Protocol Composition Frameworks
A Header-Driven Model

Technical Report IC/2005/007

Daniel C. Bünzli, Sergio Mena, Uwe Nestmann
École Polytechnique Fédérale de Lausanne (EPFL)
CH-1015 Lausanne, Switzerland
daniel.buenzli, sergio.mena, uwe.nestmann @epfl.ch

Abstract

Protocol composition frameworks provide off-the-shelf composable protocols to simplify the development of custom protocol stacks. All recent protocol frameworks use a general-purpose event-driven model to manage the interactions between protocols. In complex compositions, where protocols offer their service to more than one other protocol, the one-to-many interaction scheme of the event-driven model introduces composition problems by mixing up the targets to which data (list of headers) should be delivered. To solve these problems, we propose to shift the driving force behind interactions from the events to the headers they carry. We show that the resulting domain-specific header-driven model solves the composition problems, provides statically typed header handling and enhances protocol readability.

1 Introduction and summary

A protocol stack is a middleware infrastructure that provides a service to simplify the development of distributed applications running on a failure-prone computing infrastructure (communication or node failures). Whereas a protocol stack simplifies the development of a distributed application, the purpose of a *protocol composition framework* [7, 14, 6, 13, 1] is to ease the development of custom protocol stacks.

The vision of being able to tailor stacks to the needs of specific applications by composing off-the-shelf protocols drives research in protocol frameworks. This vision will only be realized if protocols are developed modular and hierarchical wise and powerful component languages are provided to structure and compose them (§3, §7.1). In this paper we argue that these goals are hindered by the programming model of recent protocol frameworks (§4). These frameworks [13, 6, 1] all base the interaction mechanism between protocols on an *event(-driven) model*. In this model, computations are specified by event handlers. Handlers can be bound to events and are executed when the latter are triggered. *Many* handlers can be bound to a *single* event, which means that the interaction scheme is *one-to-many* (vs. *one-to-one*).

The structure of protocol compositions is changing from *stacks* to *graphs* [10, 11] in which protocols offer their services to *more than one* protocol. The event model matches the reactive nature of distributed computing and the way protocols are described in the literature, but its one-to-many interaction scheme introduces composition problems in protocol graphs. Protocols may receive events that are not targeted at them. This compromises the definition of powerful component languages on top of an event model because *ad hoc* mechanisms need to be introduced to "route" events to the right protocols. Moreover, the event model doesn't properly handle *peer interactions*, where a protocol interacts with its peer running on another node by using the service of another protocol. The way events handle this ubiquitous pattern is (1) complex, many unnecessary bindings need to be done by the *composer* (2) obscure, the indirections introduced by events hide the logical structure of peer interactions in the code (3) unsafe, misbindings may lead to runtime type errors or erratic behaviour.

Instead, we propose a novel and simple alternative (§5, §6) that shifts the driving force behind interactions from events to the headers they carry. The resulting *header-driven model* (1) solves the composition problems of the event model (2) simplifies inter-protocol dependencies (3) concisely handles peer interactions and explicitly reveals their logical structure (no obfuscating indirections) (4) provides better static typing which avoids the runtime type errors and erratic behaviour that can occur in the event model. We show how the two models compare in the context of a component language in §7.

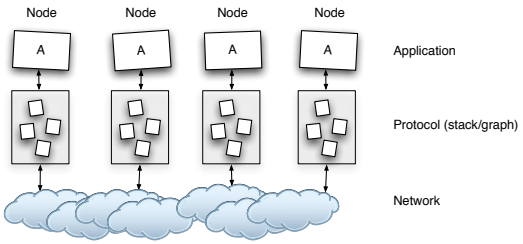


Figure 1. Protocol stacks

The contributions of this paper are (1) the demonstration that the event model[1] is *not* the right programming model for protocol composition frameworks because of the mix-up of events in protocol graphs and the unsatisfactory handling of peer interactions (2) the proposal of an alternative, header-driven model solving these problems and with better compositional properties.

Future work. Our model was implemented as a syntactic extension to the OCAML programming language [8] under the name NUNTIUS [4]. Early experiments seem to show that our programming model associated to OCAML's module system fulfills its compositional promises. Work has been done to encode these primitives as a JAVA library and to use classes, inner classes and interfaces as a rough component language. We are planning to develop a new protocol composition framework with the header-driven model at its core.

Scope. This paper is about modular programming and composition problems in protocol frameworks. More precisely, we are looking for a *composable* programming model to program protocols *inside* protocol frameworks. We focus on the case where every protocol has a peer on every node (Fig. 1). We do not address the issues related to interfacing with the application or the network. Neither do we treat the problem of introducing concurrency in protocol frameworks which is an issue we consider orthogonal to composition.

Acknowledgments. We thank Christophe Gensoul for implementing the first NUNTIUS prototype and Rachele Fuzzati, Olivier Rütti, André Schiper, Paweł Wojciechowski for their feedback.

2 Terminology

Base language and interaction scheme. A programming model defines a *base language* offering constructs to specify computations and to define data structures. The mechanism used to invoke computations defines an *interaction scheme*, it allows two distinct parts of a program to interact. For a functional base language this mechanism is function application (object-oriented: method invocation).

A *first-class value* is an entity than can be manipulated by the constructs of the base language. For example, in a functional programming language, functions are first-class values because they can be stored within data structures and given to or returned by functions.

Components. A *component* is a higher level construct that structures definitions of the base language. A *component abstraction* is a parametric component that relies on another *abstract* component without committing to a specific implementation for it. A component is *hierarchical* if it contains other components.

Components can be *composed* together to offer a new service implemented in the base language. A composition in which one can distinguish layers of abstraction in terms of services provided by groups of components is said to be *layered*. A *stack-based* composition is a layered composition in which each component provides its service to at most a single other component. A *graph-based* composition has no structural constraint, but it can still be layered.

The component language is just a syntactic way of structuring programs of the base language. There is no runtime entity representing a component. Expressions of the component language can be translated into the base language before running a composition. But for convenience, we will nevertheless discuss the runtime properties of a program in terms of its decomposition into components.

Nodes and protocols. A *node* is an unreliable unit executing computations (Fig. 1). The (usually) unreliable and asynchronous primitive communication mechanism between two nodes is called *inter-node communication*.

A *protocol* provides a well defined service by exchanging data with its *peers* which are replicas of the protocol running on other nodes. In the setting of protocol composition frameworks, a *protocol* corresponds to a component (we use these terms interchangeably) and since components can be hierarchical, a protocol can also denote a protocol composition.

Protocol interactions. Let us respectively refer to a protocol requesting a service of another as the *client* and the *server* protocol. Protocol interactions can be classified into *local* and *remote* interactions. In the former, both the client and the server protocols run on the same node whereas in the latter they are on different nodes. In this last case we can distinguish between protocols that (1) communicate directly with the primitive inter-node communication mechanism from those that (2) use the service of a local server protocol.

A particular and recurrent instance of (2) is *peer interaction*. Conceptually, a protocol A interacts with its peer as

[1] Which should not be equated with a *reactive* programming model, our proposal is also reactive.

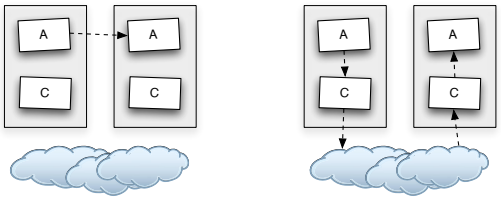


Figure 2. Peer interaction, execution paths

if it was performing a remote procedure call (Fig. 2, left). Concretely, A issues a request to a local server protocol C, which results in an interaction between their peers (Fig. 2, right). An example is the send/deliver scheme of a communication protocol.

3 Requirements

Composition. Previous research [10, 11] shows that the modular decomposition of group communication middleware results in complex interactions between various components from different abstraction levels. These compositions are out of the scope of simple stack-based composition schemes. In particular, services of protocols are sometimes used by *many* other components.

In order to tackle this complexity, a powerful component language is needed. To express dependencies between protocols and complex layers of abstraction, this language should allow the definition of parametric and hierarchical components and provide information hiding mechanisms. It should support the definition of both hierarchical and flat, cooperative, compositions. Composing protocols should be an easy task. In particular we would like to have coarse-grained composition mediated by interfaces instead of fine-grained sequences of input-output binding instructions. Coarse-grained composition is also supported by the requirement of information hiding since a single component can hide an already complex composition.

Hence we are looking for a *programming model* that supports, on top of it, the definition of a component language satisfying the requirements given above. In this paper, we use an ML module system [9] to validate our programming model since it satisfies our requirements[2].

Peer interaction. Most protocols work by interacting with their peers. These interactions almost always occur via peer interactions to benefit from the properties provided by other protocols. As such, expressing this pattern in our base language should be clear, explicit and concise.

[2]This module system was used in our setting [5] but not as a component language to create custom compositions.

$d ::= \texttt{event}\ n : t \mid \texttt{handler}\ h(x : t) \rightarrow e \mid \texttt{bind}\ h\ n$	*definitions*
$e ::= \texttt{trigger}\ e(e) \mid \ldots$	*expressions*

Table 1. An event language

4 Misfit of the event model

Event-driven primitives. A base language for an event model is given in Tab. 1. A program is a sequence of top-level definitions d. An event definition **`event`** defines an event name n which carries arguments of type t. The expression e of a **`handler`** definition is evaluated upon occurrences of events of type t bound to h; before evaluation, the argument x is substituted in e by the event's argument. A handler h is bound to an event n with **`bind`**, the type of the arguments should match. Events are triggered with **`trigger`**, the first expression should evaluate to an event name and the second to a value whose type corresponds to the event's argument. In the event model, components aggregate and structure events, event handlers and event bindings.

The discriminating feature of the event model is that *none* or *many* handlers can be bound to a *single* event and vice versa. Many semantical variations can be devised on these constructs. Static versus dynamic binding, serial versus concurrent execution of handlers, etc, but these differences are not relevant to our discussion.

In practice, protocol frameworks [6, 13] may use additional interaction schemes and structuring entities (e.g. channels [13]). The event language presented here is a rough extraction of their common essence. Besides, note that the event model used in these implementations is not as statically typed as presented above, event arguments are usually dynamically typed.

On one hand, in stack-based compositions of protocols one does not use the one-to-many interactions provided by the event model. On the other hand, this interaction scheme is not adapted to graph-based compositions where a protocol offers its service to more than one client.

The event routing problem. The problem lies in the way a server protocol responds to a request issued by a client protocol. The server's response is returned by triggering an event to which the client can bind a handler. Now if two protocols A and B use the same service provided by C (Fig. 3), they will both bind an event handler to this event. If A issues a request on C, both A and B will get the result. The problem is that this behaviour is in *most cases* un-

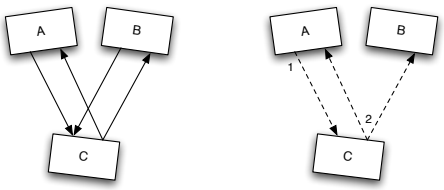


Figure 3. The event routing problem : bindings and execution paths

wanted. It seems quite natural that requests issued by two unrelated clients should not interfere. Thus in the example above a mechanism needs to be introduced so that B can discard the (wrong) data it gets. This *event routing* problem is depicted in Fig. 3; solid arrows denote events bound to handlers (left), dashed arrows denote the flow of events during a particular execution (right).

Of course the event routing problem does not show up in stack-based compositions since each protocol has at most one client. But in complex graph-based compositions [11], this problem occurs frequently. In a component language with component abstractions, the problem is even more acute: a protocol C can easily be given as a parameter to more than one protocol abstraction and the usage of C'service by those abstractions should also not interfere.

***Ad hoc* solutions.** There are several solutions to this problem in the event model. However, all these solutions are *ad hoc* and neither satisfactory nor elegant. We review *some* of them.

- *Destination check on handler invocation*. By packing additional data in the events' arguments, a handler can check whether it is the target of the event. However this burdens the programmer and introduces a performance overhead.

- *Component duplication*. Duplicate the server component so that each client has its own copy of it. This does not scale, leads to bloated compositions, and may introduce synchronization problems between duplicates.

- *Connector components*. Introduce components [10, 2] to route events to the right clients. This burdens the composer who needs to care to plug them at the right place. Furthermore, if we want to use the same protocol as a parameter for two different component abstractions, connectors need to be plugged in automatically. This may not be an easy task, leads to bloated compositions, complicates debugging and introduces performance overheads.

A solution not mentioned above is to use a call-back mechanism. This can be done by allowing event names to be first-class values. The client can give one of its event name as an argument to the service request and the server can use this name to trigger its result. However this amounts to a have *one-to-one* interaction between the client and the server which raises the question of the *purpose* of using this *one-to-many* interaction scheme in our setting.

Wrong interaction scheme. The event model allows more complex interaction patterns than, for example, function application. But it does not match what we need most of the time. It has been argued [10] that most bindings are *one-to-one*, a single handler is bound to a single event. Moreover, the indirections introduced by the binding mechanism significantly complicate and obfuscate the implementation and composition of protocols (see §5, §7). Finally, if really needed, a one-to-many pattern is easy to implement on top of a functional or object-oriented language.

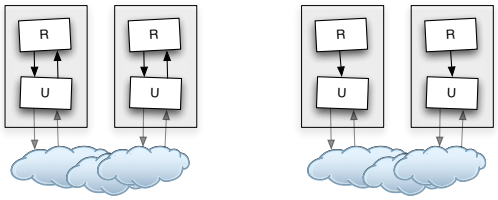
The event model can be seen as an *observer* pattern [3]. The intent of this pattern is to "define a one-to-many dependency between objects such that when one object changes state, all its dependents are notified and updated automatically". The use of this pattern in our setting is clearly not appropriate since, as we said above, state changes of a server protocol should *not*, in most cases, affect all its dependents (clients).

The observer pattern is also used to develop loosely coupled components. But the modular decomposition of a complex protocol results in *tightly* coupled components, in the sense that the role and properties of the sub-protocols are clearly defined. This tight connection is directed, higher level protocols should rely on precise lower level (possibly abstract) services, but the converse should not hold.

5 The header-driven model

The compositional shortcomings of the event model lead us to seek a new programming model for protocol frameworks. By contemplating how badly the event model manages peer interactions, we get to our new proposal.

Peer interaction in the event model. Protocols typically encapsulate communication data for their peers in messages and headers. A *message* is a list of headers and a *header* is a typed container for data. In a peer interaction a sequence of protocols is traversed. Starting in the first protocol with the empty message, each protocol pushes a specific header onto the message with the data for its peer and triggers an event to pass the resulting message to the next protocol. The last protocol of the sequence sends the message to the peer node with inter-node communication. On the peer node, the reversed sequence of protocols is traversed (provided bindings are correctly specified). Each protocol pops from



Event model Header-driven model

Figure 4. Composition bindings

the message the header transmitted by its peer and triggers an event to pass the resulting message to the next protocol.

A concrete example is given by the composition depicted on the left in figure 4. Black arrows denote the explicit bindings — dependencies — specified to compose the two protocols. Protocol R offers reliable node-to-node communication and U unreliable node-to-node communication. Conceptually R wants to give data to its peer by "calling" the handler `recv` on the other node. Protocol R packs data onto a message and gives it to the lower level component U by triggering an event. U sends the message with inter-node communication. Upon reception on the peer node, U's peer triggers an event, *hopefully* bound at composition time to R's `recv`, to deliver the message to R. The way the event model manages peer interactions has the following drawbacks.

- *Compositionally suboptimal.* The designer of R knows that `recv` should handle the data given to U. But, because of the indirections introduced by the binding mechanism, this cannot be explicitly coded in the protocol. Instead, the designer writes the handler `recv` and *hopes* that the right bindings will be done during composition. This is compositionally suboptimal because a constraint known at design time has to be explicitly enforced later, at composition time.

  Furthermore, handler `recv` is completely internal to the protocol R (and its peers). Yet it needs to show up in the *interface* of the protocol so that it can be bound by the composer to an event of U. This goes against abstraction since it prevents information hiding.

  A component language with component abstractions can slightly alleviate these problems (see §7). However there will *always* be *two* dependencies between R and U, one for the events flowing down and one for the events flowing up (Fig. 4, left).

- *Event routing problem.* If R or U is used by more than one client we get routing problems.

- *Failing or mixed up header deconstruction.* Messages are heterogeneous lists of data. They cannot be given a more informative type than "`msg`". A composer may incorrectly bind `recv` to an event unrelated to U but whose type matches `msg`. In that case, two bad things may happen. Either a *runtime type error* occurs because `recv` tries to pop a header from a message whose structure does not match its expectations. Or `recv` pops a header of the right type but that is not intended for R possibly resulting in *erratic behaviour*.

From events to headers. Protocols often use a single handler to manage the reception of peer interactions. Nevertheless, some protocols need to send different kind of data via this single handler. In order to do so, they introduce a *tag name* in the header to indicate the kind of information they transmit. Since a header usually remains internal to a protocol and its peers, it is not restrictive to impose that each header shall be *named*, and that each name shall be *declared by at most one protocol* in a composition. A composition satisfying these constraints has the following interesting property. If we look at the names of a message's sequence of headers, we can approximately see the sequence of protocols — the route — that will handle the message when it is processed by the peer composition. This means that there is no need, for the composer, to explicitly bind the upward flow of events. In other words *the message's sequence of headers drives its processing in the protocol graph*.

The event model prevents us from exploiting this property. Thus, instead of having events at the core of our interaction scheme, we should have *headers*. This is the essence of our proposal.

The essential ingredients of a *header-driven model* are headers and messages. A message is a list of headers. Headers are *named* containers carrying statically typed data. To construct a header, its name must be defined. A header handler defines a header name and associates a computation to the deconstruction of all messages starting with that name. Message dispatch is the interaction scheme, it deconstructs messages. When a message is dispatched, the unique header handler corresponding to the head of the message is invoked with the head's data and the tail of the message as arguments. Compared to the event model we can say that (1) header handlers replace event handlers (2) message dispatch replaces event triggering and (3) the binding mechanism is dropped.

In a header-driven model, the peer interaction described above occurs as follows. Protocol R pushes data onto a message using a header `recv`. Unlike in the event model, this *specifies* which header handler should be invoked on that data at the peer node. R gives this message to U using, for example, function application. U sends the message with inter-node communication. Upon reception on the peer node, U dispatches the message which becomes automati-

cally deconstructed at the right place by invoking the unique header handler `recv`.

Mirroring the defects of the event model, our proposal has the following benefits.

- *Better compositional properties.* Messages "know" where they need to be deconstructed. Therefore no bindings for the upward control flow need to be specified. This removes one dependency between the two components (Fig. 4, right). Besides, handler `recv` becomes truly internal to the protocol R and its peer, it does not appear in the interface.
- *No event routing problem.* If another protocol S uses U, it packs data into one of its own headers h. When U's peer dispatches that message, it is automatically routed to h's handler. The routing problem is trivially solved.
- *Correct header deconstruction.* A header is always constructed with the right type in the scope of a header handler for it. For any header occurring in any message, there is exactly one corresponding handler. This implies that at runtime, neither the deconstruction of a message can fail, nor can a handler get unrelated data. Both the runtime type errors and erratic behaviours found in the event model cannot occur.

6 Header-driven primitives

We show how to extend a general purpose host language with header-driven primitives. Note that these constructs are simple enough to be encoded into a tiny library.

Host language. For our presentation, we choose a small ML-like [12] statically typed functional programming language. In what follows, we write `x:t` for the type annotation of variable `x`. But we omit them most of the time assuming the existence of a type reconstruction algorithm.

The syntax of the host language is given in Tab. 2. A program is a sequence of top level definitions d. A *definition* d is either a type definition or a value definition. Meta-variables t and x, respectively range over type names and value names. We leave the language τ of type expressions unspecified, but we write `t→t'` for the type of a function from type `t` to `t'`, `t * t'` for the type of pairs whose first and second component are, respectively, in type `t` and `t'` and `unit` for the "void" type found in other programming languages. An *expression* e denotes a value. It can be, respectively, a function abstraction, a function application, a local definition of a value (local variable), a sequence of expressions, etc.

We extend this simple host language with header-driven primitives. Tab. 3 summarizes how each syntactical category is extended. Detailed explanations follow.

$d ::= \textbf{type}\ t = \tau \mid \textbf{let}\ x = e$	*definitions*
$e ::= \textbf{fun}\ x \to e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e\,;\,e \mid \ldots$	*expressions*

Table 2. Syntax of the host language

$d :+= \textbf{handler}\ h(x)\ \texttt{::}\ x \to e$	*Header handler*
$e :+= h \mid \texttt{[]} \mid e(e) \mid e\ \texttt{::}\ e \mid e\ \texttt{@}\ e \mid$	*Message construction*
$\textbf{dispatch}\ e \mid \textbf{rdispatch}\ e\ e$	*Message dispatch*

Table 3. Extension of the host language

Message construction. A *header* `h(x)` is a named entity holding statically typed data. `h` is the *name* of the header and `x` its *argument* or *data*. Header construction is strict, the argument is evaluated before the header is created. A header `h` can only be constructed in the lexical scope of a header handler for `h`.

A *message* is a list of headers. The head of the list is said to be the *destination* of the message while its tail is the *continuation*. Messages are built by pushing headers on the empty message `[]` using the right associative `::` operator. They can be concatenated using the binary operator `@`. These operators are purely functional (i.e. non destructive, referentially transparent).

In the example below two messages are concatenated to build a third one whose destination is `first`.

```
let m = first() :: second(4+2, 5) :: [] in
let m' = third("dipdip") :: [] in m @ m'
```

The second line of Tab. 3 shows how expressions of the host language are extended to allow message construction. We let meta-variable h range over header names. An expression e can also be, respectively, a header name h, the empty message, a header construction, the addition of a header to a message, a concatenation of two messages.

Messages, headers and header names are first-class values. However messages and headers cannot be deconstructed in expressions, it is only possible to access their contents upon header handler invocation.

Header handlers. A *header handler* defines a header name h and a computation to invoke when a message with destination h is dispatched. Header handlers are introduced by the keyword **`handler`**. In the declaration **`handler`** `h(x) :: m →e`, variables `x` and `m` bind in the *body* `e` of the handler. When a message with destination `h` is dispatched, variables `x` and `m` are respectively substituted by the destination's argument and the message's continuation in e

and the resulting expression is evaluated. A header handler has the following characteristics,

- It is implicitly recursive, the header name it defines can be used in the body of the handler.
- A header handler not only deconstructs the destination of a message but also the message itself by separating the head from the tail[3].
- Like event handlers of the event model, the invocation of a header handler does not return any value, it is only used for its side effect.
- Unlike functions of the host language, header handlers are not first-class values, they can only be defined at top level. This prevents the runtime creation of header names. The converse would break the symmetry between nodes and we would lose the property that a remote message dispatch never fails.

Message dispatch. *Message dispatch* is the interaction scheme that invokes header handlers' computations. Once a message `m` has been constructed it can be *dispatched* by executing the expression **`dispatch`** `m`. This primitive invokes[4] the unique handler corresponding to the message's destination. Dispatching the empty message `[]` does nothing, it evaluates to the unit value `()` (alternatively it could be an error).

The following program invokes the body of `count`'s handler exactly one hundred times. `repeat`'s handler repeats n times the behaviour of a list of arbitrary headers as long as all their handlers always take care, as does `count` below, to redispatch the continuation. After completion this program will hold the value 100 in the mutable variable[5] `i`.

```
handler repeat(n) :: m →
  if n > 0 then
    dispatch (m @ (repeat(n-1) :: m))

let i = ref 0
handler count() :: m →
  i := !i + 1; dispatch m

let main = fun () →
  dispatch (repeat(100) :: count() :: [])
let () = main ()
```

In the following example, `eat` would immediately break the loop because it doesn't redispatch the continuation.

```
handler eat() :: m → ()

let main = fun () →
  dispatch (repeat(100) :: eat() :: count() :: [])
let () = main ()
```

[3]Messages cannot be deconstructed in expressions as lists can in ML.

[4]Synchronously, but asynchronously is another option.

[5]For mutable variables definition, assignment, and dereference, we use e :+= **ref** $e \mid e$ **:=** $e \mid$!e

Header handler definition	≈ Function definition
Header creation	≈ Postponed function call
Message	≈ List of postponed function calls
Message dispatch	≈ Execute the head's postponed call

Table 4. A functional perspective of header-driven primitives

Handlers do not return values but they can continue the "potential of computation" held in the message's tail by redispatching it in the body of the handler. This is reminiscent of *continuation-passing style*, which is why we refer to the tail of a message as being its continuation.

The scoping rules and the static typing discipline ensure that message dispatch will always *succeed* in the following sense: there will *always* be a header handler with *matching type* for the destination of a message.

Remote message dispatch. Along with these primitives, a natural and essential inter-node communication mechanism is given by *remote dispatch*. A message `m` can be remotely dispatched on a node `n` by executing the (non-blocking) expression **`rdispatch`** `n m` whose effect is to execute **`dispatch`** `m` on the node denoted by `n` (a value of type `node`). Depending on the implementation, remote message dispatch may be reliable or unreliable. However if the message dispatch occurs on the remote node it will also succeed as defined above because of the scoping rules, the static typing discipline and the fact that every protocol has a peer on every node.

A functional perspective. The semantics of header-driven primitives corresponds to the semantics of functions as follows. A handler **`handler`** `h(x)::m →e` can be seen as the definition of a function **`let`** `h = fun x m →e` of two arguments where `x` is the header's argument and `m` the continuation. Dispatching a message **`dispatch`** `(h(x)::m)` becomes the function call `(h x m)` and remote dispatch is a form of remote function invocation. Following these intuitions, Tab. 4 establishes an informal correspondence between header-driven primitives and a functional programming language.

Header-driven primitives can essentially be seen as syntactic sugar for the continuation-based manipulation of lists of postponed function calls.

7 Header-driven *versus* event-driven

To compare both models and show that our proposal satisfies our requirements we implement the same simple composition in each model. First, we informally introduce a component language.

7.1 A component language

This component language is an ML-style module system [9]. It allows the definition of parametric and hierarchical components. Its ingredients are protocols (implementations), protocol abstractions (parametric implementations) and protocol types (interfaces).

Protocols. A *protocol* (module) aggregates top-level definitions and sub-protocols into a structure delimited by curly braces `{}`. This structure can be bound to a name with the **`protocol`** construct. Here is an example of a structure bound to the name `a`.

```
protocol a = {
  let succ = fun x → x + 1
  protocol b = { handler eat() :: m → () } }
```

Top-level definitions of a named structure can be referred to with a dot notation. In the example above, `succ` and `eat` are referred outside the protocol `a` with the qualified names `a.succ` and `a.b.eat`. Protocols introduce new scopes for top-level definitions. Two definitions with the same name defined in two different protocols are deemed different.

Protocol types. A *protocol type* (module type) is a collection of top-level specifications and sub-protocol specifications delimited by curly braces `{}`. We write **`val`** `x:t` for a value specification of type `t`, **`handler`** `h:t` for the specification of a header handler with argument of type `t` in the header-driven model[6] and **`event`** `e:t`, **`handler`** `h:t` for events and handlers of type `t` in the event model. A protocol type can be bound to a name with the **`protocol type`** construct. The protocol `a` given above can be described by the following type `A`.

```
protocol type A = {
  val succ : int → int
  protocol b : { handler eat : unit } }
```

The type of a protocol represents its interface. Protocol types are used to hide declarations, to specify abstract protocols and to mediate the composition of protocol abstractions. For each protocol definition, the type system infers the most informative type: all top-level definitions and sub-protocols get a specification. For example, for the protocol `a` given above, the type system would infer the signature `A`.

The inferred type of a protocol can be explicitly constrained by another (less informative) type to hide parts of the structure. Protocol type constraints are written $p : P$, where p is a protocol and P a protocol type. For example, the sub-protocol `b` of `a` could be hidden by writing,

```
protocol c = a : { val succ : int → int }
```

Trying to access `eat`'s handler of the sub-protocol `b` by referring to it as `c.b.eat` would result in a compilation time error. Note that `c` and `a` represent the same protocol but they offer two different views of its interface.

Protocol abstraction. A *protocol abstraction* (functor) declares a protocol name `c`, specifies it abstractly with a protocol type `C`, and binds this name in a structure by prefixing the latter with `(c:C)` ⇒. The abstract protocol `c` can then be used in the structure according to its protocol type. A protocol abstraction can also be named with the **`protocol`** construct. Below, the protocol abstraction `forwarder` is parametric on an abstract sender `s` specified by the type `Sender`.

```
protocol type Sender = { val send : msg → unit}
protocol forwarder = (s : Sender) ⇒
 { handler forward() :: m → s.send m }
```

Given a concrete protocol `p'` implementing the protocol needed by a protocol abstraction `p`, a *protocol instantiation*, written `p(p')`, turns `p` into a concrete, usable protocol instance. For example, to be able to use the `forwarder` abstraction given above, we have to give it a concrete protocol that satisfies the type `Sender`. Let `sender` be such a protocol, a forwarder `f` can be instantiated by writing **`protocol`** `f = forwarder(sender)`. Note that if we instantiate a second forwarder `f'` (maybe with the same `sender`), headers denoted by `f.forward` and `f'.forward` will always be different and therefore respectively deconstructed in `f` and `f'`.

7.2 Comparing the models

We compare the header-driven and the event models by implementing the composition of Fig. 4 in both models (see Tab. 5). Protocol `u` offers unreliable communication. Provided with an unreliable communication protocol `unrel`, protocol abstraction `r` offers reliable communication.

Implementation. We compare the two implementations.

Protocol type `U` specifies the abstract protocol `unrel` used by `r`. The event model needs to declare an additional interaction point, the event `deliver`. This is not necessary in our model because of the anonymous continuation like nature of message dispatch. The resulting protocol type is simpler and less constraining. In the header-driven model, client requests are made through the *function* `send`.

Protocol `u` implements the type `U`. In the spirit of the **`rdispatch`** primitive, we assume the existence of a **`rtrigger`** primitive that triggers an event on another node. The value `this` of type `node` represents the local node. Both models handle in `recv` the effect of the inter-node communication

[6] In practice header handlers remain internal to protocols.

issued in send. The event model triggers deliver and this can, depending on the final composition, invoke many event handlers. The header-driven model dispatches the message's tail and this will invoke a single handler *whatever the composition will be*. In that case, we also see that no dependency is introduced between u and the protocol that further processes the message. This is not the case in the event model, since handlers that continue to process the message have to match deliver's type.

Protocol type R specifies the protocol abstraction r. The same remarks we made for type U apply here. To break monotony send also specifies a boolean flag carried to the peer.

Protocol abstraction r implements the type R. In the event model, explicitly mentioning the dependency on protocol unrel allows to keep recv internal. If unrel is not mentioned recv needs to be declared in the interface. This is not the case in the header-driven model where recv is kept internal in both cases. In the header-driven model, in send, the intended "remote call" to recv is explicitly specified when the message is constructed and given to unrel.send. This contrasts with the event model where the whole event binding map needs to be deciphered to realize that when we trigger s, the "remote call" will eventually lead to the invocation of recv.

Composition. The composition given in Fig. 4 is implemented by the following definition,

```
protocol stack = r(u)
```

This stack-based composition will behave accordingly in both programming models. However, with the following graph-based composition, the behaviour will considerably differ.

```
protocol graph = { protocol rptp = r(u)
                   protocol rptp' = r(u) }
```

In the event model we run into an instance of the event routing problem. A send request made on rptp will invoke on the peer node both rptp.recv and rptp'.recv which is certainly not the intended behaviour. This is not the case in the header-driven model, since header rptp.recv and rptp'.recv are different and will therefore be deconstructed at the intended place when u dispatches the message to deliver.

Header deconstruction. In the event model, the implementation of handler recv in protocol r is written `handler recv((from, flag) :: m) →...` This implicitly means that we pop the first header from the message. However we do not have any kind of static guarantee that this header will have the type node * boolean as we assume here. Misbindings in the scope of a larger composition could lead the protocol standing for unrel to deliver a message whose first header is not of the right type or that is unrelated to r. This respectively causes a dynamic type error or erratic behaviour. In the header-driven model, this cannot happen, header deconstruction can only occur at the intended place with the intended type.

Readability. The implementation of this simple composition is arguably more readable in the header-driven model. The interactions are specified explicitly whereas in the event model the indirections introduced by the binding mechanism hide their logical structure. For example, in the header-driven model, the implementation of r.send clearly shows that we are "calling" recv on the peer node.

Protocols are also more readable when they need to use more than one header. In the event model, everything goes through the same recv handler and header names are distinguished by tags. This leads to the following code structure.

```
handler recv((from, tag, data) :: m) →
  if (tag = ping) then ...
  else if (tag = pong) then ...
```

In the header-driven model, declaring a new interaction point is easy — no binding overhead — therefore one would naturally declare different headers in separate handlers wich allows better header reuse.

```
handler ping(from, data) :: m → ...
handler pong(from, data) :: m → ...
```

References

[1] F. Brasileiro, F. Greve, F. Tronel, M. Hurfin, and J.-P. Le Narzul. Eva, an event-based framework for developing specialized communication protocols. In *Proc. IEEE NCA'01*, 2001.

[2] R. Ekwall, S. Mena, S. Pleisch, and A. Schiper. Towards flexible finite-state-machine-based protocol composition. In *Proc. IEEE NCA'04*, 2004.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[4] C. Gensoul. Implementing Nuntius in the Objective Caml System. Master's thesis, EPFL, 2004.

[5] M. Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, 1998.

[6] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.

[7] 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, 1991.

[8] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml System: Documentation and User's Manual*. http://caml.inria.fr, 2004.

[9] D. B. MacQueen. Modules for Standard ML. In *Proc. of the ACM Conference on LISP and Functional Programming*, pages 198–207. ACM Press, 1984.

[10] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. cactus : Comparing protocol composition frameworks. In *Proc. of SRDS*, 2003.

```
protocol type U = {
  val send : node → msg → unit

}

protocol u = { ...



  handler recv(from) :: m →
    ...; dispatch m

  let send = fun dest m →
    rdispatch dest (recv(this) :: m)


}

protocol type R = (unrel:U) ⇒ {
  val send : node → bool → msg → unit

}

protocol r = (unrel:U) ⇒ { ...



  handler recv(from, flag) :: m →
    ...; dispatch m

  let send = fun dest flag m →
    unrel.send dest (recv(this, flag) :: m)



}
```

```
protocol type U = {
  handler send : node * msg
  event deliver : msg
}

protocol u = { ...
  event deliver : msg
  event s : node * msg

  handler recv(from, m) →
    ...; trigger deliver(m)

  handler send(dest, m) →
    rtrigger dest s(this, m)

  bind recv s
}

protocol type R = (unrel:U) ⇒ {
  handler send : node * bool * msg
  event deliver : msg
}

protocol r = (unrel:U) ⇒ { ...
  event deliver : msg
  event s : node * msg

  handler recv((from, flag) :: m) →
    ...; trigger deliver(m)

  handler send(dest, flag, m) →
    trigger s(dest, (this, flag) :: m)

  bind unrel.send s
  bind recv unrel.deliver
}
```

Table 5. Header-driven versus event-driven

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

[12] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[13] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS'01*, pages 707–710, 2001.

[14] J. Pereira and R. Oliveira. Object-oriented open implementation of reliable communication protocols. *OOPSLA'97 Ws. on Dependable Distributed Object Systems*, Oct. 1997.