step in question. Since it occurs after an SA step, it must be reachable from v_I. To prove backward simulation, assume $R^1(u, v_t)$ holds. Then there is a maximal path from some initial $\tilde{v}_I$ that reaches v_t such that $R^F(u, v_t)$ holds. From (13) there is a path from v_t to some final v_F such that $R(u_F, v_F)$ holds, where u_F is some abstract outcome, that witnesses that the SA step that $St_\rho(v_{t-1}, j_t, p_t, v_t)$ follows, is outcome nondeterministic. By (7), $u = u_F$ for some such u_F, so let us assume that $R^F(u_F, v_t)$ holds, whereby $R^1(u_F, v_t)$ holds too. Prepending $St_\rho(v_{t-1}, j_t, p_t, v_t)$ to the path from v_t to v_F that witnesses $R(u_F, v_F)$, gives a path that witnesses $R^F(u_F, v_{t-1})$, and hence $R^1(u_F, v_{t-1})$, which is what is required for backward simulation of the identity on u_F. □

Since at both abstract and protocol levels, the transpose of the step relation is a partial function, backward simulation is always aligned with a decrease of nondeterminism in both abstract and protocol transition functions. Therefore we get the following (cf. [LV93]).

Corollary 3.9. Under the assumptions of Theorem 3.8, one can always use single step backward simulations throughout.

Corollary 3.9 might seem strange in the light of the well known fact that backward simulation alone is not complete for data refinement. The explanation comes from the fact that we have an asymmetry between forward and backward directions in our setup. While we can never lose 'abstract backward nondeterminism' by simulating the protocol backward (due to (8)), we *can* lose 'abstract forward nondeterminism' by simulating the protocol forward.

We also have the following.

Corollary 3.10. Under the assumptions of Theorem 3.8, suppose there are no BS steps (i.e. all SA steps are OD). Then single step forward simulations can be used throughout.

Obviously, choosing the SA as the last step of each maximal path through the protocol satisfies the hypotheses of Corollary 3.10.

Corollary 3.11. Let $MPath(v_I, \ldots, v_F)$ be a maximal path from an initial v_I to a final v_F, such that (10) holds (for suitably chosen other quantities). Let $St_\rho(v_{t-1}, j_t, p_t, v_t)$ be the SA(v_I) step along $MPath(v_I, \ldots, v_F)$. Then the simulation of $MPath(v_I, \ldots, v_F)$ by $Atomic(u_I, i, o, u_F)$ can be decomposed as follows:

1. If $St_\rho(v_{t-1}, j_t, p_t, v_t)$ is an ODFS step, the simulation of $MPath(v_I, \ldots, v_F)$ may be established by inductively forward simulating the steps of $MPath(v_I, \ldots, v_F)$ from v_I up to a state $v_{\tilde{t}}$ (which does not precede v_t), and backward simulating the steps of $MPath(v_I, \ldots, v_F)$ from v_F up to $v_{\tilde{t}}$ (if $v_{\tilde{t}} \neq v_F$), such that:
   (a) predecessors of $St_\rho(v_{t-1}, j_t, p_t, v_t)$ are forward simulated by the identity operation on u_I,
   (b) $St_\rho(v_{t-1}, j_t, p_t, v_t)$ is forward simulated by $Atomic(u_I, i, o, u_F)$ where u_F is the unique element of Outcomes(St_ρ, u_I), establishing $R^F(u_F, v_t)$,
   (c) FS successors of $St_\rho(v_{t-1}, j_t, p_t, v_t)$ are forward simulated from v_t by the identity operation on u_F, establishing $R^F(u_F, v_{\tilde{t}})$,
   (d) BS successors of $St_\rho(v_{t-1}, j_t, p_t, v_t)$ are backward simulated from v_F by the identity operation on u_F, establishing $R^F(u_F, v_{\tilde{t}})$.
2. If $St_\rho(v_{t-1}, j_t, p_t, v_t)$ is an ONFS step, the simulation of $MPath(v_I, \ldots, v_F)$ may be established by inductively forward simulating the steps of $MPath(v_I, \ldots, v_F)$ from v_I up to and including $St_\rho(v_{t-1}, j_t, p_t, v_t)$, and inductively backward simulating the steps of $MPath(v_I, \ldots, v_F)$ from v_F up to v_t, such that:
   (a) predecessors of $St_\rho(v_{t-1}, j_t, p_t, v_t)$ are forward simulated by the identity operation on u_I,

(b) $St_\rho(v_{t-1}, j_t, p_t, v_t)$ is forward simulated by $Atomic(u_I, i, o, u_F)$, for each u_F in Outcomes(St_ρ, u_I), establishing $R^F(u_F, v_t)$,

(c) successors of $St_\rho(v_{t-1}, j_t, p_t, v_t)$ are backward simulated from v_F by the identity operation on u_F, establishing $R^F(u_F, v_t)$.

Why are the above results useful? We can give a couple of reasons.

Firstly, they are illuminative. One can be convinced of the correctness of a protocol with respect to an atomic action, without having the details of a refinement already worked out. In such a situation, it may not be clear how to synchronise the atomic action with the lower level description. Theorem 3.8 shows that one can choose this synchronisation relatively freely, within the parameters of allowable synchronisation assignments.

Secondly, once having chosen a synchronisation, it is much easier to write down the 'big-step' retrieve relation and associated input and output relations, than to discover the more finegrained single step ones. Theorem 3.8 shows that with the big-step retrieve relation fixed, the single step ones, R^P and R^F may simply be *calculated*. Their generic form needs to be instantiated with the details of the protocol and big-step retrieve relation, and then one must eliminate as many existential quantifiers as possible in order to arrive at a closed form. Making clear that there *is* such a strategy to follow is a considerable improvement over the hit-and-miss approach one would otherwise need, especially when combined with uncertainty regarding synchronisation.

The theorem and its corollaries also provoke the following considerations.

One can replace some backward simulation by forward simulation. Given a synchronisation assignment, a branching BS step is a BS step $St_\theta(v_s, \ldots, v'_{s,1})$ for which there is another BS step $St_\phi(v_s, \ldots, v'_{s,2})$ (with $v'_{s,1} \neq v'_{s,2}$) such that the abstract outcomes $u_{F,1}, u_{F,2}$ corresponding to the completions of the paths from $v'_{s,1}$ and $v'_{s,2}$ are different, $u_{F,1} \neq u_{F,2}$.[4] In such a case, one *cannot* make a forward simulation inference succeed.

To see this, suppose the first hypothesis of (23) is made true by $R^1(u_{F,1}, v_s)$, and the second hypothesis is made true by $St_\phi(v_s, \ldots, v'_{s,2})$. Then the first hypothesis demands that u_F be chosen to be $u_{F,1}$, while the second hypothesis demands that u_F be chosen to be $u_{F,2}$, a contradiction. This is the standard backward simulation counterexample.

In Fig. 3, the SA element along the upper thread of the computation tree is an ONFS step, since it can reach two concrete final states that retrieve to two different abstract outcomes. Accordingly, the two BS steps immediately following it (and the two following the topmost of them along the upper thread) are branching BS steps, since they too can individually reach different concrete final states that retrieve to the two different abstract outcomes. With the dotted lines depicting R^F, it is easy to see that these steps illustrate what we have just discussed.

However, if a BS step is *not* branching, i.e. there is only one protocol successor state v'_s to v_s, then the preceding problem cannot arise since the unique successor cannot force a distinction between the choices for u_F. So for non-branching BS steps, a forward simulation inference will succeed. However, it comes at a price. If a forward simulating BS step immediately follows a backward simulating BS step, the $R^1(u_F, v)$ value at the v state that they share, occurs as a hypothesis in both the backward PO (24) and the forward PO (23). It thus remains as an unproved assumption in the overall single-step verification of the big-step refinement. As such it allows the verification to succeed vacuously. For this reason we phrased Corollary 3.11.2 as two inductive processes that meet in the middle, since it is much better to verify some $R^1(u_F, v)$ twice independently, than to leave some other $R^1(u_F, v)$ unproved, thus undermining the whole verification.

Lastly, Theorem 3.8 offers a different strategy for addressing global correctness (see Section 5). Normally, to prove a protocol such as the one we have been considering globally correct, one chooses either forward or backward simulation, establishes that each protocol step refines some atomic option or skip, and this then extends to an inductive proof for global executions as a whole. With Theorem 3.8, we can envisage a different approach, structured as follows.

1. We first study the 'big-step' refinement of atomic action to protocol, determining the protocol computation DAG and the big-step retrieve relation.
2. Next we choose a suitable synchronisation assignment.
3. Next we determine which combination of forward and backward simulations are appropriate for the synchronisation assignment.
4. Next we calculate the necessary single step retrieve relation, breaking down the big-step refinement into single step refinements.

[4] Since we speak of a BS step, there must be such $u_{F,1} \neq u_{F,2}$, as the nondeterminism in $Atomic(u_I, i, o, u_F)$ has been resolved earlier than at this BS step.

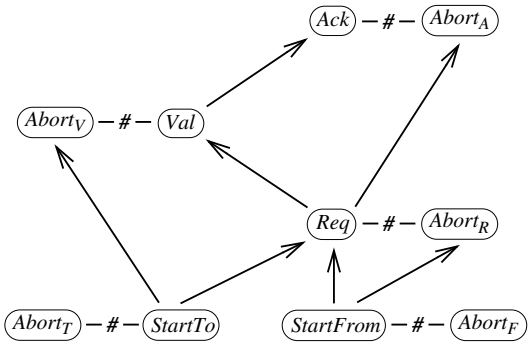

Fig. 4. An event structure for the Mondex protocol.

5. Finally, we determine how runs of the protocol can interleave to make global executions.

This alternative approach separates concerns, and in cases where a complex protocol is concerned, may offer some advantages. In any event, the mere awareness of the possibility of such an approach may make the more monlithic standard approach more tractable, since it can show that certain subgoals of the standard approach are achievable in advance.

4. Event Structures and Protocol Computation Trees

Step 1 of the alternative verification strategy just suggested relies on determining the protocol computation DAG. Usually, consideration of this computation structure is not itself the means by which a protocol is invented, so the computation DAG might well be derived from alternative starting points.

A common way of inventing a protocol is to say 'this happens after that' for a sufficiently large number of cases. Such a train of thought can be formalised quite effectively using event structures of various kinds [WN95, Bou90, NPW81, Win86, Win88, BC88, PP95]. Accordingly, we use event structures with symmetric conflict relations to encode possible playouts of a protocol, and show how to derive a computation tree from an underlying event structure of this kind. Once there, one can map the tree to a more convenient DAG if one wishes.

Definition 4.1. An (symmetric flow) event structure $\mathcal{E}$ is a triple $(E, \prec, \#)$ such that:

1. E is a set (of events).
2. $\prec$ is an asymmetric causal flow relation on E (whose transitive (resp. reflexive transitive) closure is written $<$ (resp. $\leq$)).
3. $\#$ is an irreflexive symmetric conflict relation on E compatible with $\leq$, i.e. such that $x \# y \leq z \Rightarrow x \# z$.

The preceding is a very simple definition which will do for our immediate purposes. Generalisations arise by eg. allowing the conflict relation to be asymmetric; see some of the cited literature. Since we need asymmetric conflict later (see Section 7), we formulate the semantics of symmetric flow event structures in nonsymmetric terms right away.

Definition 4.2. Let $\mathcal{E} = (E, \prec, \#)$ be a symmetric flow event structure. Let the associated nonsymmetric conflict relation $\#^a$ be the smallest relation on E closed under:

1. $x \# y \ \Rightarrow x \#^a y \wedge y \#^a x$.
2. $x \#^a y \wedge y \leq z \Rightarrow x \#^a z$.

The point of Definition 4.2 is that $\#^a$ is not deemed to be symmetric *a priori*, allowing us to introduce more elaborate notions later in terms of enhancements of $\#^a$ that are not symmetric.

An event structure defines which events may occur once other events have already occurred. Collections of events are called configurations, and the legal configurations and legal ways of passing from one configuration to a successor configuration are packaged up in the following definition.

Definition 4.3. Let $\mathcal{E} = (E, \prec, \#)$ be an event structure with associated nonsymmetric conflict relation $\#^a$. The set

$\mathcal{X}_\mathcal{E} \subseteq \mathbb{P}E$ of (legal) configurations of $\mathcal{E}$, and the legal ways of moving from a legal configuration X of $\mathcal{E}$ to a successor legal configuration Y are given by the following rules.

1. $\varnothing \in \mathcal{X}_\mathcal{E}$.
2. $X \in \mathcal{X}_\mathcal{E}$,
   $x \in E - X$,
   $(\forall x' \in E \bullet x' \prec x \Rightarrow x' \in X)$,
   $(\forall x' \in E \bullet x' \#^a x \Rightarrow x' \notin X)$,
   $\vdash \quad X \cup \{x\} \in \mathcal{X}_\mathcal{E}$.

In Fig. 4 we show an event structure for the Mondex protocol, adapted from the activity diagram of Fig. 2 to include all the 'abnormal' ways theat the protocol can play out, and flowing up the page. The constituent events are in the labelled nodes, while the arrows show the elements of the flow relation $\prec$, and the #-labelled edges show a generating set for the conflict relation. In the Mondex documentation [SCW00] the various $Abort_x$ events are all part of a single *Abort* operation, which has been split into five pieces in Fig. 4 according to which 'normal' event the *Abort* is in conflict with.

In Fig. 4 there are two root events, *StartFrom* and *StartTo*, either of which can start an 'execution' of the event structure. (For the time being, we ignore the possibility of starting with one or both of the $Abort_T$ or $Abort_F$ events, which lead to 'stillborn' executions; they are included in Fig. 4 for later convenience.) Once the first event has taken place, we have a (different) choice of two next events (depending on which *Start* event went first). If the next event is the other *Start* event, then we have a choice of three subsequent events ... and so on. Working out all the possible orderings of events yields a quite complex structure, and it is clear that the event structure formalism captures all these possibilities in a compact and convenient way.

In general, an event structure is executed by starting with the empty configuration, and then one executes one event at a time, adding a new event x to the existing configuration X, as sanctioned by the rules in Definition 4.3. So Definition 4.3 provides a proof system that enables us to derive sequences of event occurrences. The set of sequences obtained thereby can be turned into DAG-shaped and forest-shaped transition relations as follows.

Definition 4.4. Let $\mathcal{E} = (E, \prec, \#)$ be an event structure. The transition system $\mathcal{E}^{\mathsf{DAG}}$ associated with $\mathcal{E}$ is defined by:

1. the states are the configurations $X \in \mathcal{X}_\mathcal{E}$, with $\varnothing$ as intial state,
2. the transitions are the steps $X \xrightarrow{x} X'$ iff $X \in \mathcal{X}_\mathcal{E} \ldots \vdash X' = X \cup \{x\} \in \mathcal{X}_\mathcal{E}$ is a valid inference according to Definition 4.3.

Evidently $\mathcal{E}^{\mathsf{DAG}}$ is a DAG.

Definition 4.5. Let $\mathcal{E} = (E, \prec, \#)$ be an event structure and $\mathcal{E}^{\mathsf{DAG}}$ its associated transition system. The transition forest $\mathcal{E}^{\mathsf{FOR}}$ associated with $\mathcal{E}$ is defined by:

1. the states are the paths $\langle \varnothing, \ldots, X \rangle$ in $\mathcal{E}^{\mathsf{DAG}}$ which start at the initial $\mathcal{E}^{\mathsf{DAG}}$ state, with the empty path as initial $\mathcal{E}^{\mathsf{FOR}}$ state,
2. the transitions are the steps $\langle \varnothing, \ldots, X \rangle \xrightarrow{x} \langle \varnothing, \ldots, X, X' \rangle$ iff $X \xrightarrow{x} X'$ is a step of $\mathcal{E}^{\mathsf{DAG}}$.

The preceding gets us some way towards the provisions of Section 3. However we are not there yet. Section 3 is couched in relational terms. So events have to correspond to relations, and the enabledness or otherwise of these relations in any state must correspond to what the flow and conflict relations of the event structure permit in given configurations. In general, the process will be application specific, since it will depend on many factors, such as how many protagonists participate in the protocol, what their local state is envisaged to be, what knowledge of the global state they have, the role of I/O, etc. However, in the context of designing a protocol to accomplish some identified atomic goal, the process of reconciling these two approaches can provide a useful consistency/correctness check on the design activity.

Beyond that, our event structure account of Mondex left out certain state components, such as the details of purse balances and amounts transfered etc., that a full account must include — i.e. the event structure was deliberately intended to be generic. Reinstating the omitted components generates a replication of the forest indexed by the reinstated values, corresponding to the full computation forest.[5]

[5] N. B. This picture incidentally yields one useful convention for the ρ labels of the step relations St_ρ of Section 3: namely to tag each edge of the 'generic' forest by a distinct label, and then to retain these labels in each replicated forest, making the labels akin to names of 'operations' at what would be the code level.

Once the event description is in place, and one is confident that it properly corresponds to the relational picture, we can extract a computation forest via the constructions of Definitions 4.4 and 4.5.

By construction, the nodes of our forest shaped computational DAG incorporate the full history of the protocol up to the given point. Such history information is often needed in reasoning about protocols, since protocol properties frequently depend not only on knowing that the protocol has arrived at a certain point, but that certain other things must have necessarily happened prior to that point. Such facts can be trivially extracted from the full history, so our formulation may be regarded as a multipurpose canonical description, useful for things other than just the concerns of this paper. However, since different paths can often arrive at 'essentially the same' state eg. via interchanges of causally independent steps somewhere in the interior of the protocol, it is just as useful to be able to forget aspects of history, and eg. identify common suffixes of certain paths. The duality between $\mathcal{E}^{\mathsf{DAG}}$ and $\mathcal{E}^{\mathsf{FOR}}$ (given in one direction by the construction of $\mathcal{E}^{\mathsf{FOR}}$ from $\mathcal{E}^{\mathsf{DAG}}$, and in the other by forgetting all but the last component of each state in $\mathcal{E}^{\mathsf{FOR}}$) bears out the compatibility of these different points of view.

Another aspect that should be discussed is I/O. At the atomic level, the I/O for the single step that takes place must inevitably concern the environment, since there is no internal structure to engage in internal communication. At the protocol level however, I/O can either be between the environment and the protocol, or be purely internal to the protocol. In the latter case, the only restriction is that messages must be produced before they can be consumed. There is of course the option of representing messages in flight within a suitable state component —such a state component can model properties of the communication medium, eg. unreliablity— however we do not need to insist on that for the serialisation discussed in the next section.

5. Interleaving Individual Protocol Runs

Thus far, although using language such as 'protocol,' in reality we have only discussed some properties of acyclic transition systems. In genuine protocols, various agents interact by performing events and sending/receiving messages etc. We must connect our theory to this world.

The basic idea is that the previous section should be understood as describing (the various possibilities for) a single isolated protocol run, performed by however many agents would be appropriate in practice, with the protocol state recording in principle the full history of the protocol so far (regardless of whether such knowledge can indeed be posessed by the individual agents), and ignoring the rest of the universe. The latter not only regarding other agents/activities in the rest of the universe, but also regarding what the agents of the single protocol run might do both before and after the run itself. So the previous section described an idealised *pattern* or *template* for what collections of agents might do over some period of time towards the achievement of some goal described by the atomic action that the protocol implements.

Patterns or templates are normally made to correspond with what happens in the real world by some process of matching, and that is the basis of our approach too. Since we have remarked that our protocol states can in principle include unrealistically detailed history information, our matching process must include a projection mechanism to allow the unrealistic parts to be forgotten. In such a scenario, protocol states that were previously distinct can be matched to the same system state, just as we described in the previous section.

Definition 5.1. A *system* consists of a number of *agents*, $A_a, A_b, \ldots$ each with its agent state subspace $W_a, W_b, \ldots$. The system state space is $W = W_a \times W_b \times \ldots$. So agent A_a's instantaneous state is some $w_a \in W_a$, and the system's instantaneous state is $w \equiv (w_a, w_b, \ldots)$.

Each agent is a transition system, i.e. the agent can move between different elements of its state space in discrete steps, leaving the state of every other agent unaffected. The enabledness of any agent's transitions is independent of the state of any other agent. Each step can also consume input and produce output, and the I/O policy described in the previous section applies again: i.e. I/O may either be with the environment, or it may be internal to the system, and any internal message that is consumed must earlier have been produced.

The system's transitions are described by a predicate Sy_A similar to *St* in the Section 3, where the subscript '$_A$' refers to the agent performing the step, and each Sy_A step modifies only its own agent's state subspace. The transitions of the system as a whole are the interleaved agent transitions of the system's agents, each extended with skip on the irrelevant part of the total system state. The skip-extended transitions are written $\overline{Sy}_A$.

Definition 5.2. Let S be a system with agents $A_a, A_b, \ldots$. The sequence $\mathcal{T} \equiv \langle w_I, (k_1, A_1, q_1), w_1, (k_2, A_2, q_2), w_2, \ldots \rangle$ is a run of the system iff:

1. w_I is an initial state of the system,

2. A_1 is the agent that performs the first step,
3. k_1 is the input consumed by A_1 during the first step,
4. q_1 is the output produced by A_1 during the first step,
5. w_1 is the result state of the first step,
6. the change of state $w_I \rightarrow w_1$ involves change to the state space W_1 of A_1 only; the state spaces of agents other than A_1 remain unchanged,
7. ... and analogously for subsequent system transitions.

Definition 5.3. Let *Protocol* be a protocol in the sense of the previous section. An agent decomposition for the protocol is a decomposition of the protocol state space V into a cartesian product of agent subspaces $V = V_1 \times V_2 \times \ldots$, such that each step of the protocol modifies[6] at most one component in the product, leaving the other components unaltered.

The decomposition into agent subspaces just described, represents the fact that an instantiation of a protocol is normally executed by a number of agents inside a real system. However a real agent in a real system can play many roles during the running of the system, including acting out different roles in different instances of the same protocol at different times. So we need to distinguish the various agent roles in a protocol definition from the different instantiations of these during system runs. The next definition introduces the technical machinery for this.

Definition 5.4. Let $Atomic, Protocol, \ldots$ (with all the attendant machinery) be a protocol implementing an atomic action in the sense of the previous section. We say that system run $\mathcal{T}$ instantiates *Protocol* iff there is a maximal path through the protocol $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I, j_1, p_1, v_1, j_2, p_2, v_2, \ldots, v_{F-1}, j_F, p_F, v_F)$ and there are two maps: τ_A and τ_S such that:

1. there is a cartesian product of disjoint functions $\tau_{A,l} : V_l \rightarrow W_{a_l}$ from all of the agent components of V to a (possibly proper) subset of distinct agent subspaces of W, and $\tau_A = \Pi_l \, \tau_{A,l}$,
2. τ_S is an injective function from the steps of the maximal path $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ to steps of $\mathcal{T}$,
3. τ_S is order preserving, i.e. if St_β precedes St_γ in $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$, then $\tau_S(St_\beta)$ precedes $\tau_S(St_\gamma)$ in $\mathcal{T}$,
4. for each protocol step $St_\beta(v_{t-1}, j_t, p_t, v_t)$ in the domain of τ_S, if V_l is the agent component of V modified during the step, then $\tau_{A,l}(V_l)$ is the agent subspace modified during the step $\tau_S(St_\beta(v_{t-1}, j_t, p_t, v_t))$,
5. for each protocol step $St_\beta(v_{t-1}, j_t, p_t, v_t)$ in the domain of τ_S, if $\tau_S(St_\beta(v_{t-1}, j_t, p_t, v_t)) = Sy_{A_l}(w_{s-1}, k_s, q_s, w_s)$, then $\tau_{A,l}(v_{t-1}) = w_{s-1}$, $j_t = k_s$, $p_t = q_s$, $\tau_{A,l}(v_t) = w_s$,
6. if protocol step St_β modifies V_l and protocol step St_γ is the next protocol step along $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ that modifies V_l, then no step of $\mathcal{T}$ between $\tau_S(St_\beta)$ and $\tau_S(St_\gamma)$ modifies $\tau_{A,l}(V_l)$.

When we want to emphasise the details, we say that system run $\mathcal{T}$ instantiates *Protocol* via $\tau \equiv (\tau_A, \tau_S)$ at step $\tau_S(St_\alpha(v_I, j_1, p_1, v_1))$ of $\mathcal{T}$, where $St_\alpha(v_I, j_1, p_1, v_1)$ is the initial step in $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$.

In Fig. 5 we show how a particular maximal path, M say, through the protocol illustrated in Fig. 3, might be mapped, via an instantiation function τ, to a selection of steps in a system run. The system state in the run is now 'real world' state, eschewing the maximal knowledge that the idealised protocol formulation allows. In between the steps of $\tau(M)$, other protocols are being instantiated by other agents, though without interfering with the state of $\tau(M)$, by Definition 5.4.6.

Definition 5.5. Let $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ be a maximal path in *Protocol*. Step $St_\beta(v_{t-1}, j_t, p_t, v_t)$ of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ is a first use of agent subspace V_l iff: it modifies V_l, and no earlier step of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ modifies V_l. Similarly $St_\beta(v_{t-1}, j_t, p_t, v_t)$ is a last use of V_l iff: it modifies V_l, and no later step of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ modifies V_l. We say that *Protocol* is 2-phase (2P) along $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ iff all first uses of all agent subspaces of *Protocol* precede any last use of any agent subspace of *Protocol* along $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$.

Definition 5.6. Let $\overline{Sy}_A(w_{s-1}, k_s, q_s, w_s)$ and $\overline{Sy}_B(w_s, k_{s+1}, q_{s+1}, w_{s+1})$ be two successive steps of a run $\mathcal{T}$ of the system. We say that $\overline{Sy}_A(\ldots)$ and $\overline{Sy}_B(\ldots)$ can be commuted iff there is a state $\tilde{w}_s$ such that $\overline{Sy}_A(\tilde{w}_s, k_s, q_s, w_{s+1})$ and $\overline{Sy}_B(w_{s-1}, k_{s+1}, q_{s+1}, \tilde{w}_s)$ are valid steps of the system, and the pair $\overline{Sy}_A(w_{s-1}, k_s, q_s, w_s)\,;\,\overline{Sy}_B(w_s, k_{s+1}, q_{s+1}, w_{s+1})$ can be replaced in $\mathcal{T}$ by $\overline{Sy}_B(w_{s-1}, k_{s+1}, q_{s+1}, \tilde{w}_s)\,;\,\overline{Sy}_A(\tilde{w}_s, k_s, q_s, w_{s+1})$, yielding $\mathcal{T}'$, where $\mathcal{T}'$ is a valid run.

[6] Here, and in the remainder of the paper, 'modifies' should be understood to mean 'is deemed to modify' or, 'is permitted to modify in the syntactic description of the step,' since it is intended to cover not only non-trivial update, but also cases of read-only access, and cases in which the agent in fact chooses not to access the state at all (even though the syntactic description, of which the step is a specific instantiation, permits it).

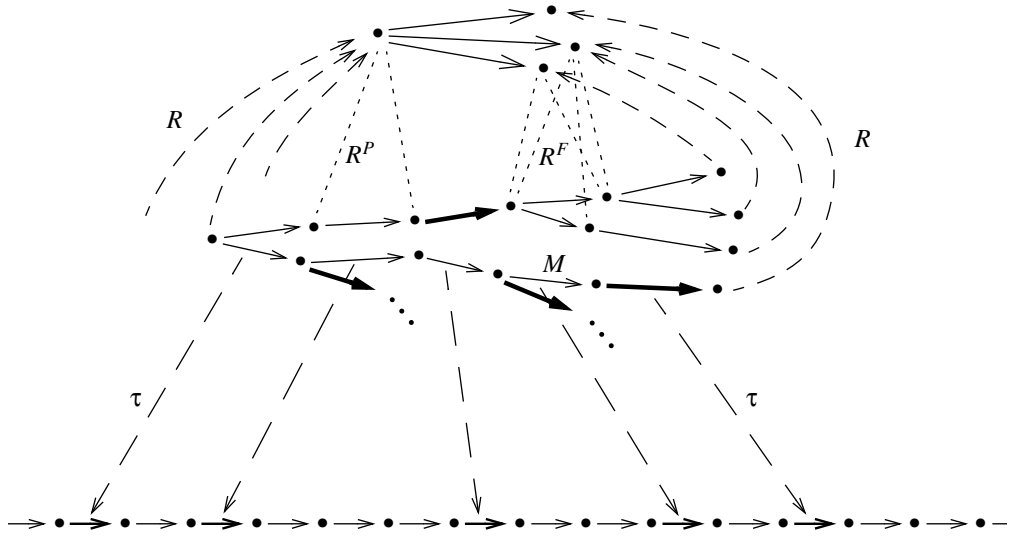

Fig. 5. An atomic action, a protocol which implements it, and a system run containing an instance of a maximal path through the protocol. The steps of the instance are shown bold.

Lemma 5.7. If $\overline{Sy}_A(\ldots)$ and $\overline{Sy}_B(\ldots)$ as in Definition 5.6, are two successive steps performed by two different agents, then, provided both inputs are available in state w_{s-1}, $\overline{Sy}_A(\ldots)$ and $\overline{Sy}_B(\ldots)$ can be commuted.

Proof. Since $\overline{Sy}_A(\ldots)$ and $\overline{Sy}_B(\ldots)$ are performed by different agents, the two agent subspaces modified by these steps are disjoint, so the changes of state can be swapped, easily yielding the state $\tilde{w}_s$ required by Definition 5.6. If both inputs are available in state w_{s-1}, then the $\overline{Sy}_B(\ldots)$ is enabled in state w_{s-1} and can be performed first. Since the input to $\overline{Sy}_A(\ldots)$ is not removed by doing $\overline{Sy}_B(\ldots)$, $\overline{Sy}_A(\ldots)$ can follow $\overline{Sy}_B(\ldots)$. That this generates a valid run is now straightforward. □

Since our formulation of a protocol does not consider the protocol's context, the only way that a protocol, as formulated in Section 3, can interact with the rest of the universe is via I/O with the environment. In the system context, this leads to a distinction within the internal system messages, between messages that are produced and consumed by the same protocol instance (which should thus correspond to internal communications of the protocol itself), and those which are produced and consumed by different protocol instances (which should thus correspond to communications with the environment in the protocol model). (System level communications with the environment must of course also correspond with protocol communications with the environment.) Since inter-protocol communications must comply with normal causality considerations, these communications must fit well with the 2-phase property for protocol state components. The next definition introduces the needed technicalities.

Definition 5.8. Suppose given a maximal path $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ of a protocol, which is 2P. An external dependency definition (XDD) for it is a pair of (not necessarily disjoint) sets (IDS, ODS) of steps of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$. *IDS* is the input dependency set: the set of steps of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ during which an external input (i.e. one originating from outside $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$) is received; and *ODS* is the output dependency set: the set of steps of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ during which an external output (i.e. one destined to outside $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$) is delivered. A protocol is 2PXDD-normal iff:

1. all *IDS* steps occur no later than any *ODS* step along $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$,
2. the producer of every input of every protocol step other than an *IDS* step is some other step of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$,
3. the consumer of every output of every protocol step other than an *ODS* step is some other step of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$,
4. each *IDS* step occurs no later than any last use of the state,
5. each *ODS* step occurs no earlier than any first use of the state.

Definition 5.9. An instantiation of a 2PXDD-normal protocol is called a (2PXDD-normal) transaction.

For the rest of this section all transactions will be 2PXDD-normal.

Theorem 5.10. Let $\mathcal{T}_0$ be a run of a system which consists entirely of the steps of transactions of a family of protocols.[7] Then there is a serialisation $\mathcal{T}_\infty$ of $\mathcal{T}_0$, generated by commuting adjacent steps, in which each instantiation occurs as a contiguous series of steps.

Proof. Consider the directed graph Dep_0 whose nodes are the transactions of $\mathcal{T}_0$, and whose edges are given by: $\tau_1 \to \tau_2$ iff:

1. an output of an *ODS* step of τ_1 is an input of an *IDS* step of τ_2, or,
2. an agent subspace V_l is used by both τ_1 and τ_2, but τ_1's modifications of V_l occur earlier in $\mathcal{T}_0$ than τ_2's.

Claim 5.10.1 Dep_0 is acyclic.

Proof of Claim. Let V be the state space of a transaction τ. Since the last first use of V precedes the first last use of V in τ, and all all *IDS* steps precede all *ODS* steps in τ, by Definition 5.8.4-5, we can deduce that there is a step in τ (which we will call the pivot), that precedes neither the last first use of V nor any *IDS* step, and simultaneously follows neither the first last use of V nor any *ODS* step (there are four cases). We identify each transaction in $\mathcal{T}_0$ with (some choice for) its pivot. Since steps are interleaved, there is a total order on the transactions, inherited from that on their pivots.

We show that Dep_0 can be interpreted in the set of pivots, and that each edge in the interpretation is oriented towards the future, yielding the acyclicity of Dep_0 immediately. For a Dep_0 edge of type 1, note that it is oriented towards the future by straightforwards causality. So pretending that the requisite message was sent during the producing transaction's pivot step, and pretending that it arrived during the consuming transaction's pivot step can increase its time of flight, but not change its orientation towards the future. For a Dep_0 edge of type 2, since the pivot steps are contained within the uses of transactions' state, and these are oriented towards the future by 2, the orientation is preserved in the interpretation. We have our claim. □□

We serialise $\mathcal{T}_0$ stage by stage. At each stage there are serialised and unserialised transactions. We call the boundary between the serialised and unserialised transactions the horizon. So at the beginning there are no serialised transactions, and the horizon lies just before the first step of $\mathcal{T}_0$. At the n'th stage, which starts with $\mathcal{T}_n$, whose unserialised transactions comprise Dep_n (a subgraph of Dep_0), we choose an unserialised transaction which is a root of Dep_n, and we serialise it, whereupon its steps —in contiguous sequence— are both appended to the serialised part, and removed from the unserialised part of the partly serialised run, moving the horizon to just beyond the newly serialised steps, and yielding $\mathcal{T}_{n+1}$ and Dep_{n+1}. If $\mathcal{T}_0$ is infinite, then the serialisation process continues forever, and every finite prefix of $\mathcal{T}_0$ has all its steps eventually included in the serialised part. If $\mathcal{T}_0$ is finite, the process stops when the last transaction of $\mathcal{T}_0$ has been serialised.

Stage n: A root transaction τ_n of Dep_n is chosen. By assumption, all transactions on which τ_n is dependent, whether through the state space, or via τ_n's *IDS* messages, have been serialised, i.e. their steps lie beyond the horizon. So any step of $\mathcal{T}_n$ that lies between the horizon and τ_n's first step neither uses any state used by τ_n's first step, nor produces a message consumed by τ_n's first step. So there is no obstacle to commuting the first step of τ_n towards the past until it it arrives immediately after the horizon. Similarly the dependencies for the second step lie either beyond the horizon, or arise from the first step, so the second step of τ_n can be commuted towards the past until it arrives immediately after the first. The process continues until the last step of τ_n has been commuted until it arrives immediately after its predecessor. This yields $\mathcal{T}_{n+1}$. Transaction τ_n is removed from Dep_n, yielding Dep_{n+1}, and the horizon is moved to just after τ_n's last step. *End Stage n*. □

The preceding amounts to a sketch of a relatively standard 2-phase serialisation proof process [BHG87, GR93, BN97, WV02]. And once the run has been serialised, it is clear that each transaction of the serialised run is a refinement of its corresponding atomic action via a retrieve function that forgets the part of the system state not relevant to the transaction.

6. Mondex and its Refinements

In this section we reflect on the Mondex protocol, and the extent to which its refinement possibilities correspond to the preceding theory. There are a number of points to be borne in mind.

[7] So there is a set of maximal paths through a set of 2PXDD-normal protocols, and a set of instantiations of them in $\mathcal{T}_0$, and the set of steps of $\mathcal{T}_0$ is the disjoint union of these instantiations.

First of all, our theory has been couched in terms of single transitions (which is less cluttered), whereas Mondex is couched in terms of Z *operations* [Spi92, DB01, ISO02]. The distinction is the same as the one discussed in Section 4 between the generic event structure and its replication in the detailed computational structure by all the permitted values of the generically omitted state. Therefore when we say below that such and such an operation is synchronised with such and such an atomic action, we are referring in bulk to all the transitions of the operation being suitably synchronised with appropriate instantiations of the atomic action.

Secondly, we will restrict our attention for now to runs of the protocol which commence with the two *Start* operations, *StartFrom* and *StartTo*, in either order, (returning to other possibilities at the end of this section). Referring to Fig. 2, this means that after the two *Start* operations, the protocol, which is henceforth serial (as is obvious from the causal dependencies of the *req*, *val* and *ack* messages), executes some prefix of the *Req-Val-Ack* sequence of operations. If it does not complete that sequence, each purse that still has elements of the *Req-Val-Ack* sequence left to do, performs an *Abort* operation (which replaces the first such unperformed *Req-Val-Ack* operation left on that purse's agenda), completing the protocol abnormally. Note however that unlike the *Req-Val-Ack* operations which are causally constrained by the *req*, *val*, *ack* messages, *Abort* operations are not causally constrained and can occur at any time. Every variation in the order of performing the protocol's operations when *Abort* events are involved, causes a branching of the computation tree structure, and leads overall, to quite a complex protocol computation tree. All of this concurs with the possibilities offered in the event structure of Fig. 4.

6.1. The Original Mondex Refinement [SCW00]

In [SCW00], the refinement is constructed to synchronise with the atomic description as early as possible, given the assumptions above. Thus the atomic action is synchronised with the *Req* operation, which refines both *AbTransferOK* and *AbTransferLost*. Since the protocol still has plenty of opportunity to fail after the *Req* operation, the *Req* operation itself does not fix the outcome, so the refinement, achieved on the basis of a global inductive proof, has to be a backward one. We can visualise to some extent the substructure of Fig. 3 that forces a backward simulation (referred to at the end of Section 3), from Fig. 2, if we add an edge from *Req* to an *Abort*, as an alternative to the message towards *Val*, since the two abstract outcomes are already available at the end of the *Req* operation. Furthermore, since for a failing transaction the protocol has already angelically chosen to refine *AbTransferLost*, the *Abort* operation(s), which actually signal the failure at the protocol level, all refine *AbIgnore* (which is Mondex-speak for an abstract skip).

6.2. The Refinement of Banach et al. [BPJS07]

In [BPJS07], amongst other things, a synchronisation with the atomic description that occured late was sought, in order to try to get a forward simulation.[8] The natural operation to refine *AbTransferOK* to is *Val*, since that is the moment that the money safely arrives at the recipient. However, if the refinement of *AbTransferOK* is 'obvious,' then the refinement of *AbTransferLost* is less so. The subtlety lies within the *Abort* operation. The deeper structure of the Mondex protocol implies that if only one *Abort* occurs in a transaction, it is harmless, and such an *Abort* can refine *AbIgnore*. Only if two *Abort* operations occur for a transaction, each while its respective purse is in a critical state, has the transaction failed non-trivially, whereupon the transaction needs to refine *AbTransferLost*. This leads to the decomposition of the *Abort* operation into cases, depending on the precise role of the operation in the transaction. In the formalism of this paper, the *Abort* operation of Mondex corresponds to a collection of events which occur at different places in the computation tree of the protocol, and are thus distinguishable.

The case analysis is interesting. The distinction between benign and non-benign instances of *Abort* is made on the basis of a purse's local state (specifically, on whether the purse is in state *epv* or *epa* (non-benign), or in some other state (benign)). However, since two *Abort*s make one *AbTransferLost*, we can only refine *AbTransferLost* to one of the pair — and it has to be the second of the pair, since if only one *Abort* in a critical state happens, then it turns out to be benign nonetheless. In [BPJS07] *non-local* state information is used to distinguish the first non-benign *Abort* from the second, and the first is then made to refine *AbIgnore* while the second refines *AbTransferLost*.

[8] Looking forward to some extent to the specific results of the present paper —which show that the essentials of a protocol can be understood by discussing the protagonists in isolation— the discussion in [BPJS07] was restricted to a world of just two purses, a single From purse and a single To purse.

6.3. The Refinement of Schellhorn et al. [SGH$^+$07]

In [SGH$^+$07] we have the second mechanized verification of Mondex using the the KIV theorem prover [RSSB98]. While the first [SGHR06b] used the original backward simulation and data refinement, the second uses abstract state machines (ASMs, [Gur95], [BS03]) together with ASM refinement and generalized forward simulations [Sch01].

The refinement, like [BPJS07], synchronizes successful transfers by having *Val* implement *AbTransferOK*. But it chooses to synchronize failed transfers at the earliest point possible. This gives two cases for the *Req* operation, which is the point where the From purse sends money. In the first, the To purse is still ready to receive the money, in which case *Req* implements *AbIgnore*. But if the To purse has already aborted then the second case applies, and *Req* implements *AbTransferLost*.[9] Instead of having two cases (as in [BPJS07]) in which the *Abort* operation implements *AbTransferLost*, the design of [SGH$^+$07] leaves only one: the case where the To purse aborts in *epv* after money has been sent.

The different choices for the synchronisation points was one motivation for us to study the general possibilities here. Another one was to provide a general formalization of using past and future simulation relations (R^P and R^F). Instances of such relations with a schematic encoding into Dynamic Logic are not only used in the case study [SGH$^+$07] but also in earlier work. Future simulation relations occur in the correctness proof of ASM refinement [Sch01]. Past simulation relations are used in coupled refinement [DW03] as noted in [Sch05].

6.4. The Refinements of Haxthausen, George et al. [HGS06]

The two refinements of [HGS06] use the RAISE specification language [The92]. They are another mechanized verification of Mondex using the theorem prover PVS [ORS92]. This case study is slightly out of scope of our theory, since it does not start with atomic actions, but with a two step protocol: the first step (called *TransferLeft*) is a send operation, which nondeterministically chooses between a success and failure, and we call the two cases *SendOK* and *SendFail*. After *SendOK*, there are again two possibilities: receiving may succeed or fail. For symmetry, we call these operations *ReceiveOK* and *ReceiveFail*, [HGS06] calls them *TransferRight* and *Abort*. Already, the splitting of transactions at the abstract level into send and receive, allows us to keep the balances of abstract and concrete level in perfect synchrony, as is required by RAISE refinement. The two refinements implement *TransferLeft* with *Req* and *ReceiveOK* with *Val*.

To compare the synchronisation points with our proofs, we have to add an additional refinement of the original abstract Mondex level to the abstract RAISE level. The refinement would have to implement *AbTransferOK* by the sequence *SendOK*;*ReceiveOK*. *AbTransferLost* would be implemented by both *SendFail* and *SendOK*;*ReceiveFail*. Because *SendOK* is ON, a forward simulation proof would have to synchronize with the last operation of every sequence. Composing the resulting simulation relation with the existing refinements, we find that the synchronization is the one used in [SGH$^+$07].

6.5. The refinements of Butler and Yadav

These refinements develop a Mondex-like money transfer protocol using the B4free tool [B4f]. They will be published as a contribution to [JWe07]. In accordance with the Event-B [AH06] methodology, the protocol is developed in many small, but easily mechanically provable refinement steps, the simulations being forward simulations. The strategy decomposes the abstract events to facilitate separate refinement of distinct pieces to distinct protocol level operations. Aside from that, it is similar to that of [BPJS07] in that failing transfers are refined by *Abort*s.

Note that with the exception of the original (backward) one, the preceding refinements are all forward simulations when viewed at the individual protocol instance level (cf. Corollary 3.10). As such, and particularly when they are based on $(1, 1)$ refinements, they all readily extend to forward simulation refinements of full system runs — just as the original $(1, 1)$ backward simulation readily extended to a backward simulation refinement for full system runs.

6.6. Other Possibilities

Our general theory shows that even more possibilities than have been discussed above are actually possible. For example, the refinement of [BPJS07] could have chosen to refine *AbTransferOK* to *Ack* instead of *Val*, since *Ack*

[9] This differs from [BPJS07], where the *Abort* of the From purse that is bound to happen in this situation implements *AbTransferLost*.

occurs as the last operation of a successful transaction. However, since in general there is a possibility that a transaction succeeds but that the *ack* message is lost, causing the *Ack* operation to be replaced by an *Abort* (which as it turns out is harmless), we infer that in such a refinement there would be a case in which *AbTransferOK* would have to be refined by *Abort*!

An alternative to the preceding is to synchronise right at the beginning, with the first (or second) *Start* event — and there are plenty of hybrid cases, combining aspects from several of the described or suggested refinements arising from the rich structure of the protocol computation tree. We leave the curious reader to work out such scenarios for himself.

6.7. The Non-2-Phase Fragments

In discussing the preceding refinements, we have always assumed that the two *Start* operations are performed first. But it could happen that one purse *Start*s and immediately afterwards *Abort*s, before the second purse has *Start*ed. This spoils the 2P property since the first purse has relinquished its use of its local state before the second purse has claimed its first use. In such a case, either purse may engage in other transactions, changing the local state, after the first purse's *Abort* and before the second purse's *Start*.

A remaining possibility is that only one purse *Start*s, and the other purse merely *Abort*s (as explicitly permitted in the event structure of Fig. 4), or indeed does nothing (a possibility allowed for in the definitions of [SCW00] though not shown in Fig. 4). In such a case, even if the other purse's *Abort* happens after the (inevitable) *Abort* of the first purse, it is arguable that the protocol is nevertheless 2P, since the other purse's use of its state amounts to no more than skip. Even if one does not accept this argument, it is evident that the breakdown of the 2P property is rather mild.

Dealing formally with such situations requires an extension of our theory and provides a major motivation for the meterial developed in Sections 8 and 9. Note though, that even if these situations are not serialisable via the standard 2P technique, the fact that we have $(1,1)$ refinements of the protocol, guarantees nonetheless that these 'rogue' interleavings preserve atomic semantics.

7. Another Motivating Example: Lock-Free Stacks

Up till now, we have considered protocols in which a fixed number of agents engaged in a set of events that together implemented some atomic action, and in which they did so *in peace*, unmolested by other agents — so called isolated protocols. In truth, the isolated protocol concept is of course an idealisation, and in reality, isolated protocols execute within an environment containing other agents.

Isolated protocols are characterised by the fact that interference by other agents can be ignored as soon as the protocol has started, i.e. as soon as each protagonist has executed his first event. In the context of event structures, the mutual exclusion among all the different possibilities can be reflected in a structure that has conflict between the *root events* of every incompatible pair of possibilities. Doing this faithfully for Mondex would result in a truly messy structure, since a choice of (pair of purses, direction of transfer, amount of transfer) cannot be concurrent with another iff at least one of the purses is in common. In real life, the complex control over the possibilities, is of course handled from the environment — and conventional event structures are rather poor at representing efficiently such choices between complex overlapping options.

For Mondex, the preceding difficulty is alleviated by the isolated nature of the protocol, permitting the transparent account of a single protocol run that we gave earlier. However, isolated protocols are not the only protocols of interest in practice, and to maximise the applicability of our earlier theory, we now examine non-isolated protocols, encouraged also by the desire to encompass the non-2P fragments of Mondex noted above.

7.1. Lock-Free Stacks

We introduce a motivating example for non-isolated protocols: lock-free stacks. Lock-free stacks were introduced in [HSY04]. However our treatment will be based on the more recent account in [CG07, CG06]. The operation of a lock-free stack is simple to describe. There are *Push* and *Pop* operations as usual. However their implementation is *optimistic*. Both *Push* and *Pop* are implemented as loops that repeatedly attempt possibly failing procedures *TryPush* and *TryPop* respectively, until success is obtained.

The stack itself is a linked list of cells, with the top pointed at by a global *TopOfStack* pointer, and with each cell

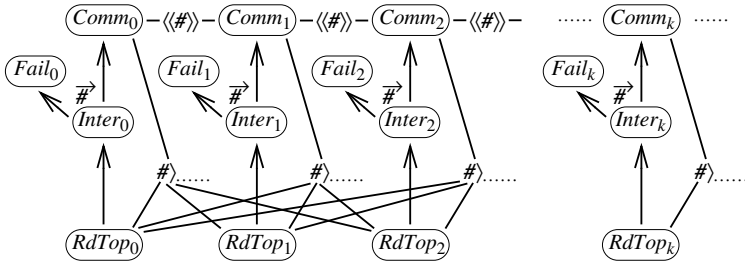

Fig. 6. An event structure that abstracts the *TryPush* and *TryPop* operations in a non-locking stack protocol.

holding a value, and pointing to the next cell down the stack via a *.next* pointer. Each of *TryPush* and *TryPop* works as follows. The global *TopOfStack* pointer is first read. Next, local code prepares the ground: for a *Push*, a new cell is prepared, containing the new value and pointing to the previously read *TopOfStack*; for a *Pop*, the value is extracted from the previously read *TopOfStack* cell, and its *.next* pointer is noted. Finally an attempt is made to atomically update the global *TopOfStack* pointer *provided no other agent has updated it in the meantime*: for *Push*, *TopOfStack* is made to point to the new cell; for *Pop*, it is overwritten with the *.next* pointer.

The atomicity is achieved via the CAS$(loc, oldv, newv)$[10] instruction. In an indivisible operation, this compares the contents of location *loc* with *oldv*, and if they are the same, overwrites *loc* with *newv*, returning success. If they are not the same, no overwrite happens, and failure is returned. Assuming that *oldv* is indeed a value that *loc* previously contained at some point, the atomicity of overwriting is evidently not itself sufficient to guarantee that no other agent has updated *loc* in the meantime, since it might have been altered to *someotherv* and then altered back again, the 'ABA' problem. However, additional mechanisms can be put in place to prevent this, and we will assume that this has been done henceforth. All together, a successful CAS thus guarantees atomicity between the earlier reading of *oldv* and its overwriting by *newv*.

So, when an attempt at *TryPush* or *TryPop* returns, its return value is examined. If it succeeded, *Push* or *Pop* itself returns. If it failed, *TryPush* or *TryPop* is retried. Of course, this permits an infinite sequence of failed attempts, but that's the price of optimism.[11]

Once a new *TopOfStack* value is established via a successful CAS, it can be read by any number of agents, each in the course of preparing his own stack operation, in optimistic anticipation of subsequently committing it via the next CAS — this is what makes the protocol non-isolated. Such a collection of agents, all sharing a common *TopOfStack* value will be called a clan. Obviously, only one member of a clan —the first to try his CAS— can commit his update; the rest will fail. The fact that clans can be non-singletons is what makes the lock-free stack a non-isolated protocol. The fact that each member of a clan acts largely alone, using instructions whose effects are relatively local, and whose conflicts are simple, is what makes the operation of the clan easy to represent using event structures.

Fig. 6 is an event structure that represents the working of a clan, and introduces some new conflict notations. As well as the elements described in Definition 4.1 we have the *transitive* symmetric conflict relation $\langle\langle\#\rangle\rangle$ (so that $a \langle\langle\#\rangle\rangle b \langle\langle\#\rangle\rangle c \Rightarrow a \langle\langle\#\rangle\rangle c$, and if $x \langle\langle\#\rangle\rangle y$, then the occurrence of either blocks the other, as for #); we have the asymmetric conflict relation $\#\rangle$ (for which $x \#\rangle y$ means that the occurrence of x blocks the subsequent occurrence of y, but the occurrence of y does not block the subsequent occurrence of x); and we have the prioritised flow conflict relation $\vec{\#}$ (so that $(a \prec x) \vec{\#} (a \prec y)$ means that x and y are in (symmetric) conflict, and once a is in the current configuration, then x can only be executed provided y is blocked (so that y has priority over x, provided a) — this captures the effects of IF statements in code particularly well).

More formally, we enhance the definitions of Section 4 to the following.

Definition 7.1. A general flow event structure $\mathcal{E}$ (with symmetric, transitive symmetric, asymmetric and prioritised conflict), is a tuple $(E, \prec, \#, \langle\langle\#\rangle\rangle, \#\rangle, \vec{\#})$ such that:

1. E is a set (of events).

[10] CAS stands for Compare And Set.

[11] However, an infinite sequence of failures witnesses that an infinite sequence of attempts by *other* agents succeeded, as becomes clear below, so that the system as a whole makes progress.

2. $\prec$ is an asymmetric causal flow relation on E (whose transitive (resp. reflexive transitive) closure is written $<$ (resp. $\leq$)).
3. $\#$ is an irreflexive symmetric conflict relation on E.
4. $\langle\langle\#\rangle\rangle$ is an irreflexive symmetric and transitive conflict relation on E.
5. $\#\rangle$ is an asymmetric conflict relation on E (we write $\langle\#$ for its transpose where convenient).
6. $\vec{\#}$ is an asymmetric prioritised flow conflict relation on $\prec$ (we write $\overleftarrow{\#}$ for its transpose where convenient).

Definition 7.2. Let $\mathcal{E} = (E, \prec, \#, \langle\langle\#\rangle\rangle, \#\rangle, \vec{\#})$ be a general flow event structure. Let the associated nonsymmetric conflict relation $\#^a$ be the smallest relation on E closed under:

1. $x \# y \Rightarrow x \#^a y \wedge y \#^a x$.
2. $x \#^a y \wedge y \leq z \Rightarrow x \#^a z$.
3. $x \langle\langle\#\rangle\rangle y \wedge y \langle\langle\#\rangle\rangle z \Rightarrow x \langle\langle\#\rangle\rangle z$.
4. $x \langle\langle\#\rangle\rangle y \Rightarrow x \# y$.
5. $x \#\rangle y \Rightarrow x \#^a y$.
6. $(a \prec x) \vec{\#} (a \prec y) \Rightarrow x \# y$

Definition 7.3. Let $\mathcal{E} = (E, \prec, \#, \langle\langle\#\rangle\rangle, \#\rangle, \vec{\#})$ be a general flow event structure with associated nonsymmetric conflict relation $\#^a$. The set $\mathcal{X}_\mathcal{E} \subseteq \mathbb{P}E$ of (legal) configurations of $\mathcal{E}$, and the legal ways of moving from a legal configuration X of $\mathcal{E}$ to a successor legal configuration Y are given by the following rules.

1. $\varnothing \in \mathcal{X}_\mathcal{E}$.
2. $X \in \mathcal{X}_\mathcal{E}$,
   $x \in E - X$,
   $(\forall x' \in E \bullet x' \prec x \Rightarrow x' \in X)$,
   $(\forall x' \in E \bullet x' \#^a x \Rightarrow x' \notin X)$,
   $(\forall a, x' \in E \bullet (a \prec x) \vec{\#} (a \prec x') \Rightarrow (\exists z \in X \bullet z \#^a x'))$
   $\vdash \quad X \cup \{x\} \in \mathcal{X}_\mathcal{E}$.

Obviously these richer general flow event structures also permit the constructions that appear in Definitions 4.4 and 4.5.

The new notations enable Fig. 6 to explain the working of a clan in a compact way. Thus agents k join the clan by executing a $RdTop_k$ event, recording the current *TopOfStack* value; this is regardless of whether they want to do a *TryPush* or a *TryPop*. Each subsequently executes his $Inter_k$ event; this is an abstraction of all the local working done prior to the commit attempt, and in reality corresponds to a large number of individual instructions that give rise to an exponentially large number of interleavings in a concurrent context. Finally comes the commit attempt, the CAS instruction, whose two outcomes are represented by $Comm_k$ and $Fail_k$. The rules for prioritised flow mean that whichever agent is the first to try his CAS, must execute his $Comm_k$ event. The rules for transitive symmetric conflict mean that as soon as he does, all other $Comm_l$ $(l \neq k)$ events become blocked, forcing their agents to execute their $Fail_l$ events in due course.[12] The rules for asymmetric conflict also imply that all remaining $RdTop_l$ events become blocked, so that the next agent to read the *TopOfStack* pointer starts a fresh clan.

7.2. Elimination Stacks

The preceding section gave a simple example of a not-so-isolated protocol, which was easily described using a suitably enhanced event structure. The authors of [CG07, CG06] develop their account of non-locking stacks further by giving an improved account of the *elimination* mechanism, which is of interest for us too.

Basically, when contention for the stack is high, there will be many failed operations on the stack. In such a case, there will be many *Push*ers and many *Pop*ers struggling to access the *TopOfStack* pointer. If we could pair up a *Push*er with a *Pop*er, the former could simply give his data to the latter, avoiding stack contention, and achieving serial semantics. This is what the elimination mechanism tries to do. To achieve it the *Push* loop, whose body previously contained just *TryPush*, now contains *TryPush* ; *TryElimination*. Similarly the body of the *Pop* loop now contains

[12] For later convenience, we note that all failures occur *after* the successful CAS.

TryPop ; *TryElimination*. *TryElimination* is a symmetric procedure that tries to swap a *Pop*er's placeholder piece of dummy data with the *Push*er's real data. The contention inherent in a non-locking approach means that success is not guaranteed.

The elimination mechanism consists of two tasks, which do not get completely separated in the implementation: pairing up *Push*ers with *Pop*ers; and actually carrying out swaps.

To allow *Push*ers and *Pop*ers to meet, clans are formed. The workings of different clans do not interfere, so we only consider a single clan.

Access to the clan is via a location *ClanLoc*.[13] *ClanLoc* contains an agent id. A new agent with id k wishing to join the clan reads the agent id in *ClanLoc* (subsequently referred to as $'k$) and attempts to atomically overwrite it with his own id k, via the CAS mechanism described earlier. If he is unsuccessful (which means that some *other* agent has overwritten *ClanLoc* with *his own* id), he retries, repeating the read-CAS cycle until he succeeds.

The preceding ensures that agents joining the clan form a chain, and that each joining agent k is aware of the id of his predecessor $'k$. The chain condition is not necessary for what follows, but consider the following. If we had a more tightly connected structure than a chain, with more than one successor of some agent a, then if both successors tried to swap with a, then the failure of at least one of them is guaranteed. While a chain arrangement does not in itself *guarantee the absence* of failures, it does at least *permit* their absence as we show below.

As well as *ClanLoc*, there is another critical global data structure, *AgD*, of agent data. *AgD* is an array indexed by agent id's. Each entry in the array is a pair $(INST, data)$, constrained to be of such a size that a single CAS instruction can update the pair atomically. *INST* can be one of $PUSH, POP, NONE$, and reflects the agent's intentions. For the *PUSH* case, *data* is the value to be pushed; for *POP*, *data* is a placeholder for the value to be received; for *NONE*, *data* is either dummy data that a *Push*er has acquired, or the desired data that a *Pop*er has acquired.

An agent k updates his entry in *AgD* prior to CAS-ing *ClanLoc*, so that $AgD(k)$ is accurate as soon as he has joined the chain. Suppose that $GetHim_k$ is the event of successfully joining the chain by agent k, acquiring the id $'k$ of k's predecessor, referred to as *him* in [CG07], in the process. Fig. 7 gives an event structure for the ensuing possibilities.

Agent k then reads $AgD('k)$ to see if the two agents form a complementary pair (i.e. one of them a *PUSH*, the other a *POP*). Suppose that check fails (event $\overline{MPr_k}$ (fail to Match Pair by k)). Then there is no point in agent k trying to implement a swap with $'k$. However, k may by now have acquired, or in the near future may acquire, a successor k'. Since k is unaware of k''s existence, all he can do is wait a while, in the hope that such a k' might implement a swap on his behalf. After a delay, k performs a CAS on $AgD(k)$, comparing with its previous value, and attempting to overwrite it with $(NONE, \ldots)$. If the CAS succeeds (event $\overline{SwP_k}$ (fail to Swap Passive by k)), then there was after all no k', no swap, and k exits the clan unsuccessfully; and whichever of *TryPush* ; *TryElimination* or *TryPop* ; *TryElimination* was under way for k is retried. If the CAS fails (event SwP_k (Swap Passive by k)), then there *is* new data in $AgD(k)$, which must have been put there by some genuine k' who detected a complementary pair and was able to implement the swap. Agent k takes the appropriate action with the data and the *TryElimination* (and hence its calling operation) succeeds.

Suppose, by contrast, that the complementary pair check performed by k succeeded (event MPr_k (succeed to Match Pair by k)). Then k will try to actively implement a swap with $'k$'s data, $AgD('k)$, which he remembers from before.

To implement a swap of two items requires at least three locations and as many updates. For agent k, locations $AgD(k)$, $AgD('k)$ and local data provide the locations, the intention being to exchange the data in $AgD(k)$ and $AgD('k)$. The updates to $AgD(k)$ and $AgD('k)$ must be atomic to prevent interference by k' and $'k$ respectively, who may also be trying to access these locations during their own attempts to implement swaps.

Agent k first tries to overwrite $AgD(k)$ with $(NONE, \ldots)$ using a CAS, and comparing with its previous value. If this fails (event $\overline{SwElMe_k}$ (Swap despite failing to Eliminate Me by k)), then there is new data in $AgD(k)$, put there by k', who accessed $AgD(k)$ earlier, having detected a complementary pair with k. Therefore, if k is a *Push*er, his data has already been taken, and if k is a *Pop*er, he can extract the new data from $AgD(k)$, exiting successfully in both cases.

If the CAS succeeds (event $ElMe_k$ (Eliminate Me by k)), k has prevented k' from interfering with his swap, and must now complete his swap process by overwriting $AgD('k)$ with $(NONE, data_k)$ (where $data_k$ is k's original data, previously stored in the second component of $AgD(k)$), comparing $AgD('k)$ with its previous value. If this last CAS fails (event $\overline{ElHim_k}$ (fail to Eliminate Him by k)), then $'k$ must have performed his $ElMe_{'k}$ earlier on $AgD('k)$, and k's swap attempt fails; k exits and must retry from the beginning.

If this last CAS succeeds (event $SwElHim_k$ (Swap achieved via Eliminate Him by k)), then k has installed his own data $data_k$ in $AgD('k)$ while simultaneously preventing $'k$ from completing his own swap attempt. Therefore, if k is a

[13] Thus by having as many *ClanLoc*s as is considered useful, as many clans as is considered useful can be run in parallel, reducing contention, and maximising concurrency. The mechanisms for determining how many *ClanLoc*s should be maintained is beyond the scope of this paper.

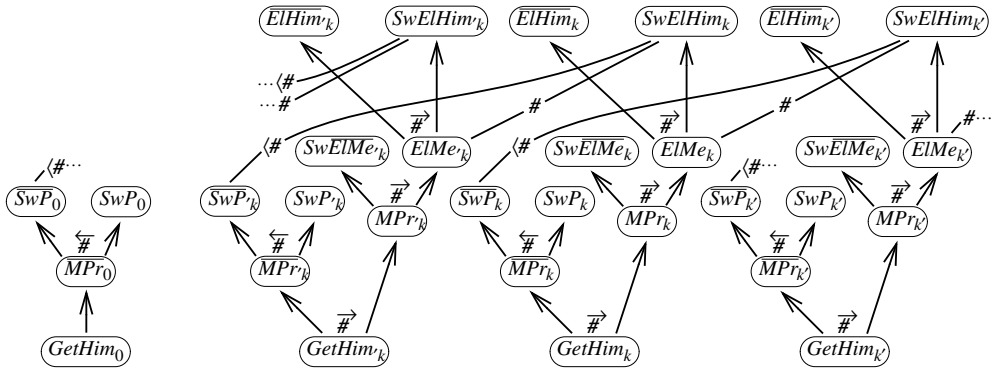

Fig. 7. An event structure illustrating the working of a clan in the elimination part of a non-locking stack protocol.

*Push*er, he has succeeded in passing his data to *Pop*er $'k$, and if k is a *Pop*er, he already has *Push*er $'k$'s data which he read earlier. In either case k can exit successfully.

The above account of the 'main path' through the behaviour of a clan leaves a few loose ends to be tied up. Firstly, the first agent in the clan, agent 0, has no predecessor to work on; his only hope therefore is to be swapped passively, and Fig. 7 shows the simpler event tree that he consequently has. Secondly, the growth of the clan, i.e. the accumulation of agents attaching themselves to the chain, stops as soon as the current last agent executes his last CAS, i.e. as soon as his *SwElHim* or $\overline{ElHim}$ takes place. The effect of either of these is to set the instruction field in his *AgD* slot to *NONE*, preventing any new arrival from having a viable predecessor to actively swap with, i.e. any new arrival becomes a new agent 0. Thirdly, any member of the chain similarly completing his *SwElHim* or $\overline{ElHim}$ breaks the chain into independent pieces, since swapping actvity among predecessors lower down the chain can no longer be interfered with by swapping actvity among successors higher up the chain. Consequently, if the chain happens to be of even length, and all the odd-numbered agents happen to successfully complete a *SwElHim* event, then all agents in the chain succeed. It is in this sense that the chain arrangement *permits* the absence of failures, while nevertheless *not guaranteeing* their absence, since nothing guarantees that the dynamics of a clan will generate as ideal outcome as just described.

8. Not-So-Isolated Atomic Actions and their Protocols

The preceding section described two not-so-isolated protocols, the non-locking stack, and its enhancement, the elimination stack. In the case of the former, we gave a straightforward description of the protocol via an (enhanced kind of) event structure, as in Section 4. This means that we can generate a protocol computation DAG for the non-locking stack simply by generating all the configuration sequences of a suitable relational model of the event structure in the manner described in Section 4.

In the case of the elimination stack however, matters are a bit more complicated. The reader will have observed that the description of the working of the elimination stack, effectively breaks up into three separate event structures, one of them not mentioned at all. The first is the event structure that describes the non-locking stack in Fig. 6, since the original *TryPush* and *TryPop* form the first (and if successful, only) step of the more elaborate mechanism. The second, not described, captures how various agents compete to attach themselves to an elimination clan via the read-CAS loop whose success is a *GetHim* event, once the *TryPush* or *TryPop* has failed. Thirdly, we have the event structure that describes the elimination mechanism itself, Fig. 7, once an agent has successfully attached to a clan.

The event structure account was structured thus, because an integrated description, unifying all the features in a single event structure, would be unhelpfully complicated. The full working of the elimination stack, contains, like the global description of the Mondex protocol in a context of many purses and many transactions, a lot of overlapping complex possibilities, hard to capture succinctly by means of event structures.

So, as before, our event structure account was generic, bringing out certain aspects while suppressing others (particularly low level state details). When these are reinstated, we get a replication of the forest (or DAG) of sequences of configurations generated by the various event structures, and indexed by the reinstated data values. In a multi-stage protocol, such as our description of the elimination stack, once this is done, it then becomes relatively easy to glue to-

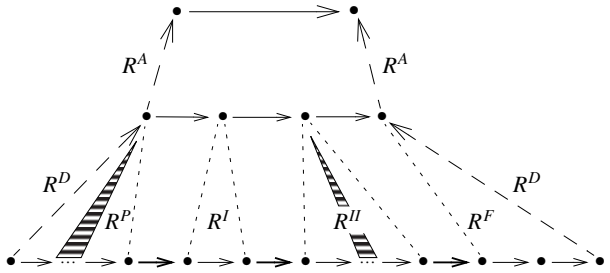

Fig. 8. A 'slice' through the factorisation of the refinement of an atomic action to a protocol, through the agent (middle) layer. The middle layer contains actions that characterise, in an atomic fashion, what each agent participating in the protocol achieves. The protocol layer below, contains a multiple synchronisation assignment, with a selected protocol step for each agent atomic step. The thin dark-hatched triangles are 'jolts,' in which agents other than the ones participating non-trivially in the protocol are able to interfere with the protocol state in a controlled manner. The two jolts shown are both trivial at the agent level since the protocol state changes both abstract to the identity.

gether appropriate copies of the DAGs corresponding to the different stages, to get an overall description, and thereby a computation DAG for the whole protocol.

Two issues —granularity and multiple agents— now need to be considered. Both affect the refinement relationship between atomic and protocol levels.

Regarding granularity, unlike Mondex, for which all paths through the protocol are finite *a priori*, the non-blocking nature of the stack algorithms opens the possibility that some activity may be repeated indefinitely without ever meeting with success. In part, such possibilities may be avoided by drawing the boundaries of the protocol appropriately. For example, in the case of the non-blocking stack, one can say that the protocol consists of the *TryPush* and *TryPop* routines themselves, each with two finite outcomes (success and failure), rather than the enclosing *Push* and *Pop* operations that only have a single finite outcome (success), but also the possibility of non-termination. However this approach is not fireproof. In the case of the elimination stack, the loop body of *Push* is *TryPush* ; *TryElimination*, and similarly for *Pop*. Considering the loop body alone does not avoid non-termination, since the middle stage of *TryElimination*, joining the chain via the *GetHim* event, is not guaranteed to succeed in a finite number of steps. So, even though choosing the granularity of a protocol description in a convenient way may avoid some cases in which infinite paths arise, we neverthless cannot completely avoid dealing with protocols that have infinite paths. On the other hand, note that an infinite sequence of failed attempts to achieve some goal is inevitably unfair. This gives us grounds for according infinite paths that arise in this way a different status to finite paths.

We can divide infinite paths into three categories. An infinite path can: (1) consume input and or generate output, (and perhaps also manipluate the state in a nontrivial way); (2) neither consume input nor generate output, but manipluate the state in a nontrivial way; (3) do no I/O and furthermore manipluate the state in a way that (according to a suitable notion of observation) is observably trivial. Our contention is that all the infinite paths that arise in our study of protocols fall into the third category. Certainly that is the case for the examples considered in Section 7 — the infinite paths that arose there consisted of indefinitely repeated attempts to access some resource via an action whose effects were null if unsuccessful. Given that protocols (as we have them in this paper) are intended to accomplish (one of) a finite number of outcomes in a finite amount of time, a sequence of steps that accomplishes an infinite number of observable state changes or I/O actions cannot really be regarded as belonging to a single protocol instance. This forces option (3) in the classification above, and furthermore forces the conclusion that an unobservable state change must necessarily be *defined by* the property that it maps (under the retrieve relation of the refinement between atomic and protocol levels) to an identity on the atomic state. In its turn, this then opens the way to two approaches to dealing with such, relatively innocuous, infinite paths. In the first, we allow protocols to have 'jolts' in them, modelled by a special purpose agent Jolt, whose transitions trivialise under abstraction — an infinite path then becomes an infinite sequence of jolts in an infinite sequence of finite protocol executions by other agents. In the second, we develop extensions of the theory of Section 3 to accomodate these very special infinite paths. We develop the tools for the first approach below.

Regarding multiple agents, unlike Mondex once more, in which all participating agents cooperate towards the achievement of a single goal, making it sensible to represent the various possible collective outcomes as single actions at the atomic level, for the non-blocking protocols, the agents compete for resources in order to achieve their own individual goals, and a representation of various possible collective outcomes as single actions at the atomic level is (while perfectly possible) considerably less useful. However, multiple agents are not hard to incorporate into the

framework of Section 3. One can take the setup in Section 3, and factor the refinement from atomic to protocol levels via an intermediate level. Let us call it the agent level. Fig. 8 shows a 'vertical slice' through the arrangement that ensues.

At the top we have the usual shallow tree description of all the possible collective outcomes as single atomic actions — Fig. 8 shows one of the possibilities. The next layer down is a computational DAG, as in Section 3, but to enable the maximum reuse of the results of Section 3, it is restricted to be a forest — Fig. 8 shows one path through it, and the 'big step' abstraction function R^A that maps the endpoints of the path to the before- and after- states of the single atomic action above that the path refines. At the bottom, the protocol layer is as before. Fig. 8 shows a single path through it, and the 'big step' abstraction function R^D that maps the endpoints of the path to the before- and after-states of the agent level path above.

Fig. 8 also shows how several individual steps of a path through the protocol DAG are mapped, in a 1-1 order-preserving manner, to (all) the steps in the agent level path itself. This is a multiple synchronisation assignment (MSA). Obviously the 1-1 order-preserving property has to be consistently maintained across all protocol DAG paths and their mapping to agent level paths above.[14]

Fig. 8 also shows how in the presence of an MSA we not only have the past oriented and future oriented retrieve relations R^P and R^F that we had before, but various intermediate retrieve relations R^I, R^{II}, ... too. It is not hard to see that these can be calculated via formulae very similar to (12) and (13). It is equally easy to see that the results in the latter part of Section 3 have analogues for this more general world.

Finally, the dark-hatched shapes in Fig. 8 are the jolts of which we spoke earlier. Jolts are points in the running of the protocol in which agents other than those deemed to be participating in the current protocol run can interfere with it, i.e. they can modify the current protocol state. Such jolt-attributed state changes must be dealt with by the protocol in an appropriate fashion. Ideally, jolts map to the identity at the agent level, but not always. When they do, the agent level clearly abstracts cleanly back to the abstract level, and this yields a rationale for allowing such jolts to be (almost) ignored altogether — certainly in the abstraction from agent level to abstract level. When they don't, the relationship between the abstract and agent levels becomes more complicated, since the interference is no longer invisible there, and appropriate arguments have to be advanced about the way in which the agent level paths with non-trivial jolts relate to the abstract level. We will see instances of both types of behaviour below.

In a world of not-so-isolated protocols, tangles of protocol instances that are unbounded in time and space may potentially arise. Jolts give us the capability of cutting such tangles into finite pieces. Given our earlier insistance that a protocol instance must consist of a finite sequence of steps, the infinite number of steps inherent in an unbounded tangle of instances must be capable of being cut up into finite pieces that do not interfere destructively with each other, if our approach is to make sense in the not-so-isolated world.

We now look at the details of the approach proposed above. Where the changes from the treatment of Section 3 are relatively few, we just list them explicitly, and describe their consequences for the theory as a whole. Where there are more extensive differences, we give fuller details. As a meta level notational device, we distinguish previous concepts from their current versions using a '$^\Delta$' superscript. So $Atomic^\Delta$ refers to the definition of the atomic level in Section 3, whereas *Atomic* refers to the atomic level here.

The Atomic Level

- The atomic level is defined as in Section 3. So *Atomic* (here) has the same properties as $Atomic^\Delta$ (there); i.e. it is a shallow computation tree. (28)

The Agent Level

The agent level captures the decomposition of the purely atomic level (in which the state changes capture the various alternative overall goals of the protocol), into atomic steps of individual agents participating in the protocol, capturing the protocol's overall goals as seen from individual agents' perspectives. This admirably clean picture is made more cluttered by needing to allow for agent level jolts.

- The agent level *Agent* is defined like a special case of $Protocol^\Delta$; i.e. it is a computation DAG with all paths finite, but in addition, it is restricted to be a forest. (29)
- The agent state space *VA* factors into a product of subspaces $VA = VA_1 \times VA_2 \times \ldots$, as in Definition 5.3, so that each step of the *Agent* transition relation modifies (in the sense of footnote 6) at most one of them. (30)

[14] N. B. Recalling Mondex, despite it being an isolated protocol and having an overall goal, the agent level makes sense. There are evidently two agents, the From purse and the To purse, and a run of Mondex accomplishes either: (1) a *Send* by the From purse followed by a *Receive* by the To purse, or (2) a *Send* by the From purse followed by an *Abort* by the To purse, or (3) *Aborts* by both purses. Cf. Section 6.4.

- There is a set of *protocol agents*. There is a bijection between the subspaces of *VA* and the protocol agents, so that VA_{PA} is the subspace associated to *PA*. (31)
- There is an additional agent, Jolt, distinct from all the protocol agents in (31). (32)
- Each step of the *Agent* transition relation is executed either by a protocol agent *PA*, in which case it modifies the subspace associated with *PA*, or by Jolt, in which case it may modify any subspace. (33)
- No protocol agent *PA* may execute more than one step of a maximal path through the *Agent* transition relation, and no pair of consecutive steps is executed by Jolt. (34)

To connect the jolted agent level with the abstract level, we need to disregard the jolts in a suitable way.

- Let $Agent^{\overline{\mathsf{J}}}$ be the subforest of *Agent* consisting of all its Jolt-free paths. (35)
- Then there is a functional 'big step' retrieve relation R^A between *Atomic* and $Agent^{\overline{\mathsf{J}}}$, and R^{AP} and R^{AF} are the related past and future retrieve relations. Using these retrieve relations, all the results of Section 3 hold between *Atomic* and $Agent^{\overline{\mathsf{J}}}$. (36)

The jolted and non-jolted parts of *Agent* need to be connected.

- There is a function $\overline{\mathsf{J}}$ which maps any maximal path of *Agent* which contains a Jolt-executed step as initial step or final step (or both) but which contains no Jolt-executed step in its interior, to a path of $Agent^{\overline{\mathsf{J}}}$, and such that $\overline{\mathsf{J}}$ is injective on the non-Jolt steps and the nodes of the non-Jolt steps and preserves the agent that executes each staep. (37)

The function $\overline{\mathsf{J}}$ relates an *Agent* path in which jolts occur only at the extremities, to an $Agent^{\overline{\mathsf{J}}}$ path in which such jolts are disregarded, allowing the $Agent^{\overline{\mathsf{J}}}$ path to be abstracted to an atomic transition, and relating the original *Agent* path to that abstraction. *Agent* paths with jolts in arbitrary places must be transformed into paths with jolts only at extremities before they can be abstracted, a matter to which we attend below.

One way of generating suitable functions $\overline{\mathsf{J}}$ on paths in a systematic way is to view the *Agent* transition relation as being generated via the unwinding of the paths of a suitable DAG transition relation, similar to the way that Definition 4.5 is related, by unwinding, to Definition 4.4 in the event structure world. With a suitable DAG, the paths without any Jolt-executed step can arise as the interior portions of paths *with* Jolt-executed steps at their extremity (or extremities), and the unwinding can unambiguously relate the latter to the former.

The Protocol Level

- The protocol level *Protocol* is defined like $Protocol^{\triangle}$; i.e. it is a computation DAG with all paths finite. (38)

To handle multiple synchronisation assignments and other modelling aspects properly, the agents and their properties need to be reflected in the protocol level.

- The protocol state space V factors into a product of subspaces $V = V_1 \times V_2 \times \ldots$, as in Definition 5.3. (39)
- For each protocol agent *PA*, there is a subset $V_{PA} = \{V_{PA,1}, V_{PA,2}, \ldots\}$ of the subspaces of V. For distinct PA_1 and PA_2, $V_{PA_1} \cap V_{PA_2}$ need not be empty.
- For each step St_γ of the *Protocol* transition relation, there is a protocol agent *PA*, or Jolt, that executes it. (40)
  - If St_γ is executed by a protocol agent *PA*, then it modifies at most one state subspace V_k, $V_k \in V_{PA}$, and St_γ may do I/O in the usual manner.
  - If St_γ is executed by Jolt, then St_γ may *not* do I/O.

We use the machinery of Section 3, superscripting with '^{A}' or '^{P}' to indicate the agent or protocol level where needed. The refinement from *Agent* to *Protocol* is then captured by (7)-(11) with uses of $Atomic^{\triangle}$ replaced by $MPath^A$, and other obvious notational changes.

Definition 8.1. Let $Path^A(\ldots)$ be a contiguous fragment of an agent path, and $Path^P(\ldots)$ be a contiguous fragment of a protocol path. We say that $Path^A(\ldots)$ and $Path^P(\ldots)$ are Σ-completable iff $Path^A(\ldots)$ and $Path^P(\ldots)$ are exendable (in either direction or both) to maximal paths $MPath^A(\ldots)$ and $MPath^P(\ldots)$ which witness the obvious analogue of (11). Any such extension is called a Σ-completion of $Path^A(\ldots)$ and $Path^P(\ldots)$.

Definition 8.2. Let v_I be an initial state of *Protocol*, and let R^D be the (functional) 'big step' retrieve relation between the *Agent* and *Protocol* transition relations. A multiple synchronisation assignment ($\mathrm{MSA}(v_I)$) for the valid DAG from

v_I is a subset of its non-Jolt steps, such that for each maximal path $MPath^P(v_I,\ldots,v_F)$ through the valid DAG from v_I: if $R^D(av_I,v_I)$ where av_I is an initial state of *Agent* and $R^D(av_F,v_F)$ where av_F is a final state of *Agent* both hold, and if $MPath^A(av_I,\ldots,av_F)$ is a(ny) maximal path of the valid DAG in *Agent* from av_I to av_F, then there is an order preserving bijection θ from all the non-Jolt steps of $MPath^A(av_I,\ldots,av_F)$ to the MSA(v_I) steps of $MPath^P(v_I,\ldots,v_F)$, such that for every St^A_γ in $MPath^A(av_I,\ldots,av_F)$, the same protocol agent executes both St^A_γ and $\theta(St^A_\gamma)$. An MSA(v_I) for every v_I constitutes an MSA for *Protocol*.

Definition 8.3. Let an MSA for $MPath^P(v_I,\ldots,v_F)$ and $MPath^A(av_I,\ldots,av_F)$ etc., be as in Definition 8.2. If the bijection θ extends to θ^{J}, in which the order preserving bijection on non-Jolt steps of extends to an order preserving inclusion of all the Jolt steps of $MPath^A(av_I,\ldots,av_F)$ also, then we say that $MPath^P(v_I,\ldots,v_F)$ and $MPath^A(av_I,\ldots,av_F)$ are Jolt-compatible. An MSA for *Protocol*, which is Jolt-compatible for every $MPath^P$ and any abstraction $MPath^A$ of it is a Jolt-compatible MSA for *Protocol*.

Definition 8.4. We define the 'past oriented' retrieve relations $R^{\gamma P}_\rho$:

$$\begin{aligned}R^{\gamma P}_\rho(av_s,v_t) \equiv (\exists\ & av_I, aj_1, ap_1, av_1, \ldots, aj_s, ap_s, \langle\alpha,\ldots,\gamma\rangle, v_I, j_1, p_1, v_1, \ldots, j_t, p_t, \langle\pi,\ldots,\rho\rangle \bullet\\ & R^D(av_I,v_I) \wedge FPath^A_{\langle\alpha,\ldots,\gamma\rangle}(av_I, aj_1, ap_1, \ldots, aj_s, ap_s, av_s) \wedge FPath^P_{\langle\pi,\ldots,\rho\rangle}(v_I, j_1, p_1, \ldots, j_t, p_t, v_t)\ \wedge\\ & R(av_I,v_I), FPath^A_{\langle\alpha,\ldots,\gamma\rangle}, FPath^P_{\langle\pi,\ldots,\rho\rangle} \text{ are } \Sigma\text{-completable with all } \Sigma\text{-completions Jolt-compatible}\ \wedge\\ & \text{the number of steps in } \langle\alpha,\ldots,\gamma\rangle \text{ and the number of MSA}(v_I) \text{ steps in } \langle\pi,\ldots,\rho\rangle \text{ are equal})\end{aligned} \tag{41}$$

and the 'future oriented' retrieve relations $R^{\gamma F}_\rho$:

$$\begin{aligned}R^{\gamma F}_\rho(av_s,v_t) \equiv (\exists\ & aj_{s+1}, ap_{s+1}, av_{s+1}\ldots, av_F, \langle\gamma,\ldots,\delta\rangle, j_{t+1}, p_{t+1}, v_{t+1}\ldots, v_F, \langle\rho,\ldots,\tau\rangle \bullet\\ & FPath^A_{\langle\gamma,\ldots,\delta\rangle}(av_s, aj_{s+1}, ap_{s+1}, av_{s+1}\ldots, av_F)\ \wedge FPath^P_{\langle\rho,\ldots,\tau\rangle}(v_t, j_{t+1}, p_{t+1}, v_{t+1}\ldots, v_F) \wedge R(av_F,v_F)\ \wedge\\ & FPath^A_{\langle\alpha,\ldots,\gamma\rangle}, FPath^P_{\langle\pi,\ldots,\rho\rangle}, R(av_F,v_F) \text{ are } \Sigma\text{-completable with all } \Sigma\text{-completions Jolt-compatible}\ \wedge\\ & \text{the number of steps in } \langle\gamma,\ldots,\delta\rangle \text{ and the number of MSA}(v_I) \text{ steps (for some } v_I\text{) in } \langle\rho,\ldots,\tau\rangle \text{ are equal})\end{aligned} \tag{42}$$

With these tools in place, we can prove evident analogues of (14)-(19) in which arbitrary *Protocol* steps can figure in various simulation-related POs, featuring appropriately chosen R relations from the family just defined. One can then complete the programme of Section 3 in this more complicated setting. There are minor changes of notation and terminology, but insisting that the *Agent* transition relation is a forest ensures that even Corollary 3.9 survives in an appropriate form. Also helpful, is the fact that the additional detail we introduced concerning protocol agents is essentially irrelevant to the mathematical requirements of the refinement proofs.

We now turn our attention from refinement proofs to serialisation, which is inevitably going to be more complicated in the presence of jolts. Unlike the situation in Sections 3 and 5, where serialisation was purely a matter for concern at system run time, we now need to lift some of the reasoning to the atomic-agent-protocol world. First we introduce some terminology.

Definition 8.5. Suppose given an *Agent* transition system and a corresponding *Protocol* transition system, and the refinement machinery given by R^D. Suppose also given a Jolt-compatible MSA for *Protocol* as in Definition 8.3. A contiguous subsequence of steps of a maximal path through the protocol DAG not including a Jolt step is called a portion. A portion is called a:

1. skip equivalent no external output portion (SENXOP) iff the before-state of its first step equals the after-state of its last step, and no step produces an output which is not input by another step in the same portion,
2. skip equivalent no external input portion (SENXIP) iff the before-state of its first step equals the after-state of its last step, and no step consumes an input which has not been output by another step in the same portion,
3. general no external output portion (GNXOP) iff it is not a SENXOP and no step produces an output which is not input by another step in the same portion,
4. general no external input portion (GNXIP) iff it is not a SENXIP and no step consumes an input which has not been output by another step in the same portion,
5. complex portion (CP) iff it is not a SENXOP, SENXIP, GNXOP or GNXIP.

In addition, any contiguous subsequence of Jolt steps is called a:

6. complex jolt (CJ) iff it includes the image under the θ^{J} function of a Jolt step of *Agent*,

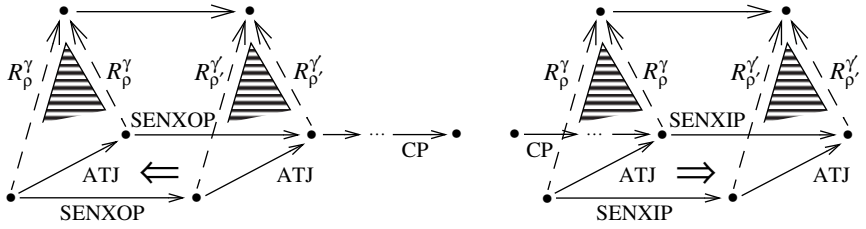

Fig. 9. The shifting of abstractly trivial jolts past SENXOPs and SENXIPs. On the left, the past of a CP contains an ATJ which occurs after a SENXOP, so the ATJ is pushed further into the past. On the right, the future of a CP contains an ATJ which occurs before a SENXIP, so it is pushed further into the future.

7. abstractly trivial jolt (ATJ) iff the before-state of its first step differs from the after-state of its last step, but it does not include the image under the θ^{J} function of a Jolt step of *Agent*,
8. skip equivalent jolt (SEJ) iff the before-state of its first step equals the after-state of its last step, and it does not include the image under the θ^{J} function of a Jolt step of *Agent*.

Evidently the status of jolt at *Protocol* level depends critically on the synchonisation assignment and on its extension to *Agent* jolts via θ^{J}. Of the three kinds of jolt, SEJs are the least objectionable as regards disturbing serial semantics — they are not even observable within *Protocol* maximal paths. This is not to say that they can be dismissed entirely, since a read-only access to some state that is being used by another transaction can reveal information which can subsequently be used to influence a system run in a way that violates serial semantics. However we will assume that this is not the case for SEJs. In other words:

- We **assume** that the fact that the *Protocol* level paths that SEJs intrude into cannot observe them, is mirrored by a corresponding inability of any system run as a whole to observe them. (43)

What (43) says is that there are *higher level system invariants* that maintain serial semantics despite the intrusions of SEJs.

The other kinds of jolt are more drastic than SEJs. Both CJs and ATJs can be observed within *Protocol* level and/or *Agent* level paths, so, if we are to recover any semblance at all of serial semantics, we must somehow get rid of such jolts from the interior of *Protocol* and *Agent* paths. We adapt some concepts from 2P theory.

Definition 8.6. (Assuming the preceding machinery) a maximal path in a protocol DAG is jolt-normal iff:

1. it contains exactly one (real or deemed) CP — if there is no real CP (as given in Definition 8.5.5), either the before-state of the first step of a SENXIP or GNXIP, or the after-state of the last step of a SENXOP or GNXOP, may be called the *deemed* CP; both eventualities are covered by the phrase '(deemed) CP',
2. before the (deemed) CP, the only kinds of portion that precede a CJ or ATJ are SENXOPs and GNXOPs,
3. after the (deemed) CP, the only kinds of portion that follow a CJ or ATJ are SENXIPs and GNXIPs,
4. the steps of the path are partitioned into contiguous subsequences, each of which is a (deemed) CP, SENXIP, GNXIP, SENXOP or GNXOP.

In addition:

5. a protocol DAG is jolt-normal iff every path is jolt-normal.

We now describe how we can exploit the preceding for serialisation.

In Lemma 5.7 and in Theorem 5.10 we felt able to interchange steps because the state components they affected were disjoint. Innocuous though this interchange of steps may be, the serialisability that it leads to is still a definition of a notion of correctness, which exists independently of, and cannot be derived from, other considerations (in particular from the notions of refinement that we have worked with). In the current context, we will be forced to adopt an even weaker notion of correctness connected with serialisation, and we make up for the laxitude this introduces by assuming that what it permits is acceptable when measured against the requirements of the system. This again brings us back to the presence of higher level system invariants, without which the semantics of the kind of 'porous' transactions we are contemplating becomes potentially nonsensical.

For protocol paths containing non-trivial jolts (i.e. ATJs and CJs) in their interior, our goal is to push the ATJs and CJs to the extremities of the path, after which they can be reasonably disregarded as far as the current atomic goal is concerned.

Let us look at the elimination of an ATJ adjacent to a SENXOP. If an occurrence of an ATJ (with before-state v and after-state v') in a path of a jolt-normal protocol is preceded by an SENXOP (with before- and after- states both v'), we can hope that we could shift the ATJ past the SENXOP, replacing the SENXOP with one that has before- and after- states both v, generating thereby some other path in the protocol DAG which was semantically acceptable as a substitute for the original one, according to the requirements of the system. If such a thing is possible we say that the ATJ has been *shifted* into the past. The left hand side of Fig. 9 illustrates this for the case that the SENXOP is a refinement of an agent level transition (according to suitable $R_\rho^{\gamma P}$ or $R_\rho^{\gamma F}$ retrieve relations); the right hand side of Fig. 9 illustrates the dual case of pushing an ATJ into the future past a SENXIP. There are degenerate cases in which the SENXOP refines the identity and the agent level consists of just a single state, of course. Pushing into the past is desirable for ATJs occurring before the CP of a jolt-normal protocol, and pushing into the future is desirable for ATJs after the CP.

At this point we emphasise one thing. Unlike the case of Definition 5.6 and Lemma 5.7, in which individual steps were interchanged, the shifts of Fig. 9 (and of Fig. 10 below) involve the interchange of contiguous sequences of steps *en bloc*. Since the states involved are in principle shared, the detailed interchange of a step at a time would almost inevitably break many detailed low level invariants while it is in progress, so we do not even contemplate such an approach. Instead, the *en bloc* shifting approach, albeit avoiding such pitfalls, nevertheless needs to be approached with care, since the applicability of a shift may well depend on a global property of the protocol transition system as a whole. Thus it must be the case that in all system runs, the adjacent presence of the two continguous subsequences of the before-side of the shift, can in all cases, be replaced by the two continguous subsequences of the after-side of the shift, and the result is still a valid system run, before the use of the shift can be sanctioned as a local transformation. This departure from purely local reasoning, while obviously regretable, is a more or less unavoidable consequence of interfering with the differing uses by different agents of shared state.

Let us now replay the above for a CJ. The analogue here would be to shift the CJ past a SENXOP. Since a CJ abstracts to a non-trivial step at agent level, the analogue of Fig. 9 would have a quadrilateral not only at the lower protocol level, but also at the higher agent level. Unfortunately the agent transition relation has been stipulated to be a forest, so it contains *no* quadrilaterals. Thus instead of being able to argue about purely local modifications to paths, moving jolts and replacing SENXOPs in a small portion of the path, we must introduce mappings on paths as a whole, and to say that one agent path is the CJ-shifting of another.

One way of generating such mappings on paths in a systematic way is to again view the *Agent* transition relation as being generated via the unwinding of the paths of a suitable DAG transition relation, as for Definition 4.5 and Definition 4.4 in event structures, as noted earlier. With such a DAG, and the mapping on paths round one of its quadrilaterals available, the required mapping on paths in the forest is then generated by relating the two DAG paths to their forest representatives, and extending to every maximal path that enters and leaves that quadrilateral.

We note moreover that a functional big step retrieve relation between agent and protocol levels also means that distinct *Agent* final states must refine to distinct *Protocol* states, so that the absence of a quadrilateral at the *Agent* level implies a corresponding absence at *Protocol* level. Thus the necessity for mappings on paths as a whole propagates down a refinement hierarchy when retrieve relations are functional.

Thus far we have explored shifting ATJs and CJs past skip equivalent portions, since that leads to quadrilaterals in the *Protocol* transition relation with the same state along two of its sides, which is the easiest situation to visualise and describe. But there is no reason to stop there. Obviously, when shifting a CJ which starts with before-state v say, and ends with after-state v' say, its source and destination positions in the shift must both start with v, and must both end in v'. After all, a CJ describes activity specifically *outside* the control of the current protocol path, so there is no possibility to change any aspect of it by arguing locally from *inside* the current protocol path. However, that is not to say the lead-in to v must be of a prescribed form before and after the shift, nor that the lead-out from v' must be of a prescribed form before and after the shift. Obviously, if we are going to reason locally (or in a manner that at least corresponds to local reasoning after unwinding, as above — and as is highly preferable), then the before and after paths must diverge at some common state v^{div} say, and (in the DAG picture) must converge again at some common state v^{con} say. This yields a formulation of shifting past GNXOPs and GNXIPs, illustrated in Fig. 10. There, the CJ is shifted, in a manner that presumes to make progress towards expelling the CJ to one end or the other of the resulting maximal path, but no assumptions are made about the segments of protocol path that act as lead-in to v after divergence from v^{div}, or that act as lead-out after v' until convergence to v^{con}, save the absence of external input or output, as appropriate. In fact, once a CJ is sufficiently close to one or other extremity of a maximal path, there is nothing to prevent the interpretation of all or part of some remaining outlying GNXOP or GNXIP as itself a jolt (from the point of view of the current protocol), and thus to expel it from the current transaction. In Section 10.3 we will see an example in which this flexibility is exploited to maximum effect.

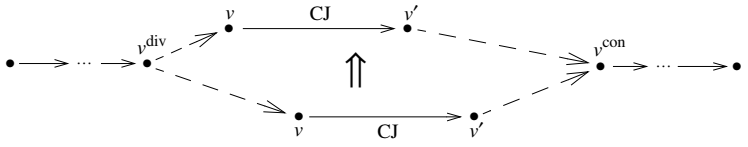

Fig. 10. A general shift of a CJ.

Definition 8.7. A *Protocol* transition relation is *resolvable* iff, via series of shifts, any maximal path can have its ATJs and CJs moved to the beginning and/or end of the resulting path, modulo the reinterpretation of any remaining GNXOP or GNXIP as itself a jolt.

The end result of resolving all the ATJs and CJs need not be a completely jolt-free protocol path: SEJs may remain in its interior. Such interruptions, which amount to 'subliminal skips' during the protocol path, are tolerated (modulo the assumption of global unobservability), since they are both unobservable at protocol level, and also furnish a mechanism whereby a potentially unbounded (in space and or time) tangle of interfering transactions may be cut up into finite pieces, as noted earlier.

9. Serialisation of Resolvable Jolted Transactions

In this section we present a serialisation construction appropriate to the 'jolted' protocols of Section 8. We reiterate that shifting amounts to a notion of correctness. Therefore the serialisability property that rests on it is also a notion of correctness. As before, we just refer to the earlier treatment where the differences are slight, giving more detail where it is more important to do so.

- As in Definition 5.1, a system has a number of system state subspaces $W_1, W_2, \ldots$ and the the total system state space $W = W_1 \times W_2 \times \ldots$ is the product of all of them. There is also a number of *system agents*, $A_a, A_b, \ldots$, each associated with one or more of the system state subspaces, eg. $W_a = \{W_{a_1}, W_{a_2}, \ldots\}$ for A_a etc. (44)

Note the distinction between *system agents* and *protocol agents* introduced in the previous section. The latter will shortly be matched to the former.

The definition of an instantiation of a *Protocol* maximal path, Definition 9.1 next, is comparable to Definition 5.4 above, but is more complicated for a number of reasons: (1) we represent agents at both protocol and system levels, to model the various agents active in (especially) not-so-isolated protocols; (2) we give each agent the possibility of having several distinct state subspaces, to conveniently model both shared and private local state; (3) we allow distinct agents to have state subspaces in common, to model shared state; (4) in Definition 9.1 we utilise more of the 'runtime information' contained within the image of the matching compared with Definition 5.4, in order to allow more flexibility for the jolts.

- A system run is defined as in Definition 5.2, except that re. point 6, each step involves change to *one of* the system state subspaces associated with the agent who executes it. (45)

Definition 9.1. Let $Atomic, Agent, Protocol, \ldots$ (with all the attendant machinery) be a protocol implementing an atomic action in the sense of the previous section. We say that system run $\mathcal{T}$ instantiates *Protocol* iff there is a maximal path through the protocol $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}(v_I, j_1, p_1, v_1, j_2, p_2, v_2, \ldots, v_{F-1}, j_F, p_F, v_F)$ and three maps: τ_{Ag}, τ_A and τ_S such that:

1. τ_{Ag} is an injective function from the set of protocol agents of *Agent* to the set of system agents,[15]
2. for each protocol agent *PA*, there is an injection $\tau_{ss}(PA, -)$ from the set V_{PA} (of the subspaces of V associated with *PA*) to the set $W_{\tau_{Ag}(PA)}$ (of the subspaces of W associated with $\tau_{Ag}(PA)$),
3. for each protocol agent *PA*, for each subspace $V_{PA,l}$ in V_{PA}, there is a map $\tau_{PA,l} : V_{PA,l} \to W_{\tau_{ss}(PA,l)}$, and $\tau_A = \Pi_{PA,V_{PA,l}}\, \tau_{PA,l}$ (where the product ranges over the subspaces $V_{PA,l}$, for each *PA*),

[15] In particular, Jolt is never in the domain of τ_{Ag}.

4. τ_S is an injective function defined on (all) steps of $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$ — if a step is executed by a protocol agent *PA*, then τ_S maps it to a single step of $\mathcal{T}$, executed by system agent $\tau_{Ag}(PA)$; if a step is executed by Jolt, then τ_S maps it to a (not necessarily contiguous) subsequence of $\mathcal{T}$ steps,
5. τ_S is order preserving, i.e. if St_β precedes St_γ in $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$, then $\tau_S(St_\beta)$ precedes $\tau_S(St_\gamma)$ in $\mathcal{T}$,
6. for each step $St_\beta(v_{t-1}, j_t, p_t, v_t)$ in the domain of τ_S executed by a protocol agent *PA*, if $V_{PA,l}$ is the agent component of V modified during $St_\beta(v_{t-1}, j_t, p_t, v_t)$, then $W_{\tau_{ss}(PA,l)}$ is the system agent subspace modified during the step $\tau_S(St_\beta(v_{t-1}, j_t, p_t, v_t))$,
7. for each step $St_\beta(v_{t-1}, j_t, p_t, v_t)$ in the domain of τ_S executed by a protocol agent *PA*, if $\tau_S(St_\beta(v_{t-1}, j_t, p_t, v_t)) = Sy_{\tau_{Ag}(PA)}(w_{s-1}, k_s, q_s, w_s)$, and the transition $w_{s-1} \to w_s$ modifies system subspace W_e, then $W_e = W_{\tau_{ss}(PA,l)} \in W_{\tau_{Ag}(PA)}$ for some l (reiterating 6 above), $\tau_{PA,l}(v_{t-1}) = w_{s-1}$, $j_t = k_s$, $p_t = q_s$, $\tau_{PA,l}(v_t) = w_s$,
8. if Sy_ρ is a step of $\mathcal{T}$ in the image of τ_S that modifies a system state subspace W_z where $W_z = \tau_A(V_{PA,l})$ for some protocol agent *PA* and l, and Sy_σ is also a step of $\mathcal{T}$ in the image of τ_S that modifies W_z, then no step of $\mathcal{T}$ between Sy_ρ and Sy_σ may modify W_z unless it too is in the image of τ_S.

When we want to emphasise the details, we say that system run $\mathcal{T}$ instantiates *Protocol* via $\tau \equiv (\tau_{Ag}, \tau_A, \tau_S)$ at step $\tau_S(St_\alpha)$ of $\mathcal{T}$, where St_α is the initial step in $MPath_{\langle\alpha,\beta,\ldots,\gamma\rangle}$.

- Suppose a shift of an ATJ or CJ is represented at *Protocol* level by transforming *MPath* into *MPath*′, such that there is a corresponding transformation from $MPath^{\mathrm{DAG}}$ to $MPath'^{\mathrm{DAG}}$ in the DAG picture, and $MPath^{\mathrm{DAG}}$ and $MPath'^{\mathrm{DAG}}$ have a common suffix $MPath^{\mathrm{DAG}}(v^{\mathrm{con}}\ldots)$ after the local transformation. If $MPath(v^{\mathrm{con}}\ldots)$ and $MPath'(v^{\mathrm{con}}\ldots)$ are the corresponding suffixes in *MPath* and *MPath*′, we **assume** that $MPath(v^{\mathrm{con}}\ldots)$ can be instantiated within a system run $\mathcal{T}$ on a given subsequence η of $\mathcal{T}$ iff $MPath'(v^{\mathrm{con}}\ldots)$ can also be instantiated in $\mathcal{T}$ on η. (46)

Lemma 9.2. Suppose that system run $\mathcal{T}$ instantiates a maximal path *MPath* of *Protocol* via $\tau \equiv (\tau_{Ag}, \tau_A, \tau_S)$, so that a shiftable ATJ or CJ and an immediately preceding SENXOP or GNXOP are mapped by τ_S into an *adjacent* pair $\tau_S(\text{-}NXOP)$ and $\tau_S(\text{-}J)$ of *contiguous* subsequences of $\mathcal{T}$. Let *MPath*′ shift $\tau_S(\text{-}J)$ into the past in *Protocol*. Then $\tau_S(\text{-}NXOP)$ and $\tau_S(\text{-}J)$ can be interchanged in $\mathcal{T}$, to yield an instantiation of *MPath*′ in a system run $\mathcal{T}'$ via $\tau' \equiv (\tau'_{Ag}, \tau'_A, \tau'_S)$ constructed in the obvious way. Dually for an ATJ or CJ and an immediately following SENXIP or GNXIP. We say that $\tau_S(\text{-}J)$ has been shifted in $\mathcal{T}$ to yield $\mathcal{T}'$.

Proof. We have to show that $\mathcal{T}'$, as described, is a valid system run, and that it instantiates *MPath*′ as claimed. But this is easy. Since a SENXOP or GNXOP produces no outputs, pushing it into the future past an ATJ or CJ in *MPath*, does not threaten to demand that any input of the ATJ or CJ is consumed before the corresponding output has been produced. Also the instantiation τ merely instantiates any external I/O of SENXOP or GNXOP and of ATJ or CJ, and extends the protocol state by the remainder of the system state. Because of the contiguity and adjacency of $\tau_S(\text{-}NXOP)$ and $\tau_S(\text{-}J)$, this additional system state remains unchanged throughout $\tau_S(\text{-}NXOP)$ and $\tau_S(\text{-}J)$, and therefore interchanging $\tau_S(\text{-}NXOP)$ and $\tau_S(\text{-}J)$ results in a valid system run $\mathcal{T}'$, which evidently instantiates *MPath*′ via a τ', constructed in the obvious way, using (46). □

Definition 9.3. Suppose given a maximal path *MPath* in a jolt-normal protocol, and assume the notions of external dependency definition (XDD), and of input and output dependency sets (IDS, ODS) as in Definition 5.8. Then the protocol is 2PJXDD-normal iff the following holds: if *MPath* contains a real CP, then the CP, considered in isolation, is 2PXDD-normal according to Definition 5.8.

Definition 9.4. An instantiation of a 2PJXDD-normal protocol is called a (2PJXDD-normal) transaction.

For the remainder of this section all transactions will be 2PJXDD-normal.

Theorem 9.5. Let $\mathcal{T}_0$ be a run of a system which consists entirely of the steps of transactions of a family of resolvable 2PJXDD-normal protocols[16] such that:

1. each instantiation of a protocol path is partitioned into SENXIP, GNXIP, SENXOP, GNXOP, CJ, ATJ, SEJ pieces,
2. every step of $\mathcal{T}_0$ is in a SENXIP, GNXIP, SENXOP, GNXOP or CP of some transaction,
3. the steps of the instantiation of any SENXIP, GNXIP, SENXOP, GNXOP belong to a single transaction.

[16] So there is a set of maximal paths through a set of resolvable 2PXDD-normal protocols, and a set of instantiations of them in $\mathcal{T}_0$, and the set of steps of $\mathcal{T}_0$ is the (*not necessarily disjoint*) union of these instantiations.

Then there is a serialisation $\mathcal{T}_\infty$ of $\mathcal{T}_0$, generated by commuting adjacent steps and shifting instantiations of jolts and portions, in which each instantiation occurs as a contiguous series of steps, interrupted, at worst, by SEJs.

Proof. For each transaction in $\mathcal{T}_0$ choose a pivot as follows:

1. if the transaction instantiates a real CP, choose a pivot as in Claim 5.10.1,
2. if the transaction instantiates a deemed CP, choose the step whose before-state or after-state is the deemed CP.

Consider the directed graph Dep_0 whose nodes are the transactions of $\mathcal{T}_0$, and whose edges are given by: $\tau_1 \rightarrow \tau_2$ iff:

1. an output of an *ODS* step of τ_1 is an input of an *IDS* step of τ_2, or,
2. there is a system state subspace modified by both τ_1 and τ_2 and τ_1's pivot occurs earlier in $\mathcal{T}_0$ than τ_2's.

Claim 9.5.1 Dep_0 is acyclic.

Proof of Claim. As in the proof of Claim 5.10.1, we can interpret Dep_0 in the set of pivots. For clause 1, external I/O can be interpreted as arriving at or issuing from the pivot of a transaction, and for clause 2, well it refers directly to pivots anyway. Since all the edges of Dep_0, thus interpreted, are oriented towards the future, and the pivots are linearly ordered by time, the claim follows. □□

The remainder of the serialisation splits into three phases: *Portion Coalescence*, *Jolt Redefinition*, *Resolution*. In all cases we proceed in stages, each of which affects only a finite portion of the system run as in Theorem 5.10, following the structure of Dep_0 from its root nodes. Of course if $\mathcal{T}_0$ is infinite, we can never complete any of the phases before starting the next. In such cases we assume that the working of the various stages of the various phases is interleaved in such a way, that whenever some stage of some phase is performed, any preceding stages of any preceding phases that it depends on in order to see the context it expects, have already been done. Since each stage has only a finite 'reach,' this will always be possible. We do not dwell on the technical details of the scheduling needed to achieve this.

Portion Coalescence. Working on the transactions in an order compatible with Dep_0, we move steps of the system run until, for each portion of any transaction, the steps of the portion are contiguous. For a given transaction, we start with the CP, and move all its steps inwards towards the pivot. Thus the CP's last step before the pivot is swapped with its successor steps, in turn, until it arrives just before the pivot itself. The assumption that the CP is 2PXDD-normal and condition Definition 9.1.8 ensure that this process succeeds. Once all the pivot's CP predecessors come just before it, the pivot's CP successors are moved to just after it. Thus the CP's next step after the pivot is swapped with its predecessor steps, in turn, until it arrives just after the pivot, and similarly for the remaining steps of the CP.

There remain the SENXIP, GNXIP, SENXOP, GNXOP portions. For a SENXOP or GNXOP, all the portion's steps except the last one are moved into the future until they abut the last one in a contiguous order-preserving sequence. Once more, Definition 9.1.8 ensures that this works. For a SENXIP or GNXIP, all the portion's steps except the first one are moved into the past until they abut the first one in a contiguous order-preserving sequence. Definition 9.1.8 ensures that this works. We call the last steps of SENXOPs, GNXOPs, and the first steps of SENXIPs, GNXIPs, the subpivots of the transaction.

Jolt Redefinition. In the previous phase, due to movement of individual steps, the gap between two consecutive non-Jolt portions of a transaction may have acquired steps other than those originally matched to the intervening jolt. Jolt Redefinition is simply the process of redefining any such 'infiltrated' jolt between two portions to include *all* the steps that now fall between them, and to maintain this condition dynamically through the resolution phase, next.

Resolution. Since the preceding phases did not move any pivots or subpivots (relative to each other), the system run, as it now appears, is a sequence of non-overlapping transaction portions which embody the original dependencies of Dep_0. Moreover, each transaction's jolts now consist of some sequences of portions belonging to other transactions. Since we assumed that all the protocols figuring in the original system run were resolvable, we can now apply the resolution strategy of Definition 8.7 *et seq*. Sweeping through the transactions in an order compatible with Dep_0, we apply for each, a series of shifts which moves all its SENXIP, GNXIP, SENXOP, GNXOP portions until they (or whatever similar portions supersede them in the process) abut the transaction's CP (which is not moved throughout). The result is a sequentialisation of $\mathcal{T}_0$ in which only those SEJs that occur between the CP and its innermost surrounding SENXOP, GNXOP and SENXIP, GNXIP portions, survive to interrupt the otherwise completely sequential execution. We are done. □

Given that after the above serialisation, there are only SEJs embedded in the interior of any transaction, and that we have a global assumption that any state that they might observe is not used to violate serial semantics, it might be possible to argue that such SEJs can be replaced by true skips, and these may then be moved to outside the transaction in which they occur, achieveing a fully serial execution.

10. Examples of Serialisation of Resolvable Jolted Transactions

In this section we make a few remarks about some situations, drawn from our motivating examples, which embody some of the more elaborate serialisation techniques discussed above. We start with the simplest case, lock-free stacks.

10.1. Lock-Free Stacks

The straightforward lock-free stacks of Section 7.1 are relatively easy to interpret in the framework we built. The overall protocol DAG decomposes into a set of cases that describe the workings of clans of all possible sizes.[17] Above it, the agent forest summarises this for the sets of agents involved, and above that, the atomic shallow forest captures in a single transition, what each more detailed outcome accomplishes overall.

Thus for a clan of size n, at the atomic level, there will be n atomic transitions, each representing the fact that agent k succeeded in his update while the others failed, for $k \in \{0 \ldots n-1\}$. Descending to the agent level, the agent forest will contain, for a clan of size n, a subtree containing $n!$ branches emerging from a common root, and each branch will consist of a $Comm_k$ step representing the success of agent k, followed by $(n-1)$ more *Fail* steps representing the failure of the other agents, ordered in some way. Since there is no restriction on the order in which the $(n-1)$ other agents *Fail*, there will be $(n-1)!$ permutations of these, and all $n!$ branches gather into a tree with a branching factor that starts at n and decreases by 1 at each level, by identifying common prefixes. Note that we have not mentioned any Jolt steps in the agent transition relation. There are none; we do not need them.

At the protocol level there will be more complexity. Each maximal path will start with the $RdTop_l$ step for the agent l that initiates the clan (assuming we are not numbering agents in the order they join the clan, which always remains an option). This step is then followed by a series of *RdTop* and *Inter* steps representing the arrivals and local workings of other agents that take place prior to the $Comm_k$ step of the successful agent. (The only restriction on these *RdTop* and *Inter* steps, is that for any *Inter* step there must have been an earlier *RdTop* step.) At some point i this activity, agent k succeeds. Once agent k has succeeded, there are no more *RdTop* steps, and the maximal path is completed by whatever outstanding *Inter* steps and *Fail* steps there are for all agents other than k, these steps being interleaved in some causally valid order. Note that as for the agent transition relation, we have not mentioned any Jolt steps; there are none, we do not need them at protocol level either.

The models we have described at the various levels of abstraction are more or less fixed. Certainly that is the case for the atomic and protocol levels. The agent level potentially allows a little more flexibility, but given our understanding of the protocol, there is little scope for inventing an agent model that appears as 'natural' as the one we have sketched.

Having fixed the models, we can contemplate possible refinements between them. Again there are some 'obviously natural' refinements. As regards the atomic to agent refinement, all playouts of the protocol event structure feature success for one agent, and failure for the remainder, and the success always comes first. So an atomic to agent refinement that captures this in the most natural way is the most convincing: synchronising the atomic action with the successful agent's step (which always comes first) does the trick. This is a simple synchronisation assignment, as in Section 3. But of course, the refinement theory of Section 3 allows plenty of other possibilities, which we do not dwell upon here.

The agent to protocol refinement is constrained by the need to match the order of agent names appearing in any maximal path $MPath^A$ of the agent model, with the order of agent names that are executing the steps synchronised with them in the MSA for any corresponding maximal path $MPath^P$ in the protocol model that refines $MPath^A$. This is a modelling constraint rather than one forced by any faeture of the refinement theory. It is simplest and most transparent if we synchronise the protocol level success or failure event for agent ag in $MPath^P$ with the relevant agent level success or failure event for ag in $MPath^A$. Thus for any $MPath^P$, we have to choose the right agent level path $MPath^A$ for it to refine: the successful agent k of $MPath^P$ has to execute the agent level success event, the first step of $MPath^A$, and the remaining fail events of $MPath^A$ must be executed by the same ordering of agents as occurs for the failing events in $MPath^P$. This gives us a multiple synchronisation assignment as demanded by Definition 8.2. The absence of jolts at either agent or protocol level implies that we do not have to worry about the additional constraints in achieving a Jolt-compatible MSA as per Definition 8.3. Of course, there are numerous alternative possibilities for refinement, as permitted by the theory of Section 8, immeasurably amplified if we decompose the *Inter* steps into the low level sequences of local instructions that they represent for each agent and then consider all the possible valid interleavings that result.

[17] This will make the protocol DAG infinitely wide and it will have paths of arbitrary length, but all paths will still be finite.

Concerning the serialisation properties of the above protocol model and accompanying refinements, we note that aside from the additional structure imposed by taking note of agent names, we have not deviated from the framework of Section 5 at all. We have managed to design the description of lock-free stacks so that that any agent that impacts a clan becomes a member of it, and thus the the entire working of the clan becomes a finite piece of the overall protocol. In particular, the entire model enjoys the full 2P serialisability property.

10.2. Elimination Stacks

Let us now consider the extension of the lock-free stack to the elimination stack. One aspect of our earlier treatment, was the decoupling of the lock-free initial part of the more elaborate mechanism from the elimination part itself. One justification for this, and a perfectly adequate one, is that the complete protocol, as described in [CG07,CG06], features an unspecified and completely non-deterministic mechanism for assigning lock-free-failing agents to clan locations. This in turn allows subsequent refinement to mechanisms which are optimised to system-specific desiderata. For this reason we will continue to discuss the two parts separately.

In the elimination mechanism, the workings of a clan are delineated by the circumstance of the last successful joiner (i.e. the agent at the end of the chain) executing his *ElMe* event before any other agent has successfully joined the clan. Obviously clans of arbitrary size are possible, again leading to protocol DAGs of unbounded width and depth, but still with all paths finite.

At the atomic level, the single transitions will represent the various possible global outcomes: since the critical actions of the protocol are swaps, each involving two adjacent agents in the chain, overall, some even number of agents which happen to be adjacent in pairs in the chain and have complementary operations can succeed, the rest will fail; this will summarise the global outcome of the working of the clan.

At the agent level, the global outcome is reflected in individual agent steps, so for a clan of size n, there will be n steps in any agent level maximal path. In such an agent level maximal path, successful pairs have a *Push* step followed (either immediately or later) by the corresponding *Pop* steps. Failing agents have a *Fail* step. These step names may be made more elaborate, to record eg. who was the active and who the passive partner of a successful pair, or the mechanism by which a *Fail* came about. And not all naively conceivable interleavings of the steps need be possible since causality (as partly captured in the asymmetric features of the general event structures) must be respected. For instance, if two contiguous adjacent pairs both succeed, then the order of events must not be such that the middle pair of the four agents involved is forced to succeed. The possibilities depend, at least partly, on the level of detail recorded in the agent level steps. To make the remaining discussion more concrete, we will assume that we merely model, for a given agent, whether its step was *Push*, *Pop* or *Fail*, and that the state modelled at agent level just concerns the single local data item that was pushed or popped in a successful case, or that was to be pushed or popped in an unsuccessful case. As for the lock-free stack, we have no need for Jolt steps at agent level.

At the protocol level, we have steps corresponding to the events in (a finite subset, corresponding to a finite number of clan members, of) Fig. 7, again interleaved in a large number of possible ways, commensurate with causality. In addition, we have the option of including events corresponding to those concurrent attempts to join the clan while the possibility to do so is still open, but that fail, events (whose successful couterparts are the *GetHim* events of Fig. 7) which are not represented in Fig. 7. Regarding these, if we assume weak fairness, then for any finite size of clan, there will be a finite (but potentially unbounded) set of them.

Rather than clutter the protocol definition itself by including these failed joining attempts as the acts of agents involved in the protocol, we can conveniently represent them as SEJs interleaved into the paths of the protocol. This makes the agents and steps of the protocol transition relation correspond to the events in Fig. 7, with the interjection of finite numbers of SEJs into the protocol's paths. This is economical in terms of representing the important aspects of the protocol. Moreover, since failing clan joiners do not affect their subsequent behaviour on the basis of the information they gain, the higher level atomicity invariant is maintained, as is appropriate for modelling such interruptions as SEJs. So at the protocol level, we do have jolts, but only of the simplest kind, those that are refinements of the identity at the agent level.

Concerning refinement, since there is no common feature of all global outcomes to fasten onto (since, *in extremis* all agents of the clan can *Fail*, eg. by executing their $\overline{SwP}$ events in numerical order), we have no 'natural' refinement between abstract and agent levels to focus on. Any of the many refinements permitted by Section 3 is as good as any other. Each simply makes the single atomic level step correspond to one of the agent level steps in an agent level maximal path that refines it.

For the agent level to protocol level refinement we can do a little better, in that the presence of agent names at both levels of abstraction and the necessity to match the order of agent names in a maximal agent level path with the order

of agent names occurring in the MSA for any maximal protocol level path that refines it enables us to construct a more 'natural' refinement. Thus we can synchronise the sequence of success or failure events at protocol level, as they occur along a maximal path for the relevant listing of the protocol agent names, with the agent level sequence of abstract agent successes or failures for the same ordering of protocol agents. But, as ever, the extension of the refinement theory of Section 3 potentially offers many more possibilities.

Concerning serialisation, we have already noted the availability of SEJs to succinctly record innocuous interference in the protocol state by agents extraneous to the protocol (i.e. unsuccessful clan joiners). Since these SEJs consist of reads and CAS operations, they reveal information (namely the identities if the last two successful joiners) that could, in principle, be used to destroy serial semantics. However the discipline adhered to by protocol agents is such that no such violation occurs, and these extraneous accesses to the state in the middle of a protocol run, indeed prove to be innocuous.[18]

Noting this, we can regard the semantics of the protocol as 'near serial,' with the only departures from 'true seriality' being these SEJs. Whether near seriality can in fact be converted to true seriality hinges on a small modelling detail. Is the sematics of joining a clan a non-determinstc choice between success and failure, or is it specified so that success always has priority over failure (as modelled in Fig. 7)? If the former, then we always have the option of moving an SEJ to the beginning or end of the transaction it interrupts, replacing it there by an event non-deterministcally chosen to be null. However, if success always has priority over failure, then at the beginning or end of the transaction, it may not be possible to find a place for a null event, since the prevailing circumstances there may not be capable of preventing success. The account in [CG07, CG06] inclines somewhat to the former view.

As a final word on these two examples, we should not forget that in reality, these protocols run as sequences of individual machine instructions, the critical parts being handled by CASs. The number of possible interleavings that this generates is truly enormous; much greater than even the number of interleavings that we noted existed at protocol level.

10.3. Non-2P Fragments of Mondex

The previous examples showed no, or almost no, departure from the straightforward refinement and serialisation proposals described in Sections 3 and 5. Even the failed clan joiners of the elimination stack could be accomodated, if desired, in the standard framework, at the cost of a fair amount of clutter. The situation with the non-2P fragments of Mondex is rather different, since the 2P discipline is palpably broken from the outset, and requires the stretching of our theory as described in Sections 8 and 9. In the ensuing, will describe one particular non-2P scenario, and how it is dealt with, in detail. Other related possibilities, such as are indicated in Section 6.7, can be dealt with in a similar manner.

To appreciate the nature of the problem properly, we will need more detail about the protocol level of Mondex than we have presented hitherto. We will introduce what we need as we go.

At the protocol level of Mondex, consider two purses, the From purse and the To purse, as before. The local state we need to consider for each purse involves its balance, and its sequence number. (There are of course other components of local state present, such as the prospective transaction amount, the purses' unique identifiers, and the purses' local logs, not to mention the local state of every other purse in the Mondex community and the global 'aborted transaction archive,' but they will not affect our discussion so we can ignore them.) So, as far as we are concerned, the state will consist of $FromSN, FromBal, ToBal, ToSN$.

Consider the non-2P run of steps in the fragment below. Particular state values are refered to by the series of superscripts on $FromSN, FromBal, ToBal, ToSN$ on various lines. In between citing the state values, various purse operations are mentioned. The From purse's steps occur towards the left, and the To purse's steps occur towards the right.

In Mondex, the environment correlates its view of a transaction which is about to start, with the two protagonist purses' internal view of it, by parameterising the two *Start* operations with (what should be) the two purses' sequence numbers at that point (information which the environment can obtain by simply asking the purses). For the fragment below, let us assume these sequence numbers are $FromSN^{A}$ and $ToSN^{A}$, as in the first state.

[18] How unlike quantum mechnics then, in which the mere presence of the means to reveal information is enough to alter the physical state.

$$
\begin{array}{l}
\mathit{FromSN}^{\mathrm{A}}, \mathit{FromBal}^{\mathrm{A}}, \mathit{ToBal}^{\mathrm{A}}, \mathit{ToSN}^{\mathrm{A}} \\
\quad \mathit{StartFrom} \\
\quad \mathit{AbortFrom} \\
\mathit{FromSN}^{\mathrm{A}'}, \mathit{FromBal}^{\mathrm{A}}, \mathit{ToBal}^{\mathrm{A}}, \mathit{ToSN}^{\mathrm{A}} \\
\qquad \ldots \text{ other stuff } \ldots \\
\mathit{FromSN}^{\mathrm{B}}, \mathit{FromBal}^{\mathrm{B}}, \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}} \\
\qquad\qquad\qquad\qquad \mathit{StartTo}\ (*) \\
\qquad\qquad\qquad\qquad \mathit{AbortTo} \\
\mathit{FromSN}^{\mathrm{B}}, \mathit{FromBal}^{\mathrm{B}}, \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}'}
\end{array}
$$

The first pair of steps, *StartFrom* and *AbortFrom*, does not alter the balance, but the *StartFrom* operation does increment the From purse's sequence number (in the hope that the transaction turns out to be a useful one, needing therefore to be isolated from any future activity), moving it from $\mathit{FromSN}^{\mathrm{A}}$ to $\mathit{FromSN}^{\mathrm{A}'}$. The 'other stuff' represents other unspecified transactions that involve the From purse and perhaps the To purse too, and moves the local state to $\mathit{FromSN}^{\mathrm{B}}, \mathit{FromBal}^{\mathrm{B}}, \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}}$. The story now depends on whether 'other stuff' really did involve the To purse or not. If not, then $\mathit{ToBal}^{\mathrm{A}}, \mathit{ToSN}^{\mathrm{A}} = \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}}$ and the *StartTo* step can run since it is parameterised by the correct To purse sequence number, namely $\mathit{ToSN}^{\mathrm{A}}$. If yes, then the *StartTo* step cannot run, and only the *Abort* step can run. This is indicated by the asterisk against the *StartTo* step.

To serialise this non-2P scenario, one possibility to consider is to move the *StartFrom* and *AbortFrom* pair, so that it occurs just before the *StartTo* $(*)$ and *AbortTo* pair. Unfortunately, the sequence number for the From purse will be wrong by this point. However we can replace the earlier *StartFrom* and *AbortFrom* combination by a later *AbortFrom* step alone (as we argued for the *StartTo* $(*)$ step). This is doubly fortunate, since an *AbortFrom* step alone will not increase the From purse sequence number, and thus cause no disagreement with the $\mathit{FromSN}^{\mathrm{B}}$ at the end of the run, whereas a successful later *StartFrom* would give us this additional headache to contend with.[19]

But this is not the worst of it. Removing the *StartFrom* and *AbortFrom* pair from its earlier position leaves an inconsistency in the From purse sequence number, which now has no means of progressing from $\mathit{FromSN}^{\mathrm{A}}$ to $\mathit{FromSN}^{\mathrm{A}'}$. Fortunately the protocol level of Mondex provides an operation *Increase* whose only effect is to increase the current purse's sequence number (at this level of abstraction). So we can plug the gap with an *IncreaseFrom* step, arriving at the run below.

$$
\begin{array}{l}
\mathit{FromSN}^{\mathrm{A}}, \mathit{FromBal}^{\mathrm{A}}, \mathit{ToBal}^{\mathrm{A}}, \mathit{ToSN}^{\mathrm{A}} \\
\quad \mathit{IncreaseFrom} \\
\mathit{FromSN}^{\mathrm{A}'}, \mathit{FromBal}^{\mathrm{A}}, \mathit{ToBal}^{\mathrm{A}}, \mathit{ToSN}^{\mathrm{A}} \\
\qquad \ldots \text{ other stuff } \ldots \\
\mathit{FromSN}^{\mathrm{B}}, \mathit{FromBal}^{\mathrm{B}}, \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}} \\
\quad \mathit{AbortFrom} \\
\qquad\qquad\qquad\qquad \mathit{StartTo}\ (*) \\
\qquad\qquad\qquad\qquad \mathit{AbortTo} \\
\mathit{FromSN}^{\mathrm{B}}, \mathit{FromBal}^{\mathrm{B}}, \mathit{ToBal}^{\mathrm{B}}, \mathit{ToSN}^{\mathrm{B}'}
\end{array}
$$

Let us now discuss how the above transformation can be viewed as the shifting of a jolt in the serialisation formalism of Section 9.

To start with, since neither *StartFrom* ; *AbortFrom* nor *StartTo* $(*)$; *AbortTo* does any I/O, we can view the first as a GNXOP and the second as a GNXIP. Since we serialised to *StartTo* $(*)$; *AbortTo*, the before-state of its its first step makes for an appropriate deemed CP. The 'other stuff' constitutes a jolt, and since the purse balances can change during it, something that will be observable at the agent level, it constitutes a CJ. Clearly the replacement of *StartFrom* ; *AbortFrom* by an *AbortFrom* working on a different state later, is not the mere displacement of a step past other steps that are independent of it, so we must justify the move on requirements grounds. However, both portions are null transactions as far as the abstract and agent levels are concerned, so we judge the replacement appropriate, satisfying the higher level invariant that demands that whatever we alter, the sequence of non-trivial transactions that take place remains the same.

Finally, the shift that implements the replacement, replaces the GNXOP ; CJ sequence by a CJ ; GNXOP′ sequence which is itself prefixed by the *IncreaseFrom*. Despite the fact that the *IncreaseFrom* is executed by the same From

[19] Besides this, the balance of the From purse might have been depleted during the 'other stuff,' to an extent that makes starting the previously scheduled transaction properly at the later time, impossible.

purse that executed the GNXOP ; CJ sequence, the *IncreaseFrom* has nothing to do with the current transaction, so, from the point of view of the current transaction, it is a jolt that occurs outside of it. In fact it is an example of the expulsion of a 'remaining piece' of GNXOP from the current transaction, as sketched at the end of Section 8, and it is perfectly reasonable to view it as a fresh trivial transaction in its own right. Thus the current transaction has been reduced to the *AbortFrom* ; *StartTo* $(*)$; *AbortTo* sequence, a contiguous sequence, as was our aim.

11. Mechanical Verification

To gain assurance in the relatively informal account of protocol theory given above, some mechanical verification has been undertaken, using the KIV theorem prover. As well as supporting the preceding theory, this constitutes an interesting exercise in formal verification in its own right.

KIV [RSSB98, KIV] is an interactive theorem prover for many-sorted many-sorted higher-order logic. There are several extensions to this logic (Dynamic Logic, Temporal Logic and a logic for Java programs), but they are not used here. Structured algebraic specifications can be built from elementary theories using the standard operators (similar to CASL [CoF04]): union, enrichment, renaming and actualization of parametric specifications. Theorem proving uses sequent calculus.

As a first step towards a formalized theory of protocols, KIV specifications and proofs have been developed for the isolated protocols of Section 3. The results are available on the Web [KIV07]. Checking theorems with KIV led to small improvements which are already incorporated in Section 3, so in this section we only discuss a few topics, which are relevant when transferring pencil-and-paper proofs to an interactive theorem prover, and we give a lemma used in Theorem 3.8, that shows a modularization of the proof.

When formalizing the notion of execution paths a first difficulty is of course that no 'three dots notation' is available in formal specifications. Instead a free data has been defined in KIV. Using Z notation this data type can be written as:

$$path ::= mkV\langle\langle V\rangle\rangle \mid mkpa\langle\langle V \times J \times P \times path\rangle\rangle \tag{47}$$

A number of operations are needed for paths. $\#pa$ is the number of steps of path *pa*, its nth node is $pa[n]$ for $0 \leq n \leq \#pa$, and its first and last nodes are $pa.first := pa[0]$ and $pa.last := pa[\#pa]$. The concatenation $pa + pa'$ of two paths *pa* and pa' is defined when $pa'.first = pa.last$. We also need the first *n* steps *pa to n* (written infix) of a path, and the rest *pa from n*. $inputs(pa)$ and $outputs(pa)$ are the inputs resp. outputs done on a path. Finally, $Step(pa, n) \in V \times J \times P \times V$ is the *n*'th step of *pa*. A predicate $Path(pa)$ is defined recursively, which holds, iff every step satisfies some $St(\rho)(Step(pa, n))$. An argument ρ from some index type *CIx* replaces the subscript in St_ρ; the (higher-order) type of *St* being:

$$St : CIx \to V \times J \times P \times V \to bool \tag{48}$$

To give formal definitions of *FPath* (3), *BPath* and *MPath*, two unspecified predicates *init* and *final* characterizing initial and final states are used. Around 40 lemmas are proved over this theory and used as rewrite rules to get some basic automation for the main proofs.

The definition of protocol (cf. (6)) becomes:

$$Protocol(v, js, ps, v') == \exists\, pa \bullet MPath(pa) \wedge inputs(pa) = js \wedge outputs(pa) = ps \tag{49}$$

A synchronization assignment is defined as a function $SA : path \to nat$. The idea is that the synchronization step of a path is $Step(pa, SA(pa))$. Function *SA* is specified by two constraints:

$$MPath(pa) \Rightarrow SA(pa) < \#pa \tag{50}$$

$$MPath(pa) \wedge MPath(pa') \wedge n \leq \#pa \wedge m \leq \#pa' \wedge pa[n] = pa'[m] \Rightarrow (SA(pa) < n \Leftrightarrow SA(pa') < m) \tag{51}$$

The first axiom should be obvious, the second is a consistency condition: for two maximal paths, which have a state in common, the synchronization point must either be before that node in both paths, or both synchronization steps must follow the common node. Based on this definition we can characterize the steps of a maximal path to be the disjoint union of FS, BS and SA steps. As an example, the *n*'th step of path *pa* is a forward skip step iff $FS(pa, n)$ holds:

$$FS(pa, n) == MPath(pa) \wedge (n < SA(pa) \vee SA(pa) < n < \#pa \wedge OD(pa\ to\ SA(pa)) \tag{52}$$

where

$$OD(pa) == FPath(pa) \wedge \forall\, pa1, pa2 \bullet MPath(pa + pa1) \wedge MPath(pa + pa2) \Rightarrow pa1.last = pa2.last \tag{53}$$

As Lemmas for Theorem 3.8 and Corollary 3.9 we then prove that all steps can be simulated forwards and backwards, the only exception being BS steps, which can only be simulated backwards:

$$\text{BS-BW} : \mathit{MPath}(pa) \wedge \mathit{BS}(pa,n) \wedge R^1(u,v') \wedge pa[n] = v \wedge pa[n+1] = v' \Rightarrow R^1(u,v) \tag{54}$$

$$\text{FS-FW} : \mathit{FS}(pa,n) \wedge R^1(u,v) \wedge pa[n] = v \wedge pa[n+1] = v' \Rightarrow R^1(u,v') \tag{55}$$

$$\begin{aligned}\text{SA-FW} : &\mathit{MPath}(pa) \wedge R^1(u,v) \wedge \mathit{Step}(pa,\mathit{SA}(pa)) = (v,j,p,v') \\ &\Rightarrow \exists\, u',i,o,k \bullet \mathit{At}(k)(u,i,o,u') \wedge R^1(u',v') \wedge \mathit{Input}1(i,j) \wedge \mathit{Output}1(o,p)\end{aligned} \tag{56}$$

$$\begin{aligned}\text{SA-BW} : &\mathit{MPath}(pa) \wedge R^1(u',v') \wedge \mathit{Step}(pa,\mathit{SA}(pa)) = (v,j,p,v') \\ &\Rightarrow \exists\, u,i,o,k.\mathit{At}(k)(u,i,o,u') \wedge R^1(u,v) \wedge \mathit{Input}1(i,j) \wedge \mathit{Output}1(o,p)\end{aligned} \tag{57}$$

$$\text{FS-BW} : \mathit{MPath}(pa) \wedge \mathit{FS}(pa,n) \wedge R^1(u,v') \wedge pa[n] = v \wedge pa[n+1] = v' \Rightarrow R^1(u,v) \tag{58}$$

$$\text{BS-BW} : \mathit{MPath}(pa) \wedge R^1(u,v') \wedge pa[n] = v \wedge pa[n+1] = v' \Rightarrow R^1(u,v) \tag{59}$$

The proof of the last two lemmas requires Ass. 3.2.2, the others do not. The lemmas are independent of Ass. 3.1.3 which require all concrete states to be reachable. Based on the characterization of steps on paths, we can now define a global characterization of steps:

$$\mathit{BS}(v,j,p,v') == \exists\, pa,n \bullet \mathit{BS}(pa,n) \wedge \mathit{Step}(pa,n) = (v,j,p,v') \tag{60}$$

$$\mathit{FS}(v,j,p,v') == \neg \mathit{BS}(v,j,p,v') \wedge \exists\, pa,n \bullet \mathit{FS}(pa,n) \wedge \mathit{Step}(pa,n) = (v,j,p,v') \tag{61}$$

$$\mathit{SA}(v,j,p,v') == \exists\, pa.\mathit{MPath}(pa) \wedge \mathit{Step}(pa,\mathit{SA}(pa)) = (v,j,p,v') \tag{62}$$

Note that a step which is an FS step on one path and a BS step on another, must be classified as a BS step, since it is the successor of an OD step on *some* path. The three classes of steps are proven to be disjoint, and provided all states are reachable every step $\mathit{St}(\rho)(v,j,p,v')$ falls into one of the three classes. This allows us to prove Theorem 3.8 formally. As an example, the definition of Clause 4 of Theorem 3.8 is proven formally as:

$$R^1(u',v') \wedge \mathit{BS}(v,j,p,v') \Rightarrow R^1(u',v) \tag{63}$$

using Lemma (59). The four clauses (58), (57) and (59) together imply Corollary 3.9. We also prove that forward simulation is always possible by choosing the synchronization step as the last step of every maximal path:

$$\begin{aligned}&(\forall\, pa \bullet \mathit{MPath}(pa) \Rightarrow \mathit{SA}(pa) = \#pa - 1) \wedge \mathit{St}(\rho)(v,j,p,v') \wedge R^1(u,v) \\ &\quad \Rightarrow \exists\, u' \bullet R^1(u',v') \wedge (u = u' \vee \exists\, i,o \bullet \mathit{Atomic}(u,i,o,u'))\end{aligned} \tag{64}$$

The KIV proofs for the theorems of Section 3 are relatively small compared to other KIV case studies (eg. the Mondex case study [SGHR06b, SGH+07] already mentioned). The tricky bit about them is mainly to get all the assumptions right for all the cases. As an example, the borderline case of an *MPath* consisting of a single node must be forbidden, since then constraint (50) is not satisfiable.

12. Conclusions, and Further Work

In the preceding sections we took the Mondex Electronic Purse —a prime example of a protocol enacted between a number of parties that was designed to achieve the effect of an atomic action— and we looked for a generalisation. We developed a refinement framework based on seeing the atomic action as a shallow computation tree and the protocol as a computation DAG, and saw that we could choose the way that the atomic action was synchronised with the protocol in a 'small diagram' refinement relatively freely. The properties of the choice, in particular how potential abstract outcomes were related to synchronisation points, was closely related to the prospects for forward and backward simulation at the small diagram level.

We then embedded this formulation of an isolated protocol run in a framework enabling different runs of perhaps different protocols to be interleaved in a natural way. When combined with a fairly standard 2-phase property, these system runs could be serialised, showing that the atomicity abstraction survives.

We then confronted the theory with various refinements for Mondex that have been created in the recent past, and showed that the flexibility regarding synchronisation points was well borne out in these various refinements.

However, although the majority of 'normal' Mondex transactions (including not only successful ones, but also ones that fail in a 'normal' kind of way) are 2-phase —and the modification of the protocol suggested by Schellhorn

et al. in [SGH+07] in order to design out the possibility of a certain kind of denial of service attack is 2-phase in its entirety— the original Mondex protocol has some (in practice rare, but in theory interesting) non-2-phase parts. A more sophisticated theory was required to handle those situations.

Besides these issues, Mondex is what we called an isolated protocol. That is to say, once the protocol has commenced, the parties engaging in it are fixed, and no intrusion by other agents is contemplated. (In practice, the Mondex purse's local state determines how much notice is taken of which messages from which agents.) Thus it is natural to ask how the theory develops for protocols having state that is genuinely shared between a number of agents, including cases where the number is not necessarily determined at the start of the protocol.

To this end, we examined an example of a not-so-isolated protocol, the lock-free stack and its more elaborate relative, the elimination stack. These are protocols that are designed to run at a very fine level of granularity and interleaving, precluding the use of any kind of powerful locking mechanism to enforce atomic semantics. The lock-free and elimination stacks were introduced via an elaborated notion of event structure, the straightforward version of which supported various aspects of the discussion of the Mondex protocol.

In the not-so-isolated world, the relatively straightforward theory that catered for the Mondex protocol, becomes more cluttered due to the necessity of credibly reflecting the structure of the various participating agents and their different interactions, an aspect that could be safely downplayed in the isolated protocol case because of its statically determined structure. These aspects do not really interfere with the main ideas of the refinement proofs since those were already available in their absence.

However the inclusion of agent structure at this level permits such things as the direct modelling of more sophisticated behaviour by the I/O environment than we have contemplated in this paper. (To capture, using the techniques of this paper, I/O behaviour more subtle than the simple delivery of messages injected into the environment, one would have to regard the environment as an agent in its own right, participating in an esential way in protocols.)

The inclusion of agent structure at this level also permits the formulation of the notion of jolted protocol: a protocol whose normal running could be interrupted by 'rogue agents' not regarded as belonging to the current family of agents executing the protocol. Such flexibility is useful to enable a tangle of agents, potentially unbounded in time and space, all interfering with one another, each in pursuit of his individual protocol aims, to be separated into finite pieces in order to apply our theory.

With the jolted protocol ideas in place, the instantiation of such protocols in system runs could be considered, and thence their serialisation. We saw that provided enough local transformations of the running of a protocol could be found, interchanges of portions of protocols larger than the single steps usually moved around during serialisation arguments, enabled system runs that departed rather more drastically from the 2-phase property to nevertheless be serialised.

We then revisited our running examples. The lock-free and elimination stacks yielded rather easily to the new techniques, requiring little departure from the earlier formulation in fact. The aforementioned non-2-phase aspects of Mondex did however need the full power of the new theory, and we illustrated this by discussing one example in detail.

It is inevitably the case that the mechanical verification of the entire theoretical framework presented in this paper would be a considerable undertaking. As such it has not been attempted yet. However, the experience in verifying formulations of refinement gained by the second author and colleagues, enabled the results of Section 3 to be formalised and mechanically proved using KIV without the investment of excessive effort. The fuller mechanisation of the techniques we have presented remains an intriguing challenge for the future.

References

[AH06] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 21, 2006.

[B4f] Clearsy. b4free tool home page. `www.b4free.com`.

[BC88] G. Boudol and I. Castellani. Concurrency and Atomicity. *Journal of Theoretical Computer Science*, 59:25–84, 1988.

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

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

[Bou90] G. Boudol. Flow Event Structures and Flow Nets. In Guessarian, editor, *Semantics and Systems of Concurrent Processes, Proc. LITP-90*, pages 62–95. Springer LNCS 469, 1990.

[BPJS07] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Retrenching the Purse: The Balance Enquiry Quandary, and Generalised and (1,1) Forward Refinements. *Fund. Inf.*, 77:29–69, 2007.

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

[CG06] R. Covin and L. Groves. Verification of a Scalable Lock-Free Stack Algorithm. Technical Report CS-TR-06-14, Victoria University of Wellington, 2006.

[CG07] R. Covin and L. Groves. A Scalable Lock-Free Stack Algorithm and its Verification. In Hinchey, M. and Margaria, T., editor, *Proc. SEFM-07*, pages 339–348. IEEE Computer Society Press, 2007.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

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

[DoTaI91] Department of Trade and Industry. Information Technology Security Evaluation Criteria, 1991. `http://www.cesg.gov.uk/site/iacs/itsec/media/formal-docs/Itsec.pdf`.

[DW03] J. Derrick and H. Wehrheim. Using Coupled Simulations in Non-atomic Refinement. In D. Bert, J. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, pages 127–147. Springer LNCS 2651, 2003.

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

[Gur95] Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford Univ. Press, 1995.

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

[HSY04] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *Proc. SPAA-04*, pages 206–215. ACM Press, 2004.

[Int05] International Standards Organisation. Common criteria for information security evaluation, 2005. ISO 15408, v. 3.0 rev. 2.

[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`.

[JWe07] C. Jones and J. Woodcock (eds.). (title to be confirmed). *Formal Aspects of Computing*, 2007. (to be published).

[KIV] *The Karlsruhe Interactive Verifier*. `http://i11www.iti.uni- karlsruhe.de/~kiv/KIV-KA.html`.

[KIV06] Web presentation of the Mondex case study in KIV, 2006. URL: `http://www.informatik.uni-augsburg.de/swt/projects/mondex.html`.

[KIV07] Web presentation of isolated protocol refinement in KIV, 2007. URL: `http://www.informatik.uni-augsburg.de/swt/projects/Refinement/protocolrefine.html`.

[LV93] N. A. Lynch and F. W. Vaandrager. Forward and Backward Simulations — Part I: Untimed Systems. Technical Report CS-R9313, C. W. I., 1993.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains. *Journal of Theoretical Computer Science*, 13:85–108, 1981.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Automated Deduction - CADE-11. Proceedings*, LNAI 607, pages 748–752, Berlin, 1992. Saratoga Springs, NY, USA, Springer.

[PP95] G. Pinna and A. Poigne. On the nature of Events: Another Perspective in Concurrency. *Journal of Theoretical Computer Science*, 183:425–454, 1995.

[RSSB98] Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured Specifications and Interactive Proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, chapter 1: Interactive Theorem Proving, pages 13–39. Kluwer Academic Publishers, Dordrecht, 1998.

[Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. URL: `http://www.jucs.org`.

[Sch05] G. Schellhorn. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. *Journal of Theoretical Computer Science*, vol. 336, no. 2-3:403–435, May 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 U. Glsser J.-R. Abrial, editor, *Proceedings of the Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, LNCS. Springer, 2007. (submitted, extended version available as [SGHR06a]).

[SGHR06a] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. Technical Report 27, Universität Augsburg, Fak. für Informatik, 2006. available at [KIV06].

[SGHR06b] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proc. FM 2006*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, second edition, 1992.

[The92] The RAISE Language Group. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall, 1992.

[Win86] G. Winskel. Event Structures. In Brauer and Reisig and Rozenberg, editor, *Advances in Petri Nets*, pages 325–392. Springer LNCS 255, 1986.

[Win88] G. Winskel. An Introduction to Event Structures. In de Bakker, de Roever, and Rozenberg, editors, *Proc. REX Workshop*, pages 364–397. Springer LNCS 354, 1988.

[WN95] G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science Volume 4 Semantic Modelling*, pages 1–148. Oxford Univ. Press, 1995.

[WV02] G. Weikum and G. Vossen. *Transaction Processing*. Morgan Kaufmann, 2002.