Rationally Reconstructing the Escrow Example

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

Sophia Drossopoulou
Imperial College, London
scd@doc.ic.ac.uk

Abstract
The Escrow Exchange Contract has been used as a case study of building up complex and trustworthy systems from basic object capabilities, in the context of concurrent and distributed programming. In this short paper we present a Rational Reconstruction of the Escrow Exchange Contract case study, expressed in Grace, concentrating on the most essential issues of trustworthiness, and ignoring issues to do with distribution or more complex protocols. We then use our notation for capability policies to specify the key features of the reconstructed case study.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; D.4.6 [Security and Protection]: Verification; K.4.4 [Electronic Commerce]: Payment Schemes, Security

1. Introduction
Miller, Cutsem, and Tulloh [10]’s Escrow Exchange Contract case study aims to demonstrate how “contracts can be specified elegantly and executed safely, given an appropriate distributed, secure, persistent, and ubiquitous computational fabric”. An escrow exchange contract is a trusted third party that guarantees that two other mutually untrusting services can exchange rights with one another — for example, trading an agreed number of shares in a company for an agreed amount of a virtual currency. For reasons both political and technical, Miller, Cutsem, and Tulloh [10]’s design does not assume a universal clearing service, a government or supranational agency, laws of contract or tort, nor their corresponding components of a software architecture such as a shared database or a trusted transaction service. Rather, the entire mechanism of contracts is built from the bottom up based on object capabilities. The Exchange Escrow Contract case study is important precisely because it is useful to gauge how well object-capability systems in fact support building up larger scale structures of cooperation and trust, even between mutually untrusting counterparties.

One of Miller et al. [10]’s goals for their case study is that “non-experts should be able to write smart contracts understandable by other non-experts.” While the design of the case study is indeed clear and elegant, the code unavoidably tangles a number of cross-cutting concerns (especially concurrency and distribution) with the code that manipulates the object-capabilities. To give a little of the chthonic flavour of the code from Miller et al. [10], the lines

```java
var makeEscrowPurseP = Q.join(srcPurseP ! makePurse, dstPurseP ! makePurse);
```

test whether the srcPurse and dstPurse objects belong to the same mint, in part by an asynchronous object identity test on the lambda closure object stored in each purse’s makePurse field. In the version in this paper, we don’t need this check, but if we did, it would be written something like: “srcPurse.mint == dstPurse.mint.”

In our previous work [4], we analysed Miller’s Mint and Purse example [8] by expressing it in Joe, a Java subset without reflection and static fields, and discussed the six capability policies that characterise the correct behaviour of the program, as proposed in [8]. We argued that these policies require a novel approach to specification, and showed some first ideas on how to use temporal logic. In our yet unpublished technical report [5], we propose a specification language, and use it to fully specify the six policies from [8]; however, their formalization showed that they allowed several possible interpretations. We also uncovered the need for another four policies and formalized them as well.

Contributions
In this paper we provide a rational reconstruction [2] of the Escrow case study into Grace [1]. Our version concentrates on the main features of the case study: the fact that a buyer and seller who do not trust each other may safely agree on a (trusted) contract, and expect, when they fulfil their side of the obligation, the contract to go through, or to receive a reimbursement. Moreover each party may back out of a contract that has not yet been completed. We do not directly address issues of synchronization, persistence, or distribution here; rather we assume an effectively object-synchronous programming model (either with actors, fully-synchronized Java-style objects, or a single threaded event loop) where underlying middleware deals with all issues of communications failure, replication, etc. We propose capability policies crucial to the correct operation of the translated case study, and express them in our policy notation [5].

Outline of paper and Structure of our Solution
We first give an outline of our solution. We have a class mint which creates, and keeps in a ledger, objects of type Purs, and a singleton object escrowAgent, which creates and keeps track of objects of class contract.

Figure 1. The type Purse
The mint was first proposed in [9] together with six capability policies. In [5] we propose how to formalize these policies. In this paper, we formulate and formalize some of the policies for the more complex escrowAgent and contract objects. For this, we first outline concepts for the specification of capability policies.

### 3. Concepts for Capability Policy Specification

In this section we briefly sketch the most crucial concepts needed for capability specifications. The concepts are essentially observations of program execution. With the exception of execution histories, introduced at the end of the section, all these concepts have been defined for a subset of Java in [5] - their definition can be easily adapted to Grace.

**Modules and Linking.** To model the open nature of capability policies, we need to describe both the program we are checking, and potential extensions of that program (through subclasses, mashups, imports etc). For this we use modules, M, to denote programs, and * to describe the combination of two programs into one larger program.

The * operator links modules together into new modules. Linking performs compatibility checks, and therefore * is only partially defined. For example, because the method balance is confidential, \( M_{mint} \ast M' \) would be undefined, if \( M' \) contained the expression \( m.newPurse(\_).balance \) where \( m \) was an object of class mint.

**Runtime Configurations and Expression Evaluation.** Execution takes place in the context of runtime configurations \( \kappa \). A configuration is a stack frame and heap. A stack frame is a tuple consisting of the following four components: the address of the receiver, a mapping giving values to the formal parameters, the class identifier, and the method identifier of the method being executed. A heap is a mapping from object addresses to objects.

```gcode
class mint.new(name': String) -> Mint {
    method ledger = col.map.new // maps Purse to Numbers
    method name ->> String (name')
    method newPurse(name': String, amount: Number) -> Purse {
        def p = object { // makes a new object every time
            method hashcode { name'.hashcode }
            method balance is confidential (ledger.get(self))
            method name -> String ("{name'}@{name'} {balance}"")
            method deposit(amount: Number, from: Purse) -> Done
                ledger.put(p, amount)
            return p
        }
        method deposit(to: Purse, amount: Number, from: Purse) -> Done
            if ((amount > 0) && ((ledger.get(from) - amount) >= 0) && (ledger.contains(to)))
                then {
                    ledger.put(from, ledger.get(from) - amount)
                    ledger.put(to, ledger.get(to) + amount)
                } else {
                    Error.raise "DEPOSIT FAILURE"
            }
        }
    }
    mint.deposit amt, src
    mint.newPurse(name'' : String, name'' : String, Ledger = col.map.new
        ledger.put(p, amount)
    }
}
```

Figure 2. The Mint expressed in Grace.
Execution of a code snippet, code for a module $M$ takes a configuration $\kappa$ and returns a value $v$ and a new configuration $\kappa'$. We assume large step semantics of shape $M, \kappa, \text{code} \rightsquigarrow \kappa', v$.

**Paths** We support path expressions $p$ (i.e., expressions which only involve field reads). For example, mint, and $\text{pr}.\text{mint}$ are paths. Paths are interpreted in the context of runtime configurations, $\rho$. So that $[p]_{\kappa} = v$ if $p$ is a path and $\emptyset, \kappa, p \rightsquigarrow \kappa, v$.

**Reached and Arising Snapshots** When defining adherence to policies, it is essential to consider only those snapshots (i.e., configuration and code pairs) which may arise through the execution of the given modules. For example, if we considered any well-formed snapshots (well-formed in the sense of the type system), then we would be unable to show, e.g., that balances are always positive as mandated in $[9, 10]$. Namely, a configuration with a negative balance would be well-formed, but will never actually arise in the execution of the program.

$\text{Reach}(M, \kappa, \text{code})$ [$5$] is the set of snapshots corresponding to the start of the execution of the body of any constructor or method called in the process of executing code in the context of $M$ and $\kappa$. Note that $\text{Reach}(M, \kappa, \text{code})$, corresponds to the complete body of a method.

$\text{Arising}(M)$ is the set of snapshots which may be reached during execution of some initial snapshot, $s_0$, code).

**Accessible Objects** $\text{AccAll}(M, \kappa)$ is the set of objects accessible from the frame in $\kappa$ through any path, including confidential fields.

The notation $z \vdash c$ indicates that $z$ is the name of an object which exists in the heap of $\kappa$ and belongs to class $c$ — with no requirement that there should be a path from the frame to this object. The notation $\kappa \in c$ expresses that the currently executing method in $\kappa$ comes from $c$, while $\kappa \in M$ expresses that the class of the currently executing method is defined in $M$.

**Execution Histories** An execution history, $h$, is a sequence of snapshots containing all the method calls which arise during one execution, in the order in which these calls were received, and where the snapshots of nested calls follow the snapshot of the nesting call. We use the operator $\cdot$ to compose histories, for example $h_1 \cdot (\kappa, \text{code}) \cdot h_2$ is a history. The set $\text{Histories}(M)$ is the set of all histories which execute code from $M$.

4. Escrow Agent

The basic mint and purse system allows two untrusting components to make a payment (see section 2 above). But this payment is a one way transaction: a payer pays a payee. Supporting “electronic rights” (in Miller et al. [10]’s redolent phrase) requires contracts, that is, two way exchanges where some currency and some goods (e.g shares) change hands atomically — again in an environment where neither payee nor payer trust each other. The remainder of the Escrow case study uses mints and purses to build such a Escrow Contract system. We now describe our Grace translation / simplification of the escrow system, showing its implementation and the capability policies that are crucial to its correctness.

Given that each side is untrusting of the other, the first challenge is for both sides to have received the same contract object with the same understanding that this object will embody the actual contract. Following Miller et al. [10]’s design we provide a trusted escrow agent object that issues contracts to buyers and sellers.

Our escrow agent is shown in figure 3. Compared with Miller et al. [10]’s escrow agent, this is simpler and more straightforward, because again our code intentionally focuses on the core behaviour of the design, especially regarding the object capabilities, rather than details of middleware infrastructure. Our design also makes a number of other simplifications: we assume Grace objects and classes are single-threaded as if in fully-synchronized Java; and we adopt a very asymmetric protocol where the seller (payee) must always ‘move’ first, followed by the buyer (payer). The waitingForSeller variable keeps track of who should move next, the buyer or the seller.

Thus, the seller asks for and is returned a contract; when a matching buyer arrives they will be issued the same contract object.

// Alice the seller moves first
def alice = object
  def alicesContract = escrowAgent.getSellerContract("some terms")
...

// Bob the buyer moves second
def bob = object
  def bobsContract = escrowAgent.getBuyerContract("some terms")
...

Of course, a real implementation would need to keep track of a list of potential buyers, so that a buyer requesting a contract with no matching seller would not deadlock — but these are essentially bookkeeping issues that do not affect the exercise of object capabilities within the system.

Compared with Miller et al. [10] we also rely on a number of end-to-end arguments. For example there is no way for a seller to determine that no buyer has accepted the contract, or for a buyer or seller to explicitly reject a contract when issued. The reason we don’t deal with these issues here is that the contract object itself, and the protocol around its use, seamlessly resolve all these issues in the end-to-end context of the system as a whole.

4.1 Capability Policies for the Escrow Agent

We propose some policies which govern the class escrowAgent. We believe that the policies below are essential, but giving a complete set of policies is beyond the scope of this paper.
4.1.1 Pol_A1: getSellerContract returns fresh contracts.
The method getSellerContract returns fresh contracts, or throws
an exception. We distinguish two versions of the policy:

\[
\text{Module } M \text{ satisfies policy Pol_A1, vrs1 } \\
\text{iff } \\
\forall \kappa. M, \kappa, \text{escrowAgent}. \text{getSellerContract(\_)} \sim \kappa', v \\
\text{v \in Error } \lor \ ( v \vdash \kappa' \text{ contract } \land v \notin \text{dom}(\kappa) ) \\
\implies \\
\text{in other words, if we execute escrowAgent.\text{getSellerContract(\_), then we either obtain an exception (v \in Error), or we obtain a fresh contract (v \notin \text{dom}(\kappa)) and v \vdash \kappa' contract).} \\
\text{On the other hand, Pol_A1, vrs2 gives sufficient conditions under which the execution will not throw an error, and describes the state after execution of \text{getSellerContract(\_).}}
\]

\[
\text{Module } M \text{ satisfies policy Pol_A1, vrs2 } \\
\text{iff } \\
\forall \kappa, t: \text{String}. [\text{escrowAgent}.\text{waitingForSeller}]_{\kappa'} \\
\land M, \kappa, \text{escrowAgent}.\text{getSellerContract(t)} \sim \kappa', v \\
\implies \\
\land [\text{escrowAgent}.\text{waitingForSeller}]_{\kappa'} = \text{false} \\
\land \ [\text{escrowAgent}.\text{contract}]_{\kappa'} = v \land \ [\text{v.terms}]_{\kappa'} = t \\
\land \ [\text{escrowAgent}.\text{offered}]_{\kappa'} = \text{false} \\
\implies \\
\text{thus, if escrowAgent is in state where \text{waitingForSeller} is true, and executes \text{getSellerContract(t)}, then \text{waitingForSeller resp. offered} will become false resp. true, and a new contract will be created, containing t as the terms.} \\
\text{the above specification is a complete description of the behaviour of \text{getSellerContract}, but exposes its implementation's details, some of which are only relevant to the working of other methods, e.g. setting the field text is necessary for the correct working of \text{getBuyerContract}.} \\
\text{the well-known approach to alleviate such exposures is the introduction of ghost variables [6, 7]; in section 4.1.4 we will show instead specifications on the possible sequences of method calls — in line with [3], we believe that this is a more abstract style.} \\
\text{note that both versions of Pol_A1 could have been expressed in a Hoare Logic variant, e.g. in [6]. they both are concerned with sufficient conditions for the creation of contracts. The next policy, Pol_A2, will be concerned with necessary conditions. Using the terminology we introduced in [4], the former is a rely property, while the latter is a deny property, i.e. a new contract cannot be created unless the method \text{getSellerContract} is called.}
\]

4.1.2 Pol_A2: Contracts created only through getSellerContract

The policy Pol_A2 guarantees that any newly created contract must have been the result of calling \text{getSellerContract} on escrowAgent.

\[
\text{Module } M \text{ satisfies policy Pol_A2 } \\
\text{iff } \\
\forall M'. \forall \kappa \notin \text{escrowAgent, contract} \\
\land M = M', \kappa, \text{code} \sim \kappa', v \\
\land \ c \vdash \kappa' \text{ contract } \land c \notin \text{dom}(\kappa). \\
\implies \\
\text{in other words, if execution of code external to } M (\kappa \in M') \text{ leads to a configuration which has access to contract object } c (\{c\}_{\kappa} \in \text{AccAll}(M \ast M', \kappa') \land c \vdash \kappa' \text{ contract}), \text{then either } c \text{ was}
\]

4.1.3 Pol_A3: Contracts may only be obtained through calls of getSellerContract or getBuyerContract

Pol_A3 guarantees that the only way to obtain contracts encapsulated in a module M is through the call of the functions \text{getSellerContract} or \text{getBuyerContract}. Thus, this policy forbids the addition, of, say, a public method \text{getContract} that returns a contract, and which would make it possible to leak a contract to a malicious third party.
already accessible to $M'$ ($\sigma_{rev} \in \text{AccAll}(M \cdot M', \kappa)$), or $c$ was returned through execution of escrowAgent.getSellerContract(_), or escrowAgent.getBuyerContract(_).

4.1.4 PolA4: Interplay of getBuyerContract and getSellerContract

PolA4, vrs1 guarantees that a call of getSellerContract(text) followed by a call to getBuyerContract(text) will return the same object, provided that it is not preceded by an unmatched call of getSellerContract(_), and provided that there is no intermediate call on the escrowAgent between the two calls.

We first define the predicate $\text{Matched}(h)$ on histories, which requires that either there are no calls on escrowAgent, or that any getSellerContract(_) are followed by a corresponding call of getBuyerContract(_):

\[
\text{Matched}(h) \iff (\_, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \notin h, \quad \text{or there exists } h_1, h_2, h_3 \text{ such that } \\
\quad h = h_1 \cdot (\kappa_1, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_2 \cdot (\kappa_2, \text{escrowAgent}.\text{getBuyerContract}(\text{text}) \cdot h_3) \\
\quad \text{and } \text{Matched}(h_1) \text{ and } (\text{escrowAgent}.\text{getBuyerContract}(\text{text}) \cdot h_2) \\
\quad \text{and } \text{Matched}(h_3).
\]

We now express the policy:

\[
\text{Module } M \text{ satisfies policy PolA4} \\
\text{if } \\
\forall M', h_1, h_2, h_3, \text{text}: \text{String}, \forall h \in \text{Histories}(M \cdot M'). \\
\quad h = h_1 \cdot (\kappa_1, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_2 \cdot (\kappa_2, \text{escrowAgent}.\text{getBuyerContract}(\text{text}) \cdot h_3) \\
\quad \text{and } \text{Matched}(h_1) \text{ and } (\text{escrowAgent}.\text{getBuyerContract}(\text{text}) \cdot h_2) \\
\quad \text{and } \text{Matched}(h_3).
\]

The policy from above, together with PolA1, vrs1 will guarantee that the returned value is a contract, which is new in $\kappa_2$.

We now consider the deny-counterpart of that policy, which says that calls of getSellerContract(_), only succeed if the previous calls on escrowAgent are matched correctly. We omit the complementary policy for the calls of getBuyerContract(_).

\[
\text{Module } M \text{ satisfies policy PolA4, vrs2} \\
\text{if } \\
\forall M', h_1, h_2, h_3, \text{text}: \text{String}, \forall h \in \text{Histories}(M \cdot M'). \\
\quad h = h_1 \cdot (\kappa_1, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_2 \cdot (\kappa_2, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_3) \\
\quad \text{and } M \cdot M', \kappa_1, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_2 \cdot (\kappa_2, \text{escrowAgent}.\text{getSellerContract}(\text{text}) \cdot h_3) \\
\quad \text{and } \text{Matched}(h_1) \\
\quad \Rightarrow \exists \kappa_3, \kappa_4, v. \\
\]

5. Escrow Contract

Figure 4 implements our version of the actual escrow contract object. Again, this uses a protocol where we expect the seller to move first and populate the contract, and then the buyer to move and accept the contract. The seller must pass two purses into the offer method — the sellersGoods purse that contains the goods to be sold, and sellersMoney a (presumably empty) temporary purse that will receive the money. Similarly the buyer supplies an empty buyersGoods purse and a (temporary) buyersMoney purse that contains the payment when they call the bid method. The transaction succeeds if the bid method completes.

// Alice the seller
\[
\text{def mDst = mint.newPurse("Alice's mDst", 0)} \\
\text{def gSrc = goods.newPurse("Alice's gSrc", 7)} \\
\text{alicesContract.offer(gSrc, 7, 10, mDst)}
\]

// Bob the buyer
\[
\text{def mSrc = mint.newPurse("Bob's mSrc", 10)} \\
\text{def gDst = goods.newPurse("Bob's gDst", 0)} \\
\text{bobsContract.bid(gDst, 7, 10, mSrc)}
\]

The seller can cancel the contract at any time before settlement by simply emptying their sellersGoods purse: this will cause the overall transaction to fail when the buyer makes a bid. Similarly
5.1 Capability Policies for Contracts

5.1.1 Pol_C1: Effects and conditions of exchange

The key policy for the escrow contract is that once it receives a bid matching the offer, it performs the transfer of goods and money, provided the corresponding purses come from the same mint. Here we write the deny counterpart of this policy, which says that if, as the result of a bid, the buyer’s goods balance changes, then the bid must have been preceded by the creation of pairs of purses in corresponding mints, and of an offer which matched the bid:

Module M satisfies policy Pol_C1

\[
\begin{align*}
\text{iff} & \quad \forall M', h_1, h_2, c, \kappa, \kappa', \forall \text{amt}, \text{prc} : N. \\
& \quad \forall h \in \text{Histories}(M \times M'). \\
& \quad c : \kappa \text{ contract } \land p_k : \text{Purse } \land p_m : \text{Purse} \\
& \quad \land h = h_1 \cdot (\kappa, \text{c.bid}(p_k, \text{amt}, \text{prc}, p_m)) \cdot h_4 \\
& \quad \land M \times M', \kappa, \text{c.bid}(p_k, \text{amt}, \text{prc}, p_m) \sim \kappa', v \\
& \quad \land [p_k \text{balance}]_{\kappa'} \neq [p_k \text{balance}]_{\kappa} \\
& \quad \Rightarrow \\
& \quad \left( [p_k \text{balance}]_{\kappa'} = [p_k \text{balance}]_{\kappa} + \text{amt} \\
& \quad \land [p_m \text{balance}]_{\kappa'} = [p_m \text{balance}]_{\kappa} - \text{prc} \\
& \quad \land [p_k \text{balance}]_{\kappa} \geq \text{prc} \\
& \quad \land \left[ \exists p_k', p_m' \text{ Mint } \land \text{mnt}_k, \text{mnt}_m \text{ Mint} \\
& \quad \land (\kappa_1, \text{mnt}_k, \text{makePurse}()) \in (\kappa_3, \text{mnt}_k, \text{makePurse}()) \in h_1 \\
& \quad \land (\kappa_2, \text{mnt}_m, \text{makePurse}()) \in (\kappa_4, \text{mnt}_m, \text{makePurse}()) \in h_3 \\
& \quad \land M \times M', \kappa_1, \text{makePurse}() \sim p_k, \kappa_3 \\
& \quad \land M \times M', \kappa_2, \text{makePurse}() \sim p_m, \kappa_4 \\
& \quad \land M \times M', \kappa_3, \text{makePurse}() \sim p_k, \kappa_2 \\
& \quad \land M \times M', \kappa_4, \text{makePurse}() \sim p_m, \kappa_3 \\
& \quad \land \exists p_m' : \text{Purse } \land p_m : \text{Purse} \\
& \quad \land [p_k' \text{balance}]_{\kappa'} = [p_k \text{balance}]_{\kappa} - \text{amt} \\
& \quad \land [p_m' \text{balance}]_{\kappa'} = [p_m \text{balance}]_{\kappa} + \text{prc} \\
& \quad \land (\left( \text{c.bid}(p_k', \text{amt}', \text{prc}', \text{p_m}')) \in h_3 \rightarrow \text{amt} \neq \text{amt}' \\
& \quad \land [p_k' \text{balance}]_{\kappa} \geq \text{amt} \right) 
\end{align*}
\]

In the above, \( p_m \) and \( p_k \) stand for the buyer’s money and goods’ purse, while \( p_m' \) and \( p_k' \) stand for the seller’s money and goods’ purse. The policy expresses that if a call of bid modified the balance in the buyer’s goods purse, then the modification was exactly by \text{amt}, while the buyer’s money purse will have been debited the price. Moreover, there exist further purses, \( p_m' \) and \( p_k' \) which were created by the same mints as \( p_m \) and \( p_k \) respectively (lines 14-25). Furthermore, \( p_k \) is debited by \text{amt}, while \( p_m \) is credited by \text{prc} (lines 26-27). Also, there was no bid of sufficient money between the offer at \( \kappa' \) and the bid at \( \kappa \) (lines 28-29) and there were enough goods in the seller’s goods’ purse (line 30).

5.1.2 Pol_C2: Bids do not leak references

Finally, this policy guarantees that execution of a bid does not leak references to third parties — satisfying the requirement that the escrow system mediate between two mutually untrusting objects.

Module M satisfies policy Pol_C2

\[
\begin{align*}
\text{iff} & \quad \forall M', \kappa, c, o, o' \\
& \quad (\kappa, \text{c.bid}(p_k, \text{amt}, \text{prc}, p_m)) \in \text{Arising}(M \times M'). \\
& \quad \land c : \kappa \text{ contract} \\
& \quad \land M \times M', \kappa, \text{c.bid}(p_k, \text{amt}, \text{prc}, p_m) \sim \kappa', v \\
& \quad \land o \in \text{AccAll}(o', \kappa') \\
& \quad o \in \text{AccAll}(o', \kappa) 
\end{align*}
\]

In other words, if after execution of \( \text{c.bid}(p_k, \text{amt}, \text{prc}, p_m) \), the object \( o' \) has access to the object \( o \), then it already had access to it before execution of the call.

6. Conclusions and Future Work

In this paper we have presented a rational reconstruction of Miller, Cutsem, and Tulloh’s contract exchange escrow case study. We have translated the code from JavaScript to Grace, and have removed code corresponding to the crosscutting concerns of distribution, asynchrony, generivity, and symmetry, leaving the core of the program that records, exchanges, and exercises object capabilities.

We have proposed some of the capability policies that must be maintained to demonstrate that the escrow design meets its specifications, and clarified them through formal specifications.

In further work, we want to complete the definition of execution observations, complete the policies, consider whether they are minimal (can any of these be inferred from the others), and prove that the Grace code adheres to these policies.

Acknowledgments We thank the anonymous referees for their comments. This work is partially supported by the Royal Society of New Zealand Marsden Fund, and by the EU FP7 project Upscale.

References