SAMOA: A Framework for a Synchronisation-Augmented Microprotocol Approach*

Paweł T. Wojciechowski, Olivier Rütti, and André Schiper
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
{First.Last}@epfl.ch

Abstract

We address programming abstractions for building protocols from smaller, reusable microprotocols. The existing protocol frameworks, such as Appia and Cactus, either restrict the amount of concurrency between microprotocols, or depend on the programmer, who should implement all the necessary synchronisation using standard language facilities. We develop SAMOA – a framework for a Synchronisation-Augmented Microprotocol Approach. It has been designed to allow concurrent protocols to be expressed without explicit low-level synchronisation, thus making programming easier and less error-prone. In this paper, we describe versioning *concurrency control algorithms. They are used by the runtime system of our framework to guarantee that the concurrent execution of a protocol is equivalent to a serial execution of its microprotocols. This guarantee, called the* isolation property*, ensures consistency of session or message-specific data maintained by microprotocols.*

1 Introduction

Modularization is a well-known technique for simplifying complex communication systems. Protocol frameworks, such the x-kernel [12], Cactus [25], Appia [17], and Ensemble [9] have been built. They allow complex protocols to be implemented as compositions of separate *microprotocols* that communicate using the framework's interface [24]. This approach helps to clarify the dependencies among properties required by a given communication system, allows for code reuse, and makes it possible to construct systems that are customized to the specific needs of the application or underlying network environment. The protocol frameworks also support primitives that can simplify the construction of network protocols, such as support for processing messages, marshalling messages to the network format, and timeouts.

The design of modular protocols is however problematic: some microprotocols may have to synchronise their actions in order to maintain consistency of session or message-specific data (the problem will be illustrated on the example of group communication protocols in Section 3). The goal of our work was to design programming support for leaving the exact implementation of this synchronisation to a runtime system rather than requiring programmers to identify what synchronisation is needed, and where to acquire and release locks. For the class of synchronisation problems we consider here, a sufficient correctness condition is the isolation property. Execution of concurrent microprotocols satisfies the *isolation property* if the protocol's state is equivalent to a state that can be produced by some serial execution of the microprotocols. Our approach has close analogies with the concept of *transactions*. The isolation property is like isolation in database transactions, however, it is not accompanied by additional properties defined in the transactional model, i.e. atomicity, consistency, and durability. In this paper, we focus on a lightweight implementation for enforcing the isolation property in protocol frameworks.

The existing software frameworks do not support the isolation property in a completely satisfactory way. For instance, Appia does not allow for concurrent execution of microprotocols: all actions which result from a single external event (such as message arrival) are handled by microprotocols sequentially using FIFO communication channels; this excludes a number of reasonable *concurrent executions* of microprotocols, e.g. for dealing with messages from the application and network at the same time, for processing time consuming I/O operations in background, or in order to support multiprocessor architectures. Cactus does not restrict the amount of concurrency but, on the other hand, it depends on the programmer, who must implement the required synchronisation policy using standard

*Research supported by the Swiss National Science Foundation under grant number 21-67715.02 and Hasler Stiftung under grant number DICS-1825.

language facilities (such as locks, semaphores, and monitors). The synchronisation code is however rather subtle and error-prone, especially for highly-concurrent protocols. In this paper, we try to find a better solution.

To study the problem of synchronisation and isolation in the design of concurrent, modular protocols, we have developed a *Synchronisation-Augmented Microprotocol Approach (SAMOA)*. The main feature of our approach is separation of the low-level synchronisation from the protocol's logic. We have implemented a protocol framework based on this approach as a Java [1] library; the implementation is available [19]. The SAMOA programmer has only to declare which events are *external*; the runtime system will use *concurrency control* algorithms to enforce that the effects of one *computation*, defined informally as the execution of all microprotocols involved in processing of a single external event, are not visible to other computations executing concurrently; from the perspective of a computation, it appears that computations execute sequentially rather than in parallel. Computations correspond to (possibly multi-threaded) transactions with no support for atomicity and durability. However, unlike transactions whose termination is defined statically using "commit" and "abort", computations are never aborted, and have scope which depends on a given composition of microprotocols.

We have designed several *deadlock-free concurrency control* algorithms that ensure the isolation property. They have been used to implement the SAMOA runtime system. The algorithms can be classified into two groups: 1) versioning algorithms with allocation of access to event handlers, and 2) timestamp-ordering algorithms with rollback/recovery. In this paper, we focus on the first group, and describe three algorithms that we use in SAMOA: the basic *version-counting* algorithm (VCA_{basic}) and two extensions of this algorithm, VCA_{bound} and VCA_{route}. The latter two algorithms can support more parallelism, however, they achieve that by demanding some additional properties of the protocols to be specified by the programmer. The VCA_{bound} algorithm requires the least upper bound on the number of times a given microprotocol can be executed by a computation, VCA_{route} requires the pattern of handler calls to be specified.

The advantages of our framework with respect to existing protocol frameworks are twofold: 1) it makes programming of concurrent protocols easier since the programmer does not need to implement the low-level synchronisation (we illustrate this point in the paper), and 2) it can also greatly simplify reasoning about protocols and doing the correctness proofs. The latter claim follows from the fact that the isolation property allows us to reason about concurrent computations as they would be executed sequentially.

The paper is organized as follows. Section 2 describes our model. Section 3 gives a brief illustrative example of SAMOA code. The code is used to show some non-trivial programming errors in the design of concurrent protocols. The construct of the SAMOA language that eliminates this type of errors is informally described in Section 4. Section 5—the heart of our paper—describes the concurrency control algorithms that we designed for SAMOA. Section 6 discusses related work, and Section 7 sketches our plans for the future development of SAMOA.

2 Model

Protocol frameworks, such as Cactus (which builds on the x-kernel), Appia, and our framework, can be described using a simple event model. In this model, protocols consist of code blocks that communicate (synchronously or asynchronously) using internal events, and may also react on external events and output values. Below we define the model, and use a small example of a protocol in this model to illustrate the notion of the isolation property.

Protocols consist of code blocks called *event handlers*. Several related handlers can be grouped into a single *microprotocol* and share a *local state* of the microprotocol's object. Execution of a handler can directly modify only the local state of its own microprotocol. The *protocol's state* is the union of (disjoint) local states of all microprotocols, and encompasses all of the in-memory and on-disk data items that affect the protocol's operation. For simplicity, we confuse handlers and microprotocols (simply assuming that each microprotocol has only one handler). In SAMOA programs, a microprotocol can consist of many handlers.

Executions of handlers are triggered by *events*. An event is a request (at run time) to call a handler. Each event must specify an *event type* — only handlers that have been bound to this event type will be executed as the result of the event. Any events generated by the execution of handler(s) that have been triggered by an event a, are *causally dependent* on a; the causality relation is reflexive and transitive. Some events are not causally related; they are *concurrent*. An event a is *pending* if there is at least one handler requested by a that has not commenced yet in response to event a.

We have two kinds of events: internal and external. An *internal event* generated during a handler's execution triggers the execution of another (or the same) handler. The execution of a handler can generate zero, one or more internal events. *External events* of a protocol are normally: 1) requests by the network layer (or application) to inject a message received from the network (or application) to the protocol, and 2) a timeout action. In practice, some internal events may be also regarded as external.

Execution of a protocol is modelled as a *run*, defined as a list of pairs (a, H), where a is an event and H is a handler requested by a that has begun execution; if H has not commenced yet, we write H^{∂}. The list is ordered according to

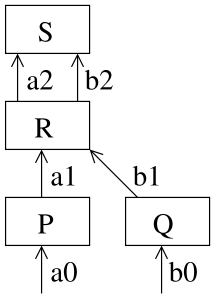

Figure 1. Events and handlers

the order of time when handlers commenced, or, in case of pairs (a, H^∂) where a is pending, when events were issued. A run is *complete* if it does not have pending events.

To illustrate the notions introduced so far, let us take an example protocol with four handlers P, Q, R, and S, as illustrated in Fig. 1. It can, for example, receive UDP packets either from *ad-hoc* network (to be processed by P) or fixed network (to be processed by Q), and deliver them to handler S; we abstract away from details. Handlers R and S are assumed to be atomic (i.e. only one instance at a time).

We assume that two external events a_0 and b_0 have occurred. In all possible executions of this protocol triggered by these events, event a_0 (correspondingly b_0) triggers the execution of handler P (correspondingly Q); and handler R is executed twice, once as the result of internal event a_1 (which causally depends on event a_0), another time as the result of internal event b_1. The events a_0 and b_0 are concurrent, also events a_1 and b_1 are concurrent. Example complete runs of the protocol are:

$$
\begin{aligned}
r_1 &= ((a_0, P), (a_1, R), (a_2, S), (b_0, Q), (b_1, R), (b_2, S)) \\
r_2 &= ((a_0, P), (b_0, Q), (a_1, R), (a_2, S), (b_1, R), (b_2, S)) \\
r_3 &= ((a_0, P), (b_0, Q), (a_1, R), (b_1, R), (b_2, S), (a_2, S))
\end{aligned}
$$

An external event c spawns a *computation*, defined as a subsequence of a run, containing c together with a set of *all* events that causally depend on c but excluding any other (causally dependent) external events $d, e, ...$, and any events that causally depend on $d, e, ...$. Computations spawned by such events $d, e, ...$ are *caused* by the computation spawned by c. In a protocol run, we require all computations to eventually complete. A computation is *complete* when the execution of all handlers triggered by (events of) the computation have completed, and no event is pending. In the runs above, we have two computations: $k_a = ((a_0, P), (a_1, R), (a_2, S))$ and $k_b = ((b_0, Q), (b_1, R), (b_2, S))$; they are not causally related.

Consider a complete run with a finite set of external events $E = \{a_0, b_0, ...\}$. The protocol execution is *serial* if for each two (distinct) external events a_0 and b_0 in E, either (each handler of) the computation spawned by a_0 commences after the computation spawned by b_0 has completed, or vice versa. In the serial run, a computation k always precedes in time any computations caused by k, i.e. they can commence only after k has completed. In our example, run r_1 is serial since computation k_b begins in this run after k_a has completed; but runs r_2 and r_3 are not serial.

Two protocol executions are *equivalent* if, considering the same sequence of external events and the same initial state of the protocol, they produce the *same* (or *equivalent*) state. A protocol execution satisfies the *isolation property* if the execution is equivalent to some serial execution.

Consider runs r_1, r_2, and r_3 (with the sequence of external events (a_0, b_0)). Note that in runs r_1 and r_2, the computation k_a visits all the microprotocol's objects that are shared with computation k_b (i.e. R and S) *before* k_b visits these objects. Thus, the effects of computation k_a in these runs, such as any modifications of the protocol's state, do not affect computation k_b that is executing concurrently. Hence, runs r_1 and r_2 satisfy the isolation property. Note however that run r_3 does not, since k_a can see any modification of object S done by event b_2 of computation k_b, and k_b can see any modification of object R done by k_a.

If we would express and execute our example protocol in Cactus, all the example runs r_1, r_2, and r_3 can occur, unless the protocol programmer would explicitly synchronise the execution of given handlers to forbid some runs. In SAMOA, the programmer can easily declare constraints on the protocol's concurrent executions so that only those runs that satisfy the isolation property are permitted (e.g. r_1 and r_2, but not r_3). Appia also supports the isolation property, however it only permits serial executions, such as r_1; other correct, concurrent runs, such as r_2, cannot occur in Appia.

3 Example

Below are fragments of three microprotocols, which are part of a group communication system that we develop using SAMOA; the system's architecture is described in [15], see also [6]. In complex middleware protocols such as ours, it is often necessary to execute some activities concurrently, in order to: 1) achieve good response time (for instance when performing slow I/O operations), 2) avoid blocking while processing different types of messages, or 3) simply to gain benefit of the multi-processor architectures. Unfortunately, introducing a lot of concurrency can lead to programming errors that are notoriously difficult to detect.

The goal of this section is to:

- illustrate on concrete protocols the model defined in Section 2, and SAMOA's programming constructs,
- explain the problem of errors that are due to lack of synchronisation between concurrent events,

- describe some simple solutions to the problem that are often used by protocol designers (in the next section, we demonstrate a more straightforward and efficient solution using SAMOA's `isolated` construct).

The SAMOA framework, implemented as a Java library, supports the event-based model of communication between microprotocols, with basic primitives for binding event types to event handlers and triggering the execution of handlers. Microprotocols in SAMOA can be expressed as subclasses of two (abstract) classes `Protocol` and `handler`. Event types are defined as instantiations of a class `Event`. Event types and names of handlers are first-class programming objects. They can be passed as arguments to functions and handlers, returned as results and stored in data structures. Events can be bound to handlers using `bind`; in our program this takes place upon creation of the microprotocol objects.

Group Communication Example. The first microprotocol, called $RelCast$, implements a reliable broadcast. It builds upon a second microprotocol, called $RelComm$, that implements reliable point-to-point communication. We assume that sites can crash at any time. The third microprotocol — $Membership$, maintains a *view*, i.e. a current set of all sites that are considered to be nonfaulty. This set is kept consistent across all sites. We omit a lot of uninteresting details, focussing on the synchronisation problem and the way how it can be solved in SAMOA. (For simplicity, we present code in a Java-like language instead of giving a concrete syntax in Java.)

```
Protocol Membership (ViewChange:Event, ...) =
{ ...
  handler joinleave (op: {+,-}, site: Site)
      trigger ABcast [op site];
  handler deliverView (op, site: Site)  {
      view = view op site;
      triggerAll ViewChange view; } ...
}
```

The $Membership$ microprotocol transforms a $view$ into a new view each time a site joins or leaves the system (voluntarily or because it is suspected to have failed). To join a site s, the $joinleave$ handler must be called. It uses `trigger` to synchronously call a (single) handler bound to the event type $ABcast$, passing as the handler's argument an operation '$+$' paired with the new site to be joined. The event is handled by the atomic broadcast microprotocol that atomically broadcasts the value $[+\ s]$ to other sites using the distributed consensus microprotocol (both microprotocols depend on $RelCast$, and are omitted here). Upon delivering the value to handler $deliverView$ (on each site) by the atomic broadcast protocol, the updated view $view + s$ is propagated locally to all interested microprotocols using a (synchronous) event of type $ViewChange$. To issue the event, the `triggerAll` construct is used, which accepts event types that can be bound to many handlers.

```
Protocol RelCast (SendOut : Event,
   DeliverOut : Event,  Bcast : Event,
   FromRComm : Event, ViewChange : Event) =
{
  GroupView view = top.initialView();
  handler bcast (m : Message) {
      for all site in view
        trigger SendOut (m, site); }
  handler recv (m : Message) {
      if (new message m) then {
        bcast m;
        asyncTriggerAll DeliverOut m; } }
  handler viewChange (new_view)
    { view = new_view; }
  bind Bcast bcast;
  bind FromRComm recv;
  bind ViewChange viewChange;
}
```

To reliably broadcast a message m to a group of sites, the $bcast$ handler of $RelCast$ can be called. The body of $bcast$ (see above) issues an event of type $SendOut$ (or, we simply say "event $SendOut$"), one for each site in the $view$; the events carry the message m and a target site $site$. The event type $SendOut$ is bound to a handler $send$ of the $RelComm$ microprotocol (explained below), which tries to send the message to a given target site. Note that upon receipt of the message on another site for the first time (see handler $recv$ above), the message is rebroadcast again so that it will be delivered to all sites in the current view, even if the sender would crash in the middle. The message is delivered locally using the (asynchronous) $DeliverOut$ event.

```
Protocol RelComm (FromRComm : Event,
   SendOut : Event, FromNet : Event,
   ViewChange : Event) =
{
  GroupView view = top.initialView();
  handler send (m:Message, target:Site) {
      if (target in view)
        try to send m to target; }
  handler recv (m:Message, sender:Site) {
      if (sender in view)
        asyncTriggerAll FromRComm m; }
  handler viewChange (new_view)
    { view = new_view; }
  bind SendOut send;
  bind FromNet recv;
  bind ViewChange viewChange;
}
```

To reliably send a network message to a target site, the handler $send$ of the $RelComm$ microprotocol can be called; it tries to deliver the message (with possible retransmissions in case of failures, omitted here). The message is discarded if the target is not known. Before delivering a message on the target site, the microprotocol checks if the sender is in the current group view (see handler $recv$); if so then the $FromRComm$ event is issued. All microprotocols bound to this event (such as the $recv$ handler of the $RelCast$ protocol) will get the message.

Both $RelComm$ and $RelCast$ depend on a group view. $RelComm$ tries to deliver a message only to a site that is in the current view (this requirement is necessary to implement finite buffers, we omit details). Therefore $RelCast$ can reliably broadcast *only* to those processes that are within the view seen by $RelComm$. Thus, each time the view has changed, a new consistent view must be propagated to $RelComm$ and $RelCast$ by $Membership$, soon after $Membership$'s distributed view-change algorithm has terminated on each site. For this, $Membership$ notifies each interested microprotocol about the new view by issuing the $ViewChange$ event; to deliver views to all in a sequential order a *synchronous* `triggerAll` is used (see $Membership$'s code). Then, the $viewChange$ handlers of $RelCast$ and $RelComm$, which have been bound to $ViewChange$, can update their local copy of the view.

Problem. Imagine that $Membership$ has just installed a new view, with a new site s added, i.e. on each site it has added s to its local view, and issued a local $ViewChange$ event. Suppose a message is being broadcast using this new view. Then, if the message has been received by $RelCast$ on some site for the first time, it has to be rebroadcast again (to satisfy $RelCast$'s algorithm). However, if $RelComm$ which is used by $RelCast$ to reliably send the message has not yet handled the local $ViewChange$ event informing (locally) about the new view, it will not send the message to s, thus breaking the algorithm. The message will be silently discarded since $RelComm$ does now know about s.

Restricted Solutions. Simple, although restricted solutions to this problem are, either to: 1) piggyback the view with every message and require each microprotocol to refer to it (then a local copy of the view is obsolete), 2) demand the current view from $Membership$ by each microprotocol (in particular $RelComm$) before processing any incoming message, or 3) provide some form of synchronisation between events so that the application/network messages cannot be processed by any microprotocol while the microprotocols are in the process of updating their local views. The first two solutions provide a mechanism which is somehow stronger then actually needed; they either require redundant data in messages (in the first case), or depend on some additional intermodule communication (in the second case). The third solution may unnecessarily restrict the amount of internal parallelism.

Solution by Isolation. Note that to solve our problem it is sufficient if the (correct) computations of the protocol, i.e.: processing each message from the network, and processing each message from the application, would be executed concurrently, but in such a way that the isolation property holds. (So essentially, the outcome would be the same as when using synchronisation suggested in the paragraph above.) In SAMOA, such desired behaviour can be simply declared, by implementing any external events of a protocol using a construct `isolated`. In the next section, we describe the construct and explain how to use it in our example protocol.

4 Language Support for Isolation

Execution of `isolated` M e in SAMOA spawns a new computation, where M is explained below, and e is code that calls some handler (or handlers) of the new computation. Each computation is executed by a separate SAMOA thread (if parts of a computation need to be executed concurrently, new threads can be created dynamically, either explicitly with a suitable construct, or implicitly through asynchronous event triggers). The runtime system guarantees that the concurrent execution of computations satisfies the isolation property defined in Section 2.

The basic construct `isolated` M e requires to specify a collection M of all microprotocols whose handlers *may* be called by the computation. For instance, an external event a_0 in Figure 1 can be expressed as `isolated [`P R S`]` `{ trigger` a_0 m`; }`, where P, R and S are all microprotocols (handlers) that may be called, and m is a message. An error exception is thrown in the thread that called `isolated`, if the computation will attempt to call a handler of a microprotocol that is not in M. There is no problem if some microprotocol declared in M is not called by the computation. In the strongly-typed language, the proper value of argument M could be inferred statically.

We currently do not deal with *dynamic binding*; all handlers declared in M must be bound before `isolated` commences and cannot be (re)bound inside any computation.

Example Revisited. In our example in Section 3, external events are requests, which are made by Network Module to call handler(s) of event type *FromNet*, and by Application Module to call handler(s) of *Bcast*. Thus, a message m received from the network must be injected to upper protocols by Network Module using `isolated [relComm relCast ...] {trigger FromNet m;}`, where *relComm*, *relCast*, ... are all microprotocol objects whose handlers may be called by the spawned computation; similarly for the other external event type.

We have also implemented a few more variants of `isolated` (each one to be used exclusively) that permit to have more parallelism by optimising unnecessary blocking (in the worst case, they behave like `isolated`). However, they demand some additional (orthogonal) properties of protocols to be known. It may not always be possible to identify these properties (e.g. to guess the least upper bound required by the first variant, if programs use recursion). The use of the variants is therefore limited. The algorithms used in the implementation of the constructs presented here are described in Section 5.

Least-Upper-Bound. The `isolated_bound` $M\ e$ requires the least upper bound on the number of times each microprotocol p declared in M can be visited by the execution of expression e (to "visit" means to call any handler of p). A runtime error exception will be thrown if the number is exhausted. There is no problem, however, if a microprotocol will be visited less times than declared in M; however, less parallelism may be permitted than in the case when M is more accurate.

Routing Pattern. Another variant, `isolated_route` $M\ e$, requires the pattern of handler calls to be specified. The argument M in this construct is a collection of arrows of a directed graph, declaring the routing pattern of the computation spawned using the construct. An arrow in this graph is a directed pair $h_1 \mapsto h_2$, declaring that the body of handler h_1 may call h_2. If a handler h_1 tries to call but there is no directed route from h_1 to h_2 in the graph, then a runtime error exception will be thrown.

5 Implementation

The implementation of SAMOA consists of the event-based communication between microprotocols, message flow control, and concurrency control. Below, we focus on concurrency control, which is responsible for the isolated execution of computations.

We defined isolation to mean that the effects of one computation are not visible to other computations executing concurrently; from the perspective of a computation, it appears that computations execute sequentially rather than in parallel. Thus, the simplest possible solution would be to block spawning of a new computation until any other computations complete. It follows from the definition in Section 2 that the isolation property is satisfied since the protocol's execution *is* serial. However, this would mean that executions of computations are not interleaved. Hence, the protocol may make poor use of its resources, and so might be too inefficient. Therefore a better solution is needed.

Essentially, we want the runtime system to process *many* computations simultaneously while providing the illusion of isolation. For this, we have designed several blocking and optimistic concurrency control algorithms, each one implementing a variant of the `isolated` construct (with a different degree of optimization). Below, we describe three example blocking algorithms that are part of the SAMOA distribution [6]; they implement the three variants of the `isolated` construct described in Section 4.

Our concurrency control algorithms regulate when handlers bound to a given event type, are allowed to be triggered by pending events of this type, so that all runs of a protocol will satisfy the isolation property. In short, a handler to be called by a computation that has invoked a corresponding "trigger event" construct, can be effectively called only if the computation holds a valid *version number* for the microprotocol whose part the handler is. Otherwise, the "trigger event" is blocked. The version numbers thus protect the state of the microprotocol's object (which is assumed to be accessed externally only through calls to the microprotocol handlers).

We assume below that all handlers are bound at the protocol's start-up and cannot be rebound inside computations.

5.1 The Basic Version-Counting Algorithm

There is a global version counter gv_p for each microprotocol p (initialised to 0). Each individual microprotocol p maintains its local version counter lv_p (also initialised to 0). We assume that each occurrence of the `isolated` construct in the source code identifies uniquely a *computation type*. Computations, denoted $k_1, k_2, ...$, relate to dynamic instances of computation types. The Basic Version-Counting Algorithm (VCA$_{basic}$) is given by the following set of rules or steps (we require Steps 1 and 2 to be *atomic*):

1. At the moment of spawning a new computation k by `isolated` $M\, e$, for each microprotocol $p \in M$ whose handler *may* be called by this computation, increase the gv_p counter by one. Create a private copy pv_k of all version numbers computed as above, i.e. pv_k is a map (dictionary) containing bindings from all microprotocols $p \in M$ to their upgraded versions gv_p.

2. A computation k can call a handler h of microprotocol p only when it holds a version for this microprotocol that matches the current (local) version maintained by the microprotocol, i.e.
$$pv[p]_k - 1 = lv_p \ . \tag{1}$$

3. After a computation k has completed its execution (i.e. all threads of the computation terminated), for each microprotocol $p \in M$, wait until (1) is true, then upgrade the local version of the microprotocol p, so that we have $lv_p = pv[p]_k$; in the end, erase map pv_k.

Lemma 1 (Isolation Property) *If handlers are called only when allowed by Step 2 of the algorithm then the isolation property is satisfied.*

Proof (sketch) Consider a protocol with just two computations k_1 and k_2 that may call different handlers of the microprotocols declared in, correspondingly M_1 and M_2. Initially, all microprotocols have their local and global versions equal zero. The computation spawned first (say k_1) would atomically increase the global versions of microprotocols in M_1 by one, and build its private set of versions (here, each version equal 1); see Rule 1. The computation k_2, spawned after k_1, will also get its private set of versions for all microprotocols declared in M_2; however, versions of those microprotocols that have been also declared in M_1 will be equal 2 (again by Rule 1).

Consider a microprotocol p in $M_1 \cap M_2$. Each handler of this microprotocol can be freely called by k_1 since its private version of p decreased by one is 0, which is equal the current local version of the microprotocol, thus satisfying the equation (1) in Rule 2. However, by Rule 2 none handler of p can be currently called by k_2 since k_2 holds a private version of p equal 2 and $2-1=1 \neq 0$. However, by Rule 3 the local version of p will equal 1, that is the private version of p hold by k_1, after k_1 terminates. Then, k_2 is allowed to call a handler of p (by Rule 2). Since at this moment k_1 has already terminated, any changes done by k_2 to the state of p's object cannot affect k_1, which is what we wanted.

Consider a microprotocol p in $M_1 \cap M_2$ that has not been called yet by any of the two computations k_1 and k_2. If computation k_2 (whose private version of p is newer than the private version of p hold by k_1) is about to terminate now (according to our assumptions, it does not need to call a handler of p to be allowed to terminate) then it has to upgrade the local version of p. However, by the wait condition in Rule 3, it will be allowed to do so only *after* computation k_1 which has an older version of p will terminate. Essentially, by Rule 1 and the wait condition in Rule 3, we ensure that the order of upgrading local versions of shared microprotocols by concurrent computations in Step 3, is always the same as the order of increasing global versions by these computations in Step 1, which is the necessary correctness condition for isolation provided by version-based concurrency control. The rest of the proof is by induction on computations. □

5.2 Version-Counting with Least-Upper-Bound

The Version-Counting with Least-Upper-Bound Algorithm (VCA_{bound}) requires to know the *least upper bound* (supremum) on the number of times each microprotocol's object (that may be visited by a computation) can be visited by the computation. This information allows the algorithm to decide if a given microprotocol p which has been visited by a computation k is not going to be revisited by k. If so then the microprotocol's local version can be safely upgraded. After supremum is reached, any computation that wants to call any handler of microprotocol p and holds a winning private version, will be allowed to call the handler, and proceed concurrently with k, thereby enabling more parallelism than in the case of VCA_{basic}, where computation k must firstly complete.

The VCA_{bound} algorithm is the same as VCA_{basic} in Section 5.1, except that Rules 1, 2, and 3 are modified and a new atomic Rule 4 is added:

1. As Rule 1 of VCA_{basic} but increment counter gv_p by $bound[p]_k$, which is the least upper bound of times microprotocol p can be visited by computation k.

2. Replace (1) by
$$pv[p]_k - bound[p]_k \leq lv_p < pv[p]_k \ . \qquad (2)$$

3. After a computation k has completed its execution (i.e. all threads of the computation terminated), check if there are any local versions lv_p of microprotocols $p \in M$ that need to be upgraded, i.e. $lv_p < pv[p]_k$; if so then for each such a microprotocol p, wait until (2) is true, and then upgrade the local version of p, so that we have $lv_p = pv[p]_k$; in the end, erase map pv_k.

4. Each time the execution of a handler h of microprotocol p has been completed by some computation k (i.e. the handler's main method returned and any threads spawned by the handler terminated), the microprotocol's local counter lv_p is incremented by one.

Proof (sketch) The proof of Lemma 1 for the VCA_{bound} algorithm is similar to the previous proof. The main difference is that computation k_2 is allowed to call a handler of microprotocol p soon after the local version of the microprotocol lv_p is equal $bound[p]_{k_1}$, that is either after k_1 has called p the number of times declared in Rule 1 by $bound[p]_{k_1}$, or by Rule 3 after k_1 has terminated (in the case when k_1 visited p less times than declared). Note that if the least upper bound was actually too small, and k_1 would try to visit p more times than it has declared, then by Rules 1 and 4, lv_p will be at least equal or greater than $pv[p]_{k_1}$, and so by Rule 2, k_1 is not allowed to call any handler of p since $lv_p \not< pv[p]_{k_1}$. Rule 3, applied after k_1's termination, takes care that any local version lv_p such as $lv_p > pv[p]_{k_1}$, i.e. upgraded by some other computations than k_1 after k_1's supremum is reached, is never *downgraded* by Rule 3. Moreover, as in the previous proof, by Rule 1 and the wait condition in Rule 3, the order of upgrading local versions of shared microprotocols by concurrent computations in Step 3, is always the same as the order of increasing global versions by these computations in Step 1, which is the necessary correctness condition for isolation provided by version-based concurrency control. □

5.3 Version-Counting with Routing Pattern

The Version-Counting with Routing Pattern Algorithm (VCA_{route}) requires an *event routing pattern* to be specified; the pattern declares not only the microprotocol objects that can be possibly visited by a given computation, but also which handlers of these microprotocols may be called and in which order. The pattern is represented as a directed graph of handler names; an arrow $h_1 \mapsto h_2$ in this graph declares that the body of handler h_1 may call h_2.

The algorithm is the same as VCA_{basic} in Section 5.1, except that: (*i*) M is now a value declaring the routing pattern of computation k, (*ii*) all handlers maintain their status, which is equal either "active" or "inactive" (initially all handlers are "inactive"); and finally, (*iii*) the meaning of "$p \in M$" is that p is a vertex in graph M, and (*iv*) Rules 2 and 3 are modified and a new atomic Rule 4, which uses the routing information, is added:

2. As Rule 2 of VCA_{basic} but to call a handler h, condition (1) (see Section 5.1) must hold *and* either h is a handler to be called directly by expression e, or there is a route (path) to h in graph M from the handler that tries to call h. Execution of a call of handler h changes the status of the handler to "active" (to satisfy Rule 4, the handler making the operation must not be allowed to complete before this change comes into effect).

3. As Rule 3 of VCA_{bound}.

4. Each time the execution of a handler h of microprotocol p has been completed by some computation k, execute the following procedure (in the specified order):

    (a) change the status of handler h in M to "inactive",

    (b) for each microprotocol p that has *all* its handlers "inactive" *and* not reachable from handlers which are currently "active" (where "not reachable" means that there is no route in M leading to each such handler from any "active" handler), remove p's handlers from graph M, and upgrade a local version of p, so that $lv_p = pv[p]_k$.

Proof (sketch) The proof of Lemma 1 for the VCA_{route} algorithm is similar to the proof of VCA_{basic}. The main difference is that computation k_2 is allowed to call a handler of microprotocol p soon after the local version of the microprotocol lv_p is equal $pv[p]_{k_1}$, that is either after p is not reachable in the graph maintained by k_1 anymore (after making modifications to the graph by Rule 4(b)), or by Rule 3 after k_1 has terminated (in the case when cycles in k_1's graph prevent the algorithm to decide about handlers reachability). Note that if the routing pattern got by k omitted some routes, then by Rule 2, k is not allowed to call any handlers that are not accessible. However, there is no problem if the pattern provides routes to some handlers that are never called by k, since by Rule 3, these handlers will be released after k has completed. Applying Rule 3 as in VCA_{bound} ensures that versions are never downgraded.

By Rule 1 of VCA_{basic} and Rule 3 of VCA_{bound}, the order of upgrading local versions of shared microprotocols by concurrent computations in Step 3, is the same as the order of increasing global versions by these computations in Step 1, which is the necessary correctness condition for isolation provided by version-based concurrency control. □

6 Related Work

The work in this paper builds on research in two areas: the design and implementation of programming language features for concurrency and transactional properties, and the construction of concurrency control algorithms.

Language Support for Isolation. Actions that guarantee only some of the ACID properties of transactions (i.e. a combination of Atomicity, Consistency, Isolation, and Durability) are becoming ever more important in applications servers and transaction systems. Different forms of transactions decomposed to satisfy only individual properties appeared in distributed operating systems, such as Camelot [4], in modern transactional platforms, e.g. IBM WebSphere [13], Microsoft MTS [16], and SunEJB [21], as well as programming languages like Arjuna [18] and ML [23]. Concurrency control in the Camelot system was factored out into a separate mechanism that the programmer could use to ensure isolation. In [7], the authors propose higher-order functions for clean (de)composability of transactional features in ML, avoiding the need to have block-structured constructs to delimit a transaction's boundary.

While our programming construct can allow multi-threaded sections of code to be executed in isolation, several researchers have proposed programming language features for *atomicity* of sequential actions executed by a thread. In [5], Flanagan and Qadeer present a type system for specifying and statically verifying the atomicity of methods in multithreaded Java programs. The type system allows methods to be annotated with the keyword `atomic`. If the program type checks, then any interaction between an atomic method executed by a thread and steps of other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behaviour. More recently, Harris and Fraser have been investigating a similar `atomic` construct in Java [8]. Their proposal implements Hoare's conditional critical regions (CCRs) [10]. The programmer can guard the region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. The implementation is based on mapping CCRs onto a *software transactional memory* which groups together series of memory accesses and makes them appear atomic.

Concurrency Control Algorithms. Research on transaction management began appearing in the early to mid 1970s. Quite a large number of concurrency control algorithms have been proposed for use in centralised and distributed database systems. Database systems use concurrency control to avoid interference between concurrent transactions that can lead to an inconsistent database. Isolation is used as the definition of correctness for concurrency control algorithms in these systems. The algorithms generally fall into one of three basic classes: *locking* algorithms, *timestamp* algorithms, and *optimistic* (or certification) algorithms. A comprehensive study of example techniques with pointers to literature can be found in [2]. Concurrency control problems had been also treated in the context of operating systems beginning in the mid 1960s. Most textbooks on operating systems survey this work, see e.g. [20, 22].

Our versioning algorithms have some resemblance with basic two-phase locking. However, instead of acquiring all locks needed (in the 1st phase) and releasing them (in the 2nd phase), a computation takes and dynamically upgrades version numbers. Execution of the algorithms by the runtime system orders conflicting operations of computations (i.e. handler calls) according to version numbers, like in the timestamp algorithms. However, we associate versions with handler calls, not with transactions. Therefore all calls can be made in the right order for the isolation property (the call requests with too high versions are simply delayed), unlike basic timestamp algorithms for transactions, where if an operation has arrived too late (that is it arrives after the transaction scheduler has already output some conflicting operation), the transaction must abort and be rolled back. The "ultimate conservative" timestamp algorithms avoid aborting by scheduling all operations in timestamp order, however, they produce serial executions (except complex variants that use transaction classes) [2].

Methods of *deadlock avoidance* in allocating resources [20, 22] are also relevant to our work. The *banker's algorithm* (introduced by Dijkstra [3]) considers each request by a process as it occurs, and assigns the requested resource to the process only if there is a guarantee that this will leave the system in a safe state, that is no deadlock can occur. Otherwise the process must wait until some other process releases enough resources. The *resource-allocation graph* [11] algorithm makes allocation decisions using a directed graph that dynamically records claims, requests and allocations of resources by processes. The request can be granted only if the graph's transformation does not result in a cycle. Resources must be claimed a priori in these algorithms.

In our case, a computation must also know a priori all its resources (handlers or microprotocols) before it can commence. However, the history of handler calls by different computations is always acyclic since versions impose a total order on call requests performed by different computations. Since computations are assumed to complete, old versions will be eventually upgraded. Therefore our versioning algorithms are deadlock-free. Moreover, the calls are assigned according to the order that is necessary to satisfy the isolation property, unlike the resource allocation algorithms, which do not deal with ordering of operations on resources.

7 Future Work

To test our framework, we have expressed in SAMOA the Atomic Broadcast protocol, sketched in Section 3, and executed it on distributed machines. We tried variants of the concurrency control with a different grain of concurrent execution among computations. Our preliminary results are encouraging; the overhead incurred by SAMOA's concurrency control algorithms while executing our example protocol is relatively low. We plan to make more experiments, and also test our framework using parallel processor architectures.

A possible avenue for further development of our framework is to introduce different types of handlers (e.g. read-only, read-and-write) and several levels of isolation, following a similar practice in database systems [14]. The isolation requirements can be sometimes relaxed for certain computations to yield better application performance. A non-trivial extension of our versioning algorithms will be to add support for dynamic binding of handlers to event types.

Acknowledgments. We would like to thank the anonymous reviewers for the feedback that they provided on the ideas in this paper.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, 2000.

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

[3] E. W. Dijkstra. Cooperating sequential processes. Technical report, Technological University, Eindhoven, the Netherlands, 1965.

[4] J. Eppinger, L. Mummeft, and A. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[5] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. PLDI 2003*, June 2003.

[6] *The Crystall Project: Correct Modular Group Communication Middleware*. http://lsrwww.epfl.ch/crystall.

[7] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM TOPLAS*, 16(6):1719–1736, Nov. 1994.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. OOPSLA '03*, Oct. 2003.

[9] M. Hayden. The Ensemble system. Tech. Report TR98-1662, Dep. of Comp. Science, Cornell University, Jan. 1998.

[10] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71, 1972.

[11] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[12] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[13] IBM. *WebSpheres*. http://www-3.ibm.com/software/info1 /websphere/index.jsp.

[14] T. Kempster, C. Stirling, and P. Thanisch. Diluting ACID. *SIGMOD Record*, 28(4):17–23, 1999.

[15] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proc. of Middleware 2003*, LNCS 2672, June 2003.

[16] Microsoft. *Microsoft MTS*. http://www.microsoft.com /com/tech/MTS.asp.

[17] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. ICDCS 2001*, Apr. 2001.

[18] G. Parrington and S. Shrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proc. ECOOP 1988*, LNCS 322, Aug. 1988.

[19] *The SAMOA Protocol Framework*. http://lsrwww.epfl.ch/ samoa.

[20] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, Sixth Edition*. John Wiley & Sons, Inc, 2002.

[21] Sun. *EJB*. http://java.sun.com/products/ejb.

[22] A. S. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice Hall, Englewood Cliff, NJ, 2001.

[23] J. M. Wing, M. Faehndrich, J. G. Morrisett, and S. Nettles. Extensions to Standard ML to support transactions. In *Proc. of Workshop on ML and its Applications*, 1992.

[24] P. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In *Proc. of Coordination 2002*, LNCS 2315, Apr. 2002.

[25] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *INFOCOM '01*, Apr. 2001.