Semantics of Protocol Modules Composition and Interaction

Paweł T. Wojciechowski, Sergio Mena, and André Schiper

EPFL, School of Computer and Communication Sciences, 1015 Lausanne, Switzerland
{First.Last}@epfl.ch

Abstract. This paper studies the semantics of protocol modules composition and interaction in configurable communication systems. We present a semantic model describing Cactus and Appia — two frameworks that are used for implementing modular systems. The model covers protocol graph, session and channel creation, and inter-module communication of events and messages. To build the model, we defined a source-code-validated specification of a large fragment of the programming interface provided by the frameworks; we developed an operational semantics describing the behaviour of the operations through state transitions, making explicit interactions between modules. Developing the model and a small example implementing a configurable multicast helped us to better understand the design choices in these frameworks. The work reported in this paper is our first step towards reasoning about systems composed from collections of modules.

1 Introduction

Modularization is a well-known technique for simplifying complex communication systems. Here, we describe an approach which is based on implementing an application's individual properties as separate protocols, and then combining selected protocols using a software framework. This approach helps to clarify the dependencies among properties required by a given communication system, and makes it possible to construct systems that are customized to the specific needs of the application or underlying network environment. We are particularly interested in implementations of group communication infrastructure (or middleware), as configurability of protocols should be clearly required here; for example, different applications may demand very different properties and guarantees as far as the quality of service and failure semantics are concerned.

In this paper, we are primarily interested in the programming abstractions provided by Cactus [10] (which subsumes the x-kernel model [3]), and Appia [4]. We have described an operational semantics of the programming interface offered by each framework, covering enough abstractions for expressing interactions between modules composed into a protocol graph. The frameworks also support primitives that can simplify the construction of protocols, such as support for processing messages, marshalling messages to the network format, and

F. Arbab and C. Talcott (Eds.): COORDINATION 2002, LNCS 2315, pp. 389–404, 2002.

timeouts, but they are not covered here. We illustrate the model with a small program, implemented in Cactus and Appia.

We have chosen Cactus and Appia for two reasons. Firstly, each of the frameworks implements a very different approach to building configurable software, with a different range of programming abstractions. Therefore, it is interesting to look at each framework in turn. More importantly, we want a model that is general enough for building any kind of communication service, not just group communication. For example, Cactus has been used to implement many configurable protocols and services in distributed systems, such as Group RPC, real-time channels, secure communication service, and QoS components for CORBA. Appia has been used for the development of group communication and real-time protocols. On the other hand, systems such as Horus and Ensemble [1] have been designed to support modular and reconfigurable group communication, however the protocol stack can only be configured from selected protocols that use predefined event types.

The frameworks for building configurable services are highly concurrent with complex programming interfaces. This complexity makes it hard to achieve a clear understanding of the framework's behaviour based only on informal descriptions, in turn making it hard to build robust configurable systems. To the best of our knowledge, there exist only informal natural-language documents describing Cactus and Appia, covering the general architecture of each design and the programming interface but not precise enough or free from ambiguities; for example, we had to inspect source code on several occasions since the documentation was not clear enough.

Our work aims at precise understanding of the behaviour of programs that are implemented using these frameworks. An important question is about the sense in which the semantics of network subsystems composed from collections of protocol modules will relate to the behaviour of the actual implementation. This is an area that is often a secondary priority for the developers of practical module composition frameworks, yet is crucial to the long-term acceptance of this approach. While the work described here has not quite reached the point of reasoning about composition, it makes the important first step in this direction.

2 Architecture

Figure 1 presents the architecture of Cactus and Appia. The protocol names are taken from our small example, which is described in Section 5. It can be noticed that the frameworks differ considerably in the way protocol modules (represented by boxes) are composed. The dynamically created instances of modules are called *sessions*. The protocol sessions (ovals) communicate using messages or events, which are sent along communication paths (arrows).

The composition of protocols is defined by a directed acyclic graph, in which edges define communication channels. The protocols in Cactus communicating using messages are internally structured as collections of microprotocols. The microprotocols communicate using events and shared data such as messages, with

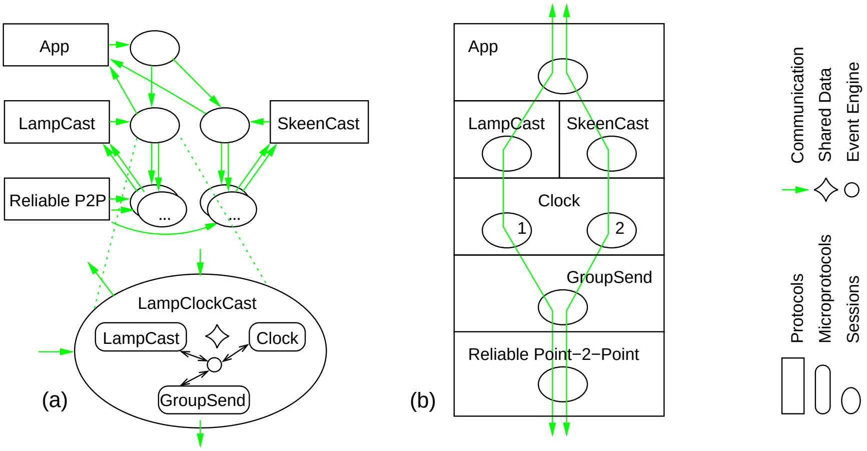

Fig. 1. Example Protocol Composition in (a) Cactus, and (b) Appia

the events dispatching actions defined by event handlers; messages coming from outside the protocol session normally trigger an event. A protocol in Cactus can create a new session dynamically, e.g. when a message arrives from a new participant. In Appia, all sessions must be created before a relevant communication channel is established (usually at the time when a protocol stack is configured).

The sessions (or protocols) in Cactus decide themselves which other sessions are to receive a message — in the case of messages incoming from the network this information is usually extracted from the message. The message is forwarded to a next session by invoking an interface method. The message arrival to a session causes appropriate event handlers (within the session) to be invoked. In Appia, there is a scheduler which forwards events to sessions in the order which is defined by a communication channel; the channel name is extracted from the event.

In the following sections, we describe each framework in turn, giving semantics of the most important operations. We do not require from the reader knowledge of the formal semantics methods, but instead we use algebraic objects that should be also well known for non-theoreticians, such as sets, lists, tuples, maps, and relations. We introduce our notation when it is first used. Due to lack of space, some rules have been omitted (they are included in [8]).

3 Cactus

Cactus extends the x-kernel hierarchical composition of protocols with fine-grain parallel composition. The Cactus protocols can be composed of semi-independent microprotocols, each of which implements a well-defined property or function of the protocol. Below we focus on Cactus/J, which is one of the prototype implementations of Cactus.

3.1 Microprotocols and Events

A *microprotocol* is a section of code, structured as a collection of *event handlers*, where an event handler is simply a procedure invoked with every occurrence of the event. We define *Microprotocol* as the set of microprotocol names, ranged over by x, y. The set *Handler* is the set of event handler names, ranged over by h. An *event* defines an occurrence that causes one or more microprotocols to be executed. For example, an event such as message arrival might trigger the event handlers of a microprotocol which detects host failures, and a microprotocol which is responsible for message ordering, etc. The events not only drive the flow of control, by executing event handler procedures associated with a given event, but also pass data from the trigger point to the handler. The set *Event* is the set of valid event names (or types), ranged over by e, e'. We denote the *occurrence* of an event e as a triple (x, e, v), where x is the caller which raised event e, and v is a value passed with the event.

In order to describe behaviour of the operations supported by the programming interface (represented as functions), we use a transition relation of the form $S, p \vdash op(n) \triangleright S'$, which means that the execution of operation op initiated or invoked by p in some state (or context) S leads to state S'; op has parameters n. The state is represented by relevant set(s) of elements. In our case, the context of every transition relation is always a single protocol stack, i.e. S always describes (part of) the state of a local runtime system only. To express and maintain the internal state, we will need maps storing bindings from keywords of type $\mathcal{T}$ to values of type $\mathcal{T}'$. We represent maps using a set $\mathcal{M}(\mathcal{T} \mapsto \mathcal{T}')$ of all mappings from elements in $\mathcal{T}$ to elements in $\mathcal{T}'$, together with operations for adding and removing bindings from a map, and looking up an element. We also use lists of elements in $\mathcal{T}$; $\mathcal{L}(\mathcal{T})$ is to denote all possible lists of elements such that each element is in $\mathcal{T}$.

In order to associate a handler with a particular event, a microprotocol invokes an operation

$$\mathsf{bind} : \mathit{Event} \times \mathit{Handler} \times \mathit{Int} \to ()$$

specifying the event name, the handler name, an integer which is used to determine an *order* in which handlers will be executed, and a static argument (omitted here) which is passed to the handler when an event occurs (this can be used to parameterize a handler and allow its use with more than one event types). Below a microprotocol x binds an event handler h to event e.

$$\frac{hl = E[e]}{E, x \vdash \mathsf{bind}(e, h, i) \triangleright E \oplus (e \mapsto \mathit{sort}_{\leq}((h, i) :: hl))}$$

This registers a handler h of event e in a map $E \in \mathcal{M}(\mathit{Event} \mapsto \mathcal{L}(\mathit{Handler} \times \mathit{Int}))$, which is part of the Cactus/J state of a composite protocol that contains x. The map E stores bindings from an event name to a list of event handlers which are to handle the event, where each handler name is paired with the order argument i; the list is ordered with increased i. The value i does not need to be unique for each handler; handler names with the same order argument are placed in an indeterminate order. With every occurrence of e, the handlers will be executed in sequence as they appear in the list $E[e]$. We use the following notation: $E[e]$ looks up e in map E and returns the list of handlers bound to e, $::$ is a concatenation symbol to append a new element to a list, $sort_{\leq}(l)$ returns a list l sorted by partial order relation $\leq$, and $E \oplus (e \mapsto l)$ returns map E with a new binding of e to l; if e was already bound in E, its previous binding disappears.

3.2 Event Raising and Handling

An event can be raised by calling

$$\mathsf{raise} : \mathit{Event} \times \{\textsc{Sync}, \textsc{Async}\} \times \mathcal{T} \to ()$$

specifying the name of the event, the calling mode, and a dynamic argument, such as a message that is associated with the event. When an event is raised, handlers bound to this event execute sequentially in the specified order. Each handler is passed both the static argument defined at binding time and the dynamic argument. The calling mode μ is either Sync, which invokes handlers immediately and blocks the caller until the last handler is executed, or Async, which allows the caller to proceed concurrently with the handlers (the handlers can be executed after a specified delay, omitted here).

Below we define the behaviour of raise, assuming that a microprotocol x raises an event e with a dynamic argument v.

$$\boxed{\mathsf{raise}(e,\mu,v)\text{:}}$$

$$\frac{\begin{array}{l}\mu = \textsc{Sync} \lor \mu = \textsc{Async} \\ \qquad\qquad\qquad E[e] = (h_1, \_) :: ... :: (h_n, \_) :: \mathit{nil}\end{array}}{E, x \vdash \mathit{invoke}(h_1, v), ..., \mathit{invoke}(h_n, v) \rhd E \land (x, e, v)_\mu \ \mathit{raised}}$$

This looks up in map E a list of handlers of event e and executes the handlers, passing v to each handler. The event raising is modelled by relation *raised*.

The caller x is either blocked until the last handler returns, or not, depending on mode μ. While the handlers of a particular event occurrence execute sequentially, it is important to note that they can execute concurrently with other occurrences of the same event or with other microprotocol code. Therefore access to any shared data should be synchronised.

3.3 Messages

Protocols in Cactus communicate using *messages*; a message is created by the application or a protocol session, and can travel through several layers of protocol sessions and across a network. Messages contain data stored in *attributes*, which can be accessed and modified by microprotocols. Message creation raises a predefined event *NewMessage*.

Below, we use the set *Session* of session names, ranged over by s, and the set *Message* of message names (or references), ranged over by m, n. A message m is modelled as a triple of message attributes a, a message type T, and send votes V^m (the last two parameters are local to a session and never transmitted), i.e.

$$m = (a, T, V^m)$$

where each named attribute in a record a has a defined scope; it can be visible only in the current session, in the current stack, or within peer sessions only; otherwise it is discarded or concealed. The message type T is equal ↑ if the message arrived from a session below, or ↓ if from above. The message type ⋄ is for a temporary message local to a session. The send votes are described in §3.4.

A protocol can send a message to a Cactus session below or above using

$$\begin{aligned} \mathsf{sendDown} &: [\mathit{Session}\times]\mathit{Message} \rightarrow () \\ \mathsf{sendUp} &: [\mathit{Session}\times]\mathit{Message} \rightarrow () \end{aligned}$$

where the first (optional) parameter is the name of a session to which the message is to be sent. If a message is sent to a non-Cactus session, e.g. to an x-kernel session, message attributes are converted into message headers by using a user-defined procedure (see also the push and demux operations in [8]). Below we define the default behaviour of sendDown, assuming that a protocol session s sends a message m downward to a session s', created by Cactus/J.

$$s \vdash \boxed{\mathsf{sendDown}(s', m)\text{:}}$$

$$\textsc{down}\quad \frac{s \vdash s'.\mathsf{fromAbove}(m)}{M, s' \vdash \mathsf{raise}(\mathit{MsgFromUser}, \textsc{Async}, m) \triangleright M \oplus (m \mapsto s')}$$

The session s invokes an operation fromAbove of the lower-level session s', passing m as the parameter. The execution of fromAbove raises asynchronously an event *MsgFromUser* which carries the message m. The message migration is recorded in map $M \in \mathcal{M}(\mathit{Message} \mapsto \mathit{Session})$ of active messages bound to their current sessions. Microprotocols which are interested in receiving messages from sessions above could handle the *MsgFromUser* event. Note, however, that any subsequent invocation of sendDown will also raise this event. Therefore, if the protocol requires to receive messages in a first-in-first-out order, some synchronisation is necessary so that the microprotocol handlers will be invoked in a sequence (e.g., in our example program, we have overwritten the operation fromAbove so that it executes a synchronous operation $\mathsf{raise}(\mathit{MsgFromUser}, \textsc{Sync}, m)$).

The semantics of message flow in the opposite direction is similar. The main difference is that sendUp calls either fromBelow of a specified higher-level session (inside which an event *MsgFromNet* is raised), or an operation demux of a higher level protocol, if no session has been specified.

3.4 Message Events

An event can be associated with a particular message type (↓,↑). This event is triggered by a collective action of all microprotocols that have registered an interest, providing a way for microprotocols to agree upon event raising. To declare the interest, a microprotocol invokes

$$\mathsf{register} : \{\downarrow,\uparrow\} \times \mathit{Event} \to ()$$

passing a message type and an event name. Every subsequent message creation of that type has the potential of triggering the event. (If the event is to be caught, a bind call is also necessary.)

$$\frac{}{R_e^T, x \vdash \mathsf{register}(T,e) \triangleright R_e^T \oplus (x \mapsto \mathit{false}) \wedge (x,e,T)\ \mathit{registered}}$$

The invocation of register adds a new entry in a map R_e^T for message type T and event e. We mark registration by relation *registered*. The map $R_e^T \in \mathcal{M}(\mathit{Microprotocol} \mapsto \mathit{Boolean})$ is created dynamically and updated each time when some microprotocol executes operation register; it is a map from names of microprotocols to Boolean values (initially *false*) that represent the microprotocol "votes" signalling readiness of the event e to be raised for message type T (where T not equal $\diamond$).

For each message m, whenever message type T is assigned, Cactus/J uses maps R_e^T to build a (local to m) map V^m. For each event e that has been associated with the message type T, map V^m stores a copy of corresponding map R_e^T, i.e. $V^m[e] = \mathit{copyOf}(R_e^T)$. Each event e can be raised only once per message; that occurrence of e will pass name m to event handlers bound to e.

For each message, the message event is raised as soon as all of the interested microprotocols have called

$$\mathsf{signal} : \mathit{Message} \times \mathit{Event} \to ()\ .$$

The signal operation requires to pass as arguments the names m of the message and e of the event which will carry the message. The behaviour of signal invoked by microprotocol x is below; the execution of signal should be atomic.

$$\frac{\begin{array}{l} m = (a,T,V^m) \\ \qquad\qquad V^m[e][x] = \mathit{false} \wedge \forall y \neq x\ V^m[e][y] = \mathit{true} \end{array}}{E, V^m[e], x \vdash \mathsf{signal}(m,e) \triangleright E, V^m[e] \oplus (x \mapsto \mathit{true}) \wedge (x,e,m)_{\mathrm{ASYNC}}\ \mathit{raised}} \quad (1)$$

$$\frac{\begin{array}{l} m = (a,T,V^m) \\ \qquad\qquad \exists y \neq x \mid V^m[e][y] = \mathit{false} \end{array}}{E, V^m[e], x \vdash \mathsf{signal}(m,e) \triangleright E, V^m[e] \oplus (x \mapsto \mathit{true}) \wedge (x,e,m)\ \mathit{signalled}} \quad (2)$$

Rule (1) checks if x signals e for the first time and if all other microprotocols set their "vote" to raise event e associated with the message. If so, the event is raised asynchronously and all event handlers which have been bound to this event will receive the name of the message (see §3.2 for details). Otherwise (2), event e cannot be raised and we only set in V^m the message readiness as far as microprotocol x is concerned (and mark that the relation *signalled* holds).

For example, we can use this mechanism to implement a collective sending by several microprotocols. Below, we have two microprotocols x and y which share a message m and want to agree when to invoke an event carrying this message.

$$\frac{\begin{array}{l}(x, e, \downarrow)\ \textit{registered} \\ (y, e, \downarrow)\ \textit{registered} \\ m = (a, \downarrow, V^m) \\ \qquad\qquad (y, e, m)\ \textit{signalled}\end{array}}{E, x \vdash \mathsf{signal}(m, e) \triangleright E \wedge (x, e, m)_{\text{ASYNC}}\ \textit{raised}}$$

We assume that microprotocols x and y registered their interest in raising an event e when a message of type $\downarrow$ will be received from above by the composite protocol. We also assume that some message m of this type eventually appeared and was handled and signalled by microprotocols x and y. Since x is the last microprotocol which signalled the readiness of message m, therefore it causes event e to be raised. A microprotocol (more precisely one of its event handlers) which has been bound to event e can now be invoked and, e.g., it might send the message out of the composite protocol.

4 Appia

A *protocol* in Appia consists of two static parts, one is called *layer* and the other one is called *session* (not to be confused with a session in Cactus). Protocols interact using one or more coordinated channels. A *channel* defines routing of events across protocols, and is defined by a set of instances of sessions (i.e. objects of class "Session").

4.1 Layers and Sessions

A *layer* declares types of events which are either generated, required, or accepted by the protocol. Appia uses the event declarations to verify partial correctness of QoS definitions (we describe this verification below). A layer is also used to create instances of its session. A *session* implements the actual protocol code, in particular it generates and handles events which have been declared by the corresponding layer. An event may carry a message. Messages can be marshalled and communicated in a network.

The set *Layer* is a set of layer names, ranged over by l. The set *Session* is a set of session names, ranged over by s. The name of a layer identifies unambiguously a protocol whose definition the layer is part of (so we may sometimes use terms "layer" and "protocol" interchangeably). A layer can use an operation

$$\mathsf{createSession} : \textit{Layer} \rightarrow \textit{Session}$$

to create many instances of its session (the name of the layer is passed as the operation argument).

Below we use a set $P \in \mathcal{S}(\mathit{Layer})$ of names of all protocols/layers which are used to form a given protocol stack ($\mathcal{S}(\mathit{Layer})$ denotes all possible subsets of the set *Layer*, i.e. $\mathcal{S}(\mathcal{T})$ is the powerset of $\mathcal{T}$, usually denoted $2^{\mathcal{T}}$). In the context of P, we define the following three maps E_{g}, E_{r}, and E_{a}, which store bindings from layer names to, respectively, a set of types of events which are generated by a protocol, types of events which are required by the protocol, and types of events which are accepted by the protocol (that includes the former set), i.e.

$$E_{\mathrm{g}}, E_{\mathrm{r}}, E_{\mathrm{a}} \in \mathcal{M}(P \mapsto \mathcal{S}(\mathit{EventType})), \quad \forall l \in P \; E_{\mathrm{r}}[l] \subseteq E_{\mathrm{a}}[l]$$

where *EventType* is a set of abstract event types. A protocol l declares some event type T to be in $E_{\mathrm{a}}[l]$ but not in $E_{\mathrm{r}}[l]$ if the absence of events of this type is not critical for the protocol execution; therefore we could use protocol l to build protocol stacks which are meaningful even if events of type T are never generated in these stacks.

The Appia state contains set P of layers which are used to form a single protocol stack, together with a map $S \in \mathcal{M}(P \mapsto \mathcal{S}(\mathit{Session}))$ from layer names to sessions created by the layers. New sessions are created as follows.

$$\frac{l \in P}{P, S, l \vdash s := \mathsf{createSession}(l) \triangleright P, S \oplus (l \mapsto S[l] \cup \{s\})}$$

This transforms the state at a time when the protocol stack is initiated, recording a new session s created by layer l in map S.

4.2 QoS Definitions and Channels

A *QoS definition* is simply a static list of layers, which is used to create a communication *channel*. The Appia framework partially verifies each QoS definition, checking if events that the layers declared as required are also declared as generated. The verified QoS definition is used to build a channel with blank slots; the slots can be filled as appropriate with sessions that are created by the layers.

A channel defines the flow of events through the sessions. Each channel maps layers from the QoS definition into concrete sessions which have been created by the layers. By selecting appropriate channels for routing different events through the protocol stack, an application can obtain a requested *quality of service* (QoS).

We model a QoS definition as a list of names of layers which are used to build a single protocol stack. A QoS definition $\mathit{qos} \in \mathcal{L}(P)$ constructed using protocols from P is *well formed* if for each event type T required by each protocol l from set L (of all elements from list *qos*) there exists some protocol l' in L which generates event e. This verification is usually done before any session is created. The set $Q \in \mathcal{S}(\mathcal{L}(P))$ of QoS definitions, such that each definition is well formed can be used by Appia to create channels.

A set $\mathit{Channel} = \mathcal{L}(\mathit{Session} \times \mathit{Layer})$ is a set of channels, ranged over by c. Let $C \in \mathcal{M}(\mathit{Id} \mapsto \mathit{Channel})$ be a mapping from channel identifiers to channels in a given protocol stack, where a single channel c in map C is modelled as a list of session names paired with names of the corresponding layers in the protocol stack, i.e.

$$c = (s, l) :: t \quad \text{where} \;\; l \in P, \;\; s \in S[l] \;\; .$$

The channel identifiers are unique per protocol stack; they are used by messages to identify a (corresponding) channel on a remote site that should be chosen to deliver the messages to peers. Here is how a new channel is created and bound to sessions (first by user-defined binding and then automatic binding).

$$\frac{c := \mathsf{createUnboundChannel}(ID, qos) \wedge qos \;\; \textit{well-formed}}{P, S, C \vdash c = \textit{defaultBind}(\textit{userBind}(c)) \triangleright P, S, C \oplus (ID \mapsto c)}$$

This first creates a new channel c from a well formed QoS definition qos using an Appia operation $\mathsf{createUnboundChannel} : \textit{Id} \times \mathcal{L}(\textit{Layer}) \rightarrow \textit{Channel}$. The channel is identified by a fresh name $ID \in \textit{Id}$. The new channel is initially *unbound*, i.e. each element (s, l) of c has a session name s equal *null*. After the channel is filled with sessions, a mapping of ID to the channel is recorded in map C.

In order to bind the free slots of an unbound channel to sessions that are created by corresponding layers (of the qos definition), the following two procedures are used. The first procedure must be set up by the protocol programmer, who can specify in this way which channels should share a common session.

$$\boxed{\textit{userBind}(c):}$$

$$c = (\textit{null}, l) :: t \quad \text{where} \;\; l \in P$$

$$\frac{\exists s \in S[l] \mid s \;\; \textit{required-by} \;\; c}{P, S, C \vdash \textit{return}((s, l) :: \textit{userBind}(t)) \triangleright P, S, C}$$

This binds free slots in channel c to some existing sessions s, which are selected by a programmer from set $S[l]$. They have been created before with $\mathsf{createSession}$. The sessions s are likely to be bound already to some other channels, so that they can process different types of events which originate from different channels. The choice of sessions is application-dependent; here modelled by relation *required-by*. If the relation does not hold, *null* slot is left.

The free slots that have not been bound explicitly by *userBind* are bound automatically by a default procedure below.

$$\boxed{\textit{defaultBind}(c):}$$

$$c = (\textit{null}, l) :: t \quad \text{where} \;\; l \in P$$

$$\frac{s := \mathsf{createSession}(l)}{P, S, C \vdash \textit{return}((s, l) :: \textit{defaultBind}(t)) \triangleright P, S \oplus (l \mapsto S[l] \cup \{s\}), C} \quad (1)$$

$$\frac{c = (s, l) :: t \wedge s \neq \textit{null}}{P, S, C \vdash \textit{return}((s, l) :: \textit{defaultBind}(t)) \triangleright P, S, C} \quad (2)$$

This creates a new session s for each session-free layer l in a channel c and returns the channel with free slots filled with the session names.

A *protocol stack* is a composition of all protocols that share (transitively) some communication channels. We define F to be a *well formed* set of channels where well-formedness means that each channel in F (built from a well-formed QoS definition) shares at least one session (selected by the user) with some other channel in the protocol stack. We represent a protocol stack as a map C from channel identifiers to channels which are taken from set F.

4.3 Routing Table

After channels have been created, Appia can use information about the channels and events declared by protocols to construct an optimal routing path for each event type that is associated with a given channel.

We model a *routing table* as a map $R \in \mathcal{M}(\mathit{Id} \times \mathit{EventType} \mapsto \mathcal{L}(\mathit{Session}))$ from channel identifiers paired with types of events to routing paths, where a *routing path* is a list of sessions (ordered from top to bottom) which accept these events. A session *accepts* an event of type T if the session was created by a protocol which declared T in its set of accepted events (in map E_{a}, which has been defined in §4.1).

The map R is created from *all* routing paths which are well formed. A routing path $r \in \mathcal{L}(\mathit{Session})$ of events of type T that are to travel in a channel identified by ID is *well formed* if r is a list of sessions constructed from a superset of sessions taken from channel c identified by ID, so that each session in r accepts events of type T and the order of sessions in r is the same as order of sessions in c. Routing paths are kept unchanged during system lifetime.

4.4 Events and Messages

Events are the only mean which can be used by protocol sessions (including the application session) to communicate with other sessions in the protocol stack. Messages are specialised events which can be marshalled and sent over network to remote sites; they contain headers with protocol-dependent data. The set *Event* is the set of valid event (and message) names, ranged over by e.

An *event* (or *message*) $e \in \mathit{Event}$ is represented as a tuple (T, ID, r, n), where T is the event type, ID is the name of the channel carrying events of type T, r is the list of sessions to be visited by e (which is built from the channel), and n is the event content. We say that a channel l *carries* (or *accepts*) events of type T if the QoS definition used to create the channel contains at least one layer l, such that $T \in E_{\mathrm{a}}[l]$. The event content n has two components *attrs* and m (denoted $n = \mathit{attrs} + m$), where *attrs* is the record (with named fields) of event attributes, and m is the list of message headers (attached to e by visited protocols). The m fragment is marshalled and sent over network together with T and ID. If e is not a message then m is empty; if e is a message then two attributes s and d of *attrs* are predefined and should contain the source and destination of the message.

Before a message of type T which arrived from a network can be injected into a local channel c identified by ID, it must be first verified (by a user-defined procedure) and then "wrapped" by one of the event tuples below

$$e^{\downarrow} := (T, \mathit{ID}, R[(\mathit{ID}, T)], n)$$
$$e^{\uparrow} := (T, \mathit{ID}, \mathit{reverse}(R[(\mathit{ID}, T)]), n) \ .$$

The event tuples contain local routing data, which is found in R. The routing data will not change during e's lifetime. The choice between tuples $e^{\downarrow}$ and $e^{\uparrow}$ depends on if the event/message uses channel ID to travel downward, or upward ($\mathit{reverse}(l)$ returns a reversed list l). The verification procedure should check if T is accepted by channel ID.

4.5 Event Scheduling and Routing

Below we confuse events and messages for simplicity, and describe the flow of messages in a channel, modelled by modifications to a map of events $E \in \mathcal{M}(\mathit{Session} \times \mathit{Id} \mapsto \mathit{Event})$ from channel sessions to events.

A session s holding an event $e = (T, \mathit{ID}, r, n)$ can pass it along a channel identified by ID by invoking an operation $\mathsf{go}(e)$.

$$\textsc{down/up} \quad \frac{E[(s, \mathit{ID})] = e = (T, \mathit{ID}, r, n)}{C, E, \Phi, s \vdash \mathsf{go}(e) \triangleright C, E \ominus (s, \mathit{ID}), \Phi \cup \{(s, e)\}} \quad (1)$$

This transfers control to a (default or user-defined) scheduler ϕ, modelled as a set Φ of events paired with their last visited session, together with a function *takeEvent*, which returns one element from the set. We record the change of state by modifying the map of events and the scheduler set ($E \ominus e$ returns map E without a binding of e). The scheduler ϕ selects an event e from Φ (the choice depends on the implemented scheduling algorithm), and passes e to the next session to be visited by the event.

$$\textsc{down/up} \quad \frac{\begin{array}{l}(s, e) = \mathit{takeEvent}(\Phi) \\ e = (T, \mathit{ID}, r, n) \\ \qquad\qquad s' = \mathit{next}(s, r)\end{array}}{C, E, \Phi, \phi \vdash s'.\mathsf{handle}(e) \triangleright C, E \oplus ((s', \mathit{ID}) \mapsto e), \Phi \setminus (s, e)} \quad (2)$$

This selects an event e together with its last visited session s from Φ, and uses s to find out which is the next session s' to visit by e according to the routing path r (which has been extracted from the event tuple). It then invokes an operation handle of session s' to handle event e. We record the change of state by modifying a map of events, and removing (s, e) from the scheduler set. The handle operation will recognise a type of e and invoke a user-defined procedure to handle e. For simplicity, we assume in the rule above that a session can only hold one event at a time. The scheduling of events depends on the event scheduler. The default policy is such that each two events which are initially processed by some session in a certain order (e.g. defined when the events are injected into a protocol stack by an application, or received from the network) will never be processed in the opposite order by any other session in the protocol stack. This implies that the whole protocol stack (i.e. all channels) behaves like a distributed queue which holds a first-in-first-out property.

5 Example Protocol Decomposition

To experiment with Appia and Cactus/J we have implemented in each of these frameworks a small example application that uses two communication services. The first service (AB) sends a message atomically to all processes in a distributed system and guarantees Atomic Broadcast. The second service is an Atomic Multicast (AM), which sends only to a specified group of processes. We have decomposed each service into several modules, each implementing a small protocol, so that some modules in the protocol graph can be shared by the two services (in a given system). The modules are presented in Figure 1. The Atomic Broadcast algorithm and pseudocode of an example modular implementation in Appia and Cactus are described in [8]. In Cactus/J, we have decided to place modules *LampCast* and *Clock* in one composite protocol so that they can share a clock variable C_i, which both modules need to read (see Figure 1a); we did the same for modules *SkeenCast* and *Clock* (not shown in Figure). We might experiment with even finer grain protocols; e.g. the *LampCast* module could be further decomposed into two "microprotocols", one for receiving an application message, and the second one for receiving an acknowledgement message.

The clock variable in module *Clock* of the Appia implementation is not shared by other protocols. Therefore, we need to send a specialised event *ClockEvent* (1) in order to propagate the current clock value to *SkeenCast*, each time a new message arrives from the network. Also, we need to send another specialised event *TimeEvent* (2) with a timestamp of the message which is required by *LampCast*. The events are illustrated in Figure 2. Notice that both events must be propagated upward *before* the event of type *Msg* carrying the message (types *AppMsg*, *AckMsg*, and *GroupMsg* are all subtypes of *Msg*). Unfortunately, we cannot pass the clock and timestamp values between modules using network messages since the message headers can only be accessed at the level on which they have been created by a peer participant (e.g. a header which contains the timestamp required by *LampCast* is stripped by layer *Clock*). Also, for sanity reasons, attributes should not be used for this either since, e.g. the current clock value is required only by *SkeenCast* — it does not seem reasonable to extend the message format to include this value because we want to be able to remove or replace module *SkeenCast* at any time, however the format of network messages should not change so often (the same for the timestamp value and *LampCast*).

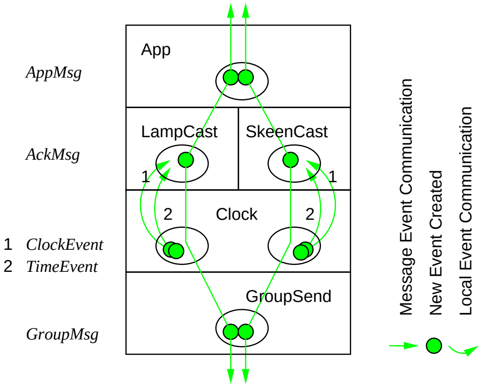

Fig. 2. Example Decomposition in Appia

6 Brief Comparison

Cactus supports fine-grain composition of microprotocols, which communicate using events or shared data. A composite protocol (built from a collection of microprotocols) can also be composed with other (composite) protocols, forming a protocol graph. This two-level architecture allows to decompose a given service in an arbitrary way. Appia offers less flexibility of the composition — modules are composed into a graph, and the pattern of communication between modules is restricted by the communication channels. The channels are static, optimised routing paths in the protocol stack. In Cactus, the idea is that each well-defined property or function of a protocol could be implemented as a microprotocol. However, we need more experience to attain confidence when such fine-grain composition would be justified. In particular, increasing the number of concurrent microprotocols per composite protocol (which have to share resources) may increase the number of mutual dependences, in turn making it harder to notice possible deadlocks.

Appia supports partial evaluation of the protocol composition — for each communication channel it can verify if events declared as required are also generated. This helps to reject protocol compositions which are clearly not meaningful, however, of course it does not guarantee correctness (see [8]). This simple evaluation could be improved if a programmer was able to specify some additional (application-dependent) constraints when defining a module, e.g. a requirement that *all* modules below in the communication channel should declare some event(s) as accepted, or required. In the context of Cactus, Hiltunen [2] developed a methodology which is based on identifying relations between modules that dictate which combinations are correct; a configuration tool based on these relations allows only correct configurations to be created.

7 Related Work

To the best of our knowledge there is no other work that models the behaviour of Cactus or Appia operations. An understanding of the behaviour is critical for actually programming with these frameworks. In the Ensemble project, formalisation using the Nuprl theorem prover provided insight into the structure of the layered protocols and their optimization [1], however the framework itself has not been described formally. There has been work on formalisation of modules composition, e.g. [7], however it further abstracts away from programming frameworks.

The approach of Serjantov *et al.* [5] is similar to ours in that they aim to model the behaviour of partial systems, making explicit the interactions that the infrastructure offers to applications. They constructed an experimentally-validated specification of the standard UDP/ICMP sockets interface, including loss and failure, and integrated the above with semantics for an executable fragment of a programming language (OCaml) with OS library primitives. In our case, the “infrastructure” and “application” correspond in turn to the protocol framework and communicating protocols, with correspondingly more complex dependencies and mutual interactions. However, unlike them we do not need to deal with the distributed phenomena and complex failure semantics.

The goals put forth in [6] in the area of the location-independent communication for mobile agents are also related to the approach described here in the sense that the choice or design of protocols must be somewhat application-specific. However, unlike the Nomadic Pict programming language [9] which has been implemented and used to design many different communication infrastructures, provided as encodings of the high-level language primitives, the frameworks described in this paper use standard language facilities and support multi-level protocol composition.

8 Conclusion

8.1 Contribution

We have given a mathematically precise and experimentally validated model of protocol modules composition and interaction in Appia and Cactus (which subsumes the x-kernel model). It has been illustrated with a simple example application that uses two (idealised) group communication algorithms. The model consists of a set of inference rules defining operations and state transitions. The contribution of the formalisation is twofold. It provides a clear and concise description of a fragment of the programming interface provided by each framework. Moreover, we think that this specification is at the right level of abstraction to help reasoning about the design differences — it describes the frameworks' behaviour (sufficiently accurately) but without going into too many implementation details. The specification is also precise enough to give some useful hints for the designers and implementors of such systems. However, the model is not complete — our primary goal was to understand the design features of the example frameworks, instead of developing concrete reasoning tools that could be applied for programs in Cactus or Appia. Nevertheless, it might be interesting to see how we could express and verify certain properties in this model, like for instance deadlock freedom. Due to lack of time, we also did not cover the whole programming interface and some operations are missing, e.g. for dealing with timeouts and dynamic microprotocol loading; also the description of threads, error situations, and event scheduling should be sufficiently covered. Developing and refining a small example application identified a bug in one of the frameworks, which has been fixed up in a newer release of the system.

8.2 Further Research

The work described in this paper is a step towards a better understanding of protocol modules composition and interaction. However, it provides only a starting point — much additional work is required on algorithms decomposition, semantics, and implementation. We hope to address some of this within our Crystall project, that aims at the design of group communication services with solid semantics foundations. In our future work, we would like to design a language with clean abstractions for module composition and interaction in the context of fault-tolerant computing. One way of making an application tolerant to partial failures, is to replicate its services on different machines using group communication algorithms. The goal is to decompose the algorithms into configurable modules in such a way that module dependencies are reduced, and the (internal) communication between modules is optimised. The language should adopt a model which allows an application to specify its requirements so that they can be adequately reflected by a protocol suite built from modules. The language should also support a type system that can be used to verify certain properties of the protocol suite. Eventually, it should be possible to integrate the language abstractions with standard frameworks that are used to build component based software, in order to increase the applicability of the method.

Acknowledgements. We would like to thank Rick Schlichting and anonymous reviewers for useful comments that helped us to improve the paper. We also thank Luis Rodrigues for explaining some details of the Appia implementation. The project is supported by EPFL grant "Semantics-Guided Design and Implementation of Group Communication Middleware".

References

1. Mark Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
2. Matti A. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, October 1998.
3. Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
4. Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *ICDCS'01*, 2001.
5. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *TACS'01 (Sendai)*, October 2001.
6. Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages*, LNCS 1686, pages 1–31, 1999.
7. Purnendu Sinha and Neeraj Suri. On simplifying modular specification and verification of distributed protocols. In *HASE'01*, October 2001.
8. Paweł T. Wojciechowski, Sergio Mena, and André Schiper. Semantics of protocol modules composition and interaction. Technical report, School of Computer and Communication Sciences, EPFL, February 2002.
9. Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April-June 2000.
10. Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *INFOCOM'01*, April 2001.