Object Protocols as Functional Parsers

Gert Florijn

Utrecht University, Department of Computer Science
P.o. Box 80.089, 3508 TB Utrecht, the Netherlands
E-mail: *florijn@cs.ruu.nl*

Abstract. A service definition is an abstract specification of the behavior of a software component. It provides the interface between the users of an object and its (hidden) implementation. A protocol can be a part of a service definition. It captures the conditions under which interface operations can be invoked. This is of use for the designers of clients, but also for implementers of the service, at least if the protocol mechanism provides automatic (static or dynamic) acceptability checking of invocations or messages. Existing protocol formalisms are mostly based on finite state machines that describe legal orderings of messages. This is too limited, however, to model more complex services or to handle conditions that go beyond the ordering (such as access-control or time-dependencies) without referring to a service implementation.

In this paper we explore a grammar-based approach to protocol definition, i.e. we define protocols as non-deterministic grammars extended with provisions for parallellism. The resulting protocols not only define legal message sequences but can also impose context-sensitive constraints on properties of messages, like its sender or a time stamp. Protocotols are truly abstract, i.e. independent of the implementation of a service. Since we want to investigate the kinds of constructs needed in specifying protocols we do not introduce a new notation, but instead express protocols as parsers written in a lazy functional language using a technique called combinator parsing. Besides giving a complete picture of the semantics of the constructs this also allows us to show how dynamic protocol checking can be provided. The approach is illustrated through several examples and its potential and limitations are discussed.

Keywords: service definitions, interface definitions, protocols, functional programming.

1 Introduction and background

Encapsulation is one of the cornerstones of object-oriented development. The separation between the interface of externally visible operations of an object and its hidden implementation provides a firewall that limits the effects of changes to a system. As long as the interface of an object stays the same, its internal operation can be adapted without affecting the rest of the system.

The specification of an object-interface – a *service definition* – thus is an important element of an object-oriented design. It defines the operations that are part of the interface and specifies their behaviour. This involves two viewpoints: 1) the conditions under which the operations can be used and 2) the conditions that the implementation of the operations should meet.

A service definition therefore is a contract that enables clients and implementors to cooperate [Meyer88]. It should be sufficiently detailed to allow (designers of) clients to know how a particular component providing the service should be used. Similarly, it should provide an implementor of the service with enough information to choose a suitable implementation. To give the implementer enough freedom it is important that a service definition is abstract, and does not reflect any particular implementation.

In most programming languages, interfaces are defined in terms of signatures of operations. This is insufficient for both viewpoints. Service implementors need to know more about the required behaviour, while clients need to know more about when and how operations can be invoked. This can involve constraints on data that cannot be captured in the typing-system, but also legal orderings of operations, e.g. a file that must be opened before it can be read or written. In concurrent object-oriented languages these conditions are perhaps even more important since they define the synchronization constraints under which messages by senders are accepted by receivers.

Lacking suitable specification constructs, these conditions are mostly captured in comments or in a separate design document written in another language. In most cases these design documents are not part of the implementation of a system, even if a formal specification language is used. The conditions specified in the design have to be coded in the service implementation to make sure that they are checked in an actual system. This situation has obvious drawbacks. We end up with two versions of a service definition that have to be maintained separately. Since there are no direct benefits for a service implementor in maintaining the specification, doing so can become tedious and lead to inconsistencies between code and specification, even in a development environment where formal specifications and implementations are dealt with in an integrated fashion [Perry89, Weide91]. The end-result is that the implementation is the only definition we can trust.

Some programming languages have introduced specific mechanisms to encode the conditions. In Eiffel [Meyer88] for example, we can use assertions: pre- and postconditions that are associated with the (interface) methods of a class. Pre-conditions encode the client perspective and post-conditions encode the behaviour side. Since assertions are represented as boolean expressions in the Eiffel language and thus are executable, there is a direct benefit for clients and implementors because there is no need to add or maintain extra code to check assertions. If a client does not meet a pre-condition, the assertion and thus the invocation will fail. Likewise, failure to meet a post-condition will create an exception that a client may or may not want to handle. However, there is a potential problem with this approach. Since the assertions are part of a class definition which combines both interface definition and implementation (at least if it is not a deferred class), it is all too easy to let assertions refer to implementation aspects. If developers do not use the mechanism properly it is easy to break the encapsulation barrier and thus make service definitions less abstract than they should be.

Experience with mechanisms to define synchronization constraints in concurrent object-oriented languages points out that a separation of these conditions from the implementation would help to prevent the so-called inheritance anomalies [Matsuoka93, Meseguer93]. These can occur when behaviour added to a sub-class can affect the synchronization constraints defined in the super-class.

1.1 Protocols

Ideally, therefore, a service definition is a distinct part of a software component, possibly written in language different from the implementation language. It should be defined independently from any implementation, but still be of use for an actual implementation to prevent the problems of double maintenance.

As far as the client perspective is concerned, *protocols* can meet this goal. A protocol defines the conditions under which a particular interface operation can be performed. Typically this is done by specifying legal sequences of invocations of these operations in terms of a (finite) state machine. The transitions of the state machine are (invocations of) interface operations, while the states represent abstract states of the particular service (not to be confused with the implementation state associated with objects). Consequently, a protocol is defined independently of any particular implementation.

Protocols can be of use in the implementation of a service, at least if the protocol mechanism is linked to the implementation. One way in which this can be done is by dynamic checking. During execution of a program, the state machine for the protocol can act as a recognizer that checks whether incoming messages (invocations) are acceptable or not. By bouncing messages that are refused in the protocol, the service implementor does not have to take the cases excluded by the protocol into account. Protocol mechanisms like this have been introduced in several (concurrent) object-oriented languages, like Procol [Bos89, Laffra92] and Sina [Aksit92].

Nierstrasz [Nierstrasz93] discusses *static* checking of protocol compatibility. He defines the notions of a regular process, a, possibly non-determinstic, finite state machine whose transitions are invocations of interface operations, and of a regular type, a specification of a regular process. Based on this, he defines a subtyping relationship on regular types which allows us to determine statically whether one protocol can replace another, e.g. in the case of inheritance. Similarly, the notion of request satisfiability is derived, which defines whether a client protocol (the behaviour that a client expects of a service) is compatable with the protocol offered by a service.

Protocols defined by finite state machines are not sufficient to handle all situations. Nierstrasz gives the example of a protocol for a stack, where a "pop" operation is only acceptable if the stack is not empty. To model this protocol as a deterministic state machine – i.e. a machine where there is at most one transition labelled with a particular operation from each state – we need an infinite amount of states, where each state corresponds to a stack with a particular number of elements. Since this makes it hard to reason about satisfiability, Nierstrasz proposes to model this by a non-deterministic finite state machine, where a hidden (implementation) decision determines whether a pop leads to the empty or the full state (see figure 1).

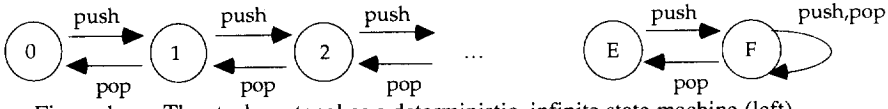

Figure 1: The stack protocol as a deterministic, infinite state machine (left) and as a non-deterministic, finite state machine (right).

In the protocol mechanism offered by Procol, some of this non-determinism can be captured by adding guards to protocol expressions. The guard is a boolean expression that determines which transitions are enabled. Itcan refer to instance variables in the object. In this case we could add a guard referring to an instance variable holding the number of elements currently on the stack, leading to (where + separates alternatives):

```
Obj  STACK(int len)          -- len is max length
 Declare  int n;             -- n is current position

 Protocol (n<len-1): ANY->push + (n>0): ANY->pop
 ...
Endobj STACK;
```

However, this implies a violation of the encapsulation principle – the protocol is not defined in abstract terms but refers to elements of the service implementation (the instance variable n). Clearly, a more abstract way to handle this is necessary, e.g. by letting the guards invoke an interface method of the object such as `isfull` or `isempty`.

A shortcoming of protocols defining only legal sequences is that there other issues involved in deciding whether an incoming message is admissable or not. One aspect is *access-control*: acceptance of an invocation can depend on the identity (or class or another property) of the sender. This is needed for example when update operations performed on the contents of a file object can only be performed by the object that has performed the open operation in an earlier stage. Procol allows access constraints on the class and even the identity of the sender. For example, the keyword ANY in the example above can be replaced by a class name. However, to model the file-protocol it again relies on the implementation. The implementation of the open operation has to record the sender in an instance variable, that can be used to guard subsequent operations.

The access constraints on the file update operations are a particular example of the more general notion of context dependency. Whether or not a particular message is acceptable can depend on properties of earlier messages. In the file-protocol there is a dependency on the sender of an earlier open message. There can also be dependencies on data: arguments passed in a particular invocation may affect legal arguments of future operations. Other aspects like time could play a role, e.g. when other messages should arrive within a certain time-frame after other messages or when there should be a certain delay between invocations.

1.2 About this paper

The aim of this paper is to explore a richer protocol mechanism that can express complex object protocols in an abstract way, i.e. independently of a particular implementation, while making (dynamic) checking possible. The approach we take is grammar-oriented: a protocol defines legal traces of messages, where a message is the invocation of a particular interface operation by a particular object. Messages have properties on which the protocol can place constraints. In this paper we consider the sender, the argument data and time.

Since we want to experiment with the constructs that are needed to model protocols, we do not introduce a new notation. Instead we encode them as non-

deterministic, functional parsers written in Gofer, a lazy functional language with strong typing. More specifically, we use a technique called combinator parsing [Hutton92], and express the parsers as monads [Wadler90, Wadler92]. While making the examples somewhat less "easy" to consume, the advantage of using Gofer is that it is easy to define new constructs (e.g. through higher order functions), while at the same time retaining executability. All the examples given are stylized versions of "working" protocols that can be run in Gofer[1].

As indicated before, our goal is that protocols can be specified independently from any service implementation. Therefore we do not introduce a particular (object-oriented) programming language. The basic idea is that once a message is accepted by the protocol, the corresponding operation will be performed on the object on which the protocol parser is active. As a consequence, we assume that normal type checking (checking whether actual and formal arguments and result type are compatible) is done by the underlying language.

The paper is organized as follows. In chapter 2 we introduce to the notation and technology used to express parsers, i.e. we briefly discuss parsing with combinators, monads and Gofer. Chapter 3 shows how parsers can be used for defining protocols by defining parsers of messages. Chapters 4 and 5 provide several examples of protocols, while at the same time extending the expressive power of the parsers with parallellism and shared state. Then, in chapter 6 we discuss incremental checking of protocols by defining an implementation of parsers that will react to individual incoming messages. Finally, in chapter 7, we evaluate the work and discuss some possible extensions and problems.

2 Parsing with Combinators

Parsing in functional languages is very simple. A parser is modeled as function that takes a sequence of input items, tries to match some items from it, and returns a result e.g. a (partial) parse tree. Basic parsers match invididual input items. They are glued together into larger parsers by higher-order functions or combinators [Hutton92] to handle constructs like sequencing or alternatives. Parsers can be non-deterministic: one input-sequence can be parsed in different ways. To model this, the "list of successes" approach is used. Parsers return the results of all possible parses on a given input.

In Gofer, we can define a parser as a function of the following parametrized type:

```
type Parser is a  =  is -> [( a, is )]
```

A parser is function (denoted by the arrow) that takes a state of type `is` (typically, a list of tokens) and produces a list of tuples. Each tuple represents a succesful parse, that is, a result value of type `a` together with the adapted state (typically the remainder of the input that was not parsed).

Parsers are constructed by combining elementary parsers. Two of the basic parsers are: `fail` which always fails, and `okay` which succeeds with a given value as its result:

---

[1]The programs can be obtained via World-Wide Web at URL:
http://www.cs.ruu.nl/~florijn/reseach/ariadne.html

```
fail     ::  Parser is a        okay        :: a -> Parser is
fail xs  =   []                 okay v xs   =  [ (v,xs) ]
```

Note that the first line in these definitions gives the type of the function. Another basic parser is `item`. It succeeds with the next item of the input, and fails when no input is left.

```
item ::  Parser [a] a
item [] = fail []
item (i:is) = okay i is
```

Note that we assume here that the input is modelled as a list of items of type `a`; `(a:b)` denotes a list whose head element is `a` and whose tail is the list denoted by `b`.

2.1 Combinators

Elementary parsers are combined using combinators. The first one discussed here is `into` which performs a parser on the input and – if it was succesful – performs a second parser on the remaining input. The second parser gets the result of the first parser as an argument. `into` returns the result of the second parser. The second combinator is `orelse`. It applies both parsers on the current input and returns the combined results.

```
infixr 6 `into`
infixr 4 `orelse`

into   :: Parser i a -> (a -> Parser i b) -> Parser i b
(p `into` q) xs = [(r,zs) | (v,ys) <-p xs, (r,zs) <- q v ys]

orelse :: Parser i a -> Parser i a -> Parser i a
(p1 `orelse` p2) xs  =  p1 xs ++ p2 xs
```

`into` and `orelse` are defined as infix operators, where `into` binds more strongly. As follows from the definition, `orelse` concatenates (the ++ operator) the results of attempting both alternatives, implying that a parser returns a list of sucesses. The definition of `into` uses list-comprehension: `[r|v<- e1, r<- e2 v, p r]` denotes the list of `r`'s obtained by iterating `v` over the values returned by `e1`, binding `r` to the result of applying `v` to `e2` and checking whether `p v` is true.

The fact that parsers are functions means that we can combine them easily and write higher order functions. Take for example the function `oneof` which succeeds when one parser of a list of parsers suceeds and which returns the result of that match. We must insert the `orelse` operator between all the parsers:

```
oneof   :: [Parser a b]  -> Parser a b
oneof l = foldr1 orelse l
```

For a real application a few other definitions are useful. The first is `sat`. It succeeds when the next input item satisfies a given predicate, and returns the input item if successful. Using `sat` it is easy to define `lit`, which matches a literal item in the input

```
sat  ::  (a -> Bool) -> Parser [a] a
sat p = [ (r,xs') | (r, xs') <- item xs, p r]

lit :: Eq i => [i] -> Parser [i] i
lit x = sat (==x)
```

Now we can define a parser that matches nested parenthesized expressions from a string of characters and returns the nesting-depth (an integer):

```
nest :: Parser [Char] Int
nest = (lit '(' `into` (\_ -> nest `into` (\d ->
                          lit ')' `into` (\_ -> okay (d + 1)))))
       `orelse` okay 0  -- empty alternative
```

Here we use the lambda abstraction (`\args -> expr`) to turn parsers in a form suited for use with `into`. The underscore character is used if we are not interested in naming the argument. Note that this definition requires (Parser i) to be a monad (see below). Finally, to actually run a parser on some input, we define:

```
parse :: Parser i a -> i -> [(a,i)]
parse p xs = [(x,s) | (x,s) <- p xs]
```

Applying "`parse nest "(())"`" will produce a list of two results: one with depth 2 that consumed all input, and one with depth 0 that consumed no input.

2.2 Parsers as monads

Given our definitions we can define parsers as monads [Wadler90, Wadler92]. Conceptually, monads allow us to make a distinction between a value and the computation that produces that value. Specifically, a monad `m a` denotes a computation of type `m` which, when evaluated, will produce a value of type `a`. To define a type as (various kinds of) monads we have to give definitions for certain functions. Using the monad constructor classes [Jones93] defined in Gofer, this results in the following definitions (the comments indicate the type of the function in terms of a monad `m`):

```
instance Monad   (Parser i) where
         result = okay  -- result :: a -> m a
         bind   = into  -- bind   :: m a -> (a -> m b) -> m b
```

The main advantage of defining parsers as monads is that we can now glue together parsers using "`bind`" and "`result`". If we change the implementation of parsers, the definitions of our parsers can remain unchanged, as long as the new implementation can again be mapped to a monad, that is, as long as we can define new implementations for `bind` and `result`. By adding a few more definitions, for a zero element and for the concatenation:

```
instance Monad0    (Parser i) where
         zero   = fail  -- zero :: m a

instance MonadPlus (Parser i) where
         (++)   = orelse-- (++) :: m a -> m a -> m a
```

we can even use monad-comprehensions to define our parsers. Monad comprehensions are similar to list-comprehensions (in fact, the list is a monad). To illustrate this, we give two definitions of `seq` which performs two parsers in sequence and returns the combined results in a tuple:

```
seq :: Monad m => m a -> m b -> m (a,b)
p1 `seq` p2 = p1 `bind` (\a -> p2 `bind` \b -> result (a,b))
p1 `seq` p2 = [ (a,b) | a <- p1, b <- p2]
```

The second form is internally transformed to something similar to the first form. Note that this version of `seq` works for any datatype that is a monad. We illustrate the use monad comprehensions further by defining a few other combinators. The first is `serie` which matches a list of parsers in sequence and returns the combined results as a list. The second one is `option`, which implements the optional construct. If the argument parser succeeds, its result is returned (in a list), otherwise the empty list is returned. In fact, these functions also operate on any monad of the given class. Finally, we define `many` and `many1` that succeed when the argument parsers matches zero or more (one or more for `many1`) times:

```
serie :: Monad m => [m a] -> m [a]
serie [] = result []
serie (p:ps)  = [ a:as | a <- p, as <- serie ps ]

option :: MonadPlus m => m a -> m [a]
option p   =  [ [x] | x <- p] ++ result []

many, many1 :: MonadPlus m => m a -> m [a]
many  p = [ (f:r) | f <- p, r <- many p] ++ [[]]
many1 p = [ (f:r) | f <- p, r <- many p]
```

3 Protocols as message parsers

A protocol is a specification of the conditions under which messages are acceptable to an object that implements a service. By modelling protocols as functional parsers that match incoming messages, we can specify the protocol independently of a particular implementation. In this section we illustrate the general idea by defining a suitable representation for messages, and by discussing a simple example.

3.1 The general idea

Conceptually, the protocol parsers are used as a per-object arbiter function operating on incoming messages. There will be a running instance of a protocol parser for each object that should meet the protocol. If an incoming message is accepted by the protocol, it will lead to a method invocation. If the message is rejected, we assume that the sender is informed of this, e.g. by an exception. Objects could run concurrently, e.g. with synchronization between sender and receiver on the acceptance of a message by the protocol (as in Procol). In this case we expect that multiple incoming messages are queued for handling by the protocol.

As mentioned before, we do not consider any concrete (object-oriented) programming language for which the protocol mechanism is used. As a consequence we do not consider the type-checking of invocations of interface methods. The main reason for this is that we want to focus on the constructs needed to define protocols. Adding type-checking would muddle the definitions and make the discussion too language-specific. So, although type checking could be introduced in the protocol (constraints on arguments for example are possible, see the next chapter) we assume that it is handled by the programming language implementation itself (e.g. in the

compiler). In particular, we assume that type-checking is performed before a message is passed to the protocol for deciding whether it is acceptable.

As far as the implementation of protocol parsers is concerned, we assume for the time being that the complete input, i.e. all the messages that are sent to an object, is available in a list. The implementation of incremental checking that is necessary for pratical applications is discussed in chapter 6.

3.2 Message representation

Protocols are parsers of messages. Any constraint defined in the protocol must thus be defined in term of information associated with incoming messages. So, if we want to impose constraints on the sender of a message, the sender should be part message structure that is offered to the protocol. In this paper we consider three of such properties, i.e. the data that makes up the arguments, the identity of the sender object and a timestamp denoting the moment the message was sent. Additional properties that could be of interest would be the location from which a message is sent (in a distributed system) or the class of the object that sent the message (as used in Procol).

In Gofer terminology this means that a message is a tuple consisting of an operation name or selector, arguments (data), the sender and a timestamp. As far as data values are concerned, we take a simple approach (see the discussion about type-checking above). We consider elementary values of type Text and Number and composite values of type Set and Record, a mapping from names to values. We can express this through the following types:

```
type Selector   = String
type Sender     = String
type Timestamp  = String
type Message    = (Selector, Value, Sender, Timestamp)

type Label = String
type Assoc = (Label, Value)
type Env   = [Assoc]

data Value = Text String
           | Number Int
           | Set [Value]
           | Rec [Assoc]
           | None        -- No value

assoc x ((y,v):as) | x == y = v
                   | otherwise = assoc x as
```

In addition to this, we define a number of functions that extract certain elements out of `message` tuples or `value` tuples:

```
timestamp (_,_,_,x) = x        tv (Text i) = i
sender (_,_,x,_)    = x        nv (Number i) = i
value (_,x,_,_)     = x        sv (Set i) = i
selector (x,_,_,_)  = x
```

For the implementation of the examples it is also necessary to make the `message` and `value` types instances of classes `Eq` and `Ord`, so that they can be compared and ordered. These definitions have been omitted here, but involve defining two comparison functions on each type.

3.3 A simple example: the file-protocol

If we assume that the input is a (complete) list of `message` tuples we now can actually define and run protocols. We can use `sat` to match specific messages, since it applies a predicate on the next input item. For convenience we define a parser `lmsg` which matches any message with a particular selector. Recall that the message is a 4-tuple as defined above.

```
lmsg :: Parser [Message] Message
lmsg l = sat (\(s,v,o,t) -> s == l)
```

Using combinators defined earlier we can write a simple protocol parser for a file object (henceforth the type definitions for the parsers have been omitted for brevity):

```
file = many session

session = [ (o,rw,c) | o <- lmsg "open", rw <- readwrite,
                       c <- lmsg "close" ]

readwrite = many ( oneof [lmsg "read", lmsg "write"] )
```

The parser `session` matches one valid walk-through of the protocol and returns the messages that were parsed in a 3-tuple. The combinator `many` is used to match zero or more or occurences of read or write operations and to get the "infinite" loop providing multiple sessions on the same file.

4 Examples of protocol parsers

In this section we illustrate the expressive power of parsers by discussing several examples. In the course of this discussion we also introduce some new combinators. We begin with variations of stack protocols that show how context-dependency and access-control can be provided. Then, we show how dynamic instantiation of parsers can be used to model a fairly complex voting protocol. We conclude this section with some discussion.

4.1 Simple stacks

In the introduction we discussed the protocol for stacks. Using the existing combinators, we can define a simple stack protocol in a straightforward fashion:

```
stack      = many push
push       = [ f | f <- lmsg "push", r <- full]
full       = [ r | f <- many push,   r <- lmsg "pop" ]
```

The basic parser is `stack` that represents the empty stack. The parser `full` is invoked whenever an item is pushed onto the stack. Each invocation of `full` thus corresponds to a stack with a particular number of items on it. `Full` can handle arbitrarily many extra "pushes" (which lead to recursive invocations) but it will complete by matching the "pop" of the item that was pushed onto the stack just before this particular invocation. Compared to the state-machine solutions in the introduction, this solution thus creates a (possibly infinite) list of states. Since the return values of parsers are not really relevant in this case, the `push` parser returns the "push" message while the `full` parser returns the "pop" message.

Obviously, real-life stacks are not infinite in size, but have an – implementation determined – maximum size. To incorporate this into our definitions is – conceptually – simple. We can let the protocol query the underlying component for its maximum number of elements, count the number of states generated so far and use this in checking the acceptability of an incoming "push" message. The examples below illustrate the general mechanisms to do this.

4.2 Context sensitivity

Now suppose that we want to adapt the definition of stacks such that the items pushed on it are non-negative numbers in non-descending order, i.e. the number on stack position *i* should be greater or equal to the number that is below it. We now would like the protocol to reflect this constraint, so that messages that want to push a wrong value are rejected.

Basically what we do here is introduce context sensitivity. A message is only accepted if its properties conform to properties of other messages that were parsed earlier. Modelling this particular case is relatively easy by adding parameters to our parsers and by passing the data values of messages around:

```
stack2   = many (push2 0)
push2 n = [ f | f <- sat (\(l,v,u,t) ->
                                  l == "push" && n <= nv v),
                r <- full2 (nv (value f))]
full2 n = [ r | f <- many (push2 n), r <- lmsg "pop" ]
```

The parser that handles "push" is given the number that is currently on the top of the stack as an argument (or zero if the stack is empty). A push message now has to have a number value that is larger than or equal to this value. This is coded in an application of `sat`. The new "top" number is passed on to `full` and the process continues.

4.3 Access control

A perhaps more appealing application of context-sensitivity occurs when we use protocols to encode access-control: constraints on who can send a message to this particular object. The stack protocol serves again as the vehicle for the example: we want to enforce that items that are pushed on a stack are removed by other objects than the ones that pushed them on the stack. So the sender of a message that pushed item *i* on the stack should be different from the sender of the corresponding pop. Again we use parameters to pass this context information around:

```
stack3   = many (push3 "arbitrary object-id")
push3 n = [ f | f <- lmsg "push",  r <- full3 (sender f)]
full3 n = [ r | f <- many (push3 n),
                r <- sat (\(l,v,u,t) -> l == "pop" && u /= n)]
```

The `sat` application in `full3` only succeeds if the sender is not equal to the sender that pushed the item and that was passed as an argument to `full3`.

Obviously, a similar approach could be used to add access control to the file protocol that was introduced earlier. Since we only consider the sending object as part of the message we cannot express constraints on the class of the sender (e.g. all the push messages should come from objects of class X). Of course, this could be repaired easily by adding the class information to the message.

4.4 The voting protocol

As a final example in this section we discuss a fairly complex protocol that can handle a voting session on a particular topic. The definition given here is an adapted version of an example introduced in [Florijn94].

The idea is that a software component performs the bookkeeping surrounding a vote on a particular topic. The vote is broken down into three "phases". In the preparation phase the initiator of the vote defines the topic, the choices that voters can make, the participants that can cast a vote, the time at which voting can start and the deadline before which votes should be cast. In phase two, the actual votes are cast. It is a bounded period of time (from start to deadline). Of course, each participant can only vote once, and can only chose one of the given options. Voting may or may not be compulsory. In the final phase, the results of the vote can be inspected by the participants and by the initiator. We assume that intermediate results cannot be inspected.

The top level parser of this protocol is straightforward. It decomposes into parsers for the three phases. Likewise, the parser for the preparation phase matches the messages that define the various initial settings.

```
voting  = [ (p,v,i) | p <- prepare, v <- votes p,
                      i <- inspect p ]

prepare = [ (tv (value t)),ps,cs,tv (value s),tv (value d)) |
                t  <- lmsg "topic", ps <- voters,
                cs <- choices, s  <- lmsg "starttime",
                d  <- lmsg "deadline" ]


voters  = [ Set (map value x) | x <- many1 (lmsg "voter") ]
choices = [ Set (map value x) | x <- many1 (lmsg "choice") ]
```

Note that we represent senders and timestamps (the starting time and the deadline) as strings. Also we assume that the topic and the choices that can be made are represented as strings. Furthermore, this definition is not complete. It lacks a check for the fact that voters and choices do not contain duplicates and there is no access control check for the fact that all these messages are sent by one particular sender, the initiator object. Adding these constraints is easy, so we will not do it here.

Once the preparation phase is completed, we enter the voting-phase. Basically this involves the reception of "vote" messages from all voters identified earlier (assuming that voting is compulsory). One constraint on these votes is that they must occur within the time-frame identified. This is a variation on the constraints defined above. The other constraints to check are that each voter votes precisely once, and that each vote makes a choice from the set of choices identified earlier. We model this by "generating" suitable combinations of parsers using the sets of voters and choices defined during preparation.

The elementary parser is `vote` which succeeds when a particular choice c is made by a particular voter object p within the time-frame between b and e:

```
vote p c b e = sat (\(l,v,u,t) ->
          l == "vote" && p == u && v == c && b <= t && t <= e)
```

Given a particular voter p, the set of choices cs, the start time b and the deadline e, we can define a parser that accepts one of all possible choices as follows (and returns the message that was parsed succesfully):

```
pvote p cs b e = oneof [ vote p c b e | c <- sv cs]
```

Note that the list comprehension here generates a list of parsers, handled by `oneof`. Similarly, we can generate a series of parsers for the set of participants and thus define the version of the parser for `votes` that returns the set of choices made from the vote messages.

```
votes (t,ps,cs,b,e) = [ Set (map value x) |
            x <- serie [ pvote (tv p) cs b e | p <- sv ps]]
```

The problem with this definition is that we use the `serie` combinator. The votes by the participants now are expected to arrive in a particular order. Of course, this is undesired: we want to accept all possible permutations of votes [Cameron94]. We need a new combinator `unordered` that will match a list of parsers occurring in arbitrary order. Defining `unordered` is conceptually simple. We generate all possible permutations of the list of parsers, put each permutation in a `serie`, and combine the set of permutations as alternatives using `orelse`, relying on lazy evaluation to optimize the actual generation and evaluation of alternatives:

```
unordered ps = oneof (map serie (perms ps))
             where gaps [] = []
                   gaps (x:xs) = xs:(map (x:) (gaps xs))
                   perms [] = [[]]
                   perms l = [ x:p | (x,r) <- zip l (gaps l),
                                     p <- perms r]
```

Note that there is one flaw in this solution: the order of the results is not fixed but depends on the alternative actually found. To correct this, we must add functions to re-order each alternative once it has been matched. We will not discuss this further here. Applying `unordered` in our example completes the definition of `votes`:

```
votes (t,ps,cs,b,e) = [ Set (map value x) |
       x <- unordered [ pvote (tv p) cs b e | p <- sv ps]]
```

The restriction on compulsory voting can be removed easily by using `option`:

```
votes2 (t,ps,cs,b,e) = [ Set (map value (concat x)) |
         x <- unordered [ option (pvote (tv p) cs b e) |
                          p <- sv ps]]
```

The final part of this protocol is the parser for `inspect`. It is straightforward, omitting the check on the senders and on the time stamp that would be similar to examples provided earlier:

```
inspect p  = many (lmsg "inspect")
```

4.5 Discussion

At this point it is worthwile to reflect on the work so far. A crucial element is that protocols are defined indepently of any implementation. This means that most of the problems surrounding the inheritance anomaly can be avoided [Matsuoka93,

Meseguer93]. For example, creating a sub-class of stacks with a method pop2 that pops the two top elements of a stack implies the definition of the new method and the definition a new protocol for this sub-class, but does not affect any *existing* definitions in the super-class. A similar approach applies when adding a sub-class with a method ppop which can not be invoked directly after a preceding push operation.

Another observation to be made is that the variations on stacks discussed above only differ in the protocol, and not in the visible interface methods or their semantics. This holds an interesting promise for creating different classes by vaiation in the protocol, and not in the actual service implementation.

5 Parallellism and shared state

The previous section has illustrated some of the expressive power of functional parsers. We have modelled non-trivial protocols that involve context-sensitivity and dynamic instantiation to express constraints on data, time and senders. In this section we discuss an extension to our formalism, i.e. the ability to express parallellism in the protocol. This is needed among others for modelling objects that support multiple roles. Based on this we also introduce shared state among parallel parsers.

5.1 The need for parallellism

The parser for voting assumes that inspection of the results is only done after all the votes have been cast, or more precisely, after the deadline for voting has passed. Suppose however that we want to allow such inspection to occur in parallel with the casting of votes. How can we model this?

At first it would seem that the `unordered` combinator could be of use here. Using the existing definitions of `votes` and `inspect`, the top-level procedure for `voting` would become:

```
voting = [ (p,v, i)| p <- prepare,
                     [v,i] <- unordered [votes p, inspect p]]
```

Unfortunately, this does not work. `Unordered` matches any permutation of its argument parsers, but does not provide interleaving of the parsers invoked by them, which is what we need. We want to allow the messages accepted by `inspect` and all the parser it invokes to occur arbitrarily interleaved with the "vote" messages accepted by `votes` and its descendants.

What is needed therefore is a combinator `par` that takes two parsers, matches arbitrary interleavings of the items matched by the two parsers and returns the results in an orderly fashion. The type signature of this combinator is:

```
par :: Parser i a -> Parser i b -> Parser i (a,b)
```

Using this combinator, the definition of voting with parallel inspection now becomes:

```
voting = [ (p,v,i) | p <- prepare,
                     (v,i) <- votes p `par` inspect p ]
```

The implementation of this combinator requires a revision of the implementation of the parsers. Roughly speaking it means that parsers internally do not return values directly, but *continuations* whose execution is interleaved in all possible ways using

the definition of the merge-operator from process algebra [Baeten90]. The details of this are beyond the scope of this paper, but are described fully in [Florijn94]. There we also define a constructor `atomic`, that prevents interleaving of its argument parser and turns it into an atomic action again.

The voting protocol is not the only case in which parallellism is needed. In [Florijn94] for example we discuss an electronic-mail protocol in which messages can be added to arbitrarily nested discussion threads. Here we give two other examples: a file-protocol with a single writer and multiple readers, and a shared drawing object with coordination control.

5.2 Parallel file access

First consider the file-protocol defined earlier. It models exclusive access to a file object for the client that has opened it. In practice, we may want a more elaborate definition, where multiple clients can have a file opened for read-access, but only one client at a time can have a file opened for write-access. The visible behaviour of such an object can be seen as the parallel composition of two protocols: one that handles writers and one for readers. So:

```
file = reading `par` writing
```

The protocol that controls the writers is simple:

```
writing = many writer

writer = [ o | o  <- lmsg "openw",
               rw <- many (oneof [read (sender o),
                                  write (sender o)]),
               c  <- close (sender o)]

read n  = sat (\(l,v,u,t) -> l == "read" && u == n)
write n = sat (\(l,v,u,t) -> l == "write" && u == n)
close n = sat (\(l,v,u,t) -> l == "close" && u == n)
```

To model multiple readers, we need the parallel composition again. After an open for read has occurred, a parallel parser is "forked" to handle other opens for read:

```
reading = [ o | o <- lmsg "openr",
                _ <- reader (sender o) `par` reading]

reader n = [ (r,c) | r <- many (read n), c <- close n ]
```

5.3 A shared drawing object

As another example, consider a shared drawing object. It provides operations to access and manipulate the drawing, e.g. to add graphical objects to the drawing or to remove them. Multiple client objects have read access but only one client has write access (the owner). Write permission is not handled by the clients themselves, but by a separate "moderator" object. It can transfer ownership from one client to another dynamically.

The shared drawing object performs two roles: on the one hand it offers the operations to access and update the drawing to the client objects, while on the other hand it offers operations to change ownership to the moderator object. Using parallel

composition we get the following definition for the top-level:

```
drawing = [(i,w,m) | i <- init,
                     (w,m) <- work (snd i) `par` mod (fst i)]
```

In the initialization we identify the moderator and the initial owner and return them for use by other parsers:

```
init = [ (txtval m, txtval o) | m <- lmsg "moderator",
                                o <- lmsg "owner" ]
```

The work involves many read or update operations, while the moderate part involves changes of ownership.

```
work o = many (oneof [ readop, changeop o])

changeop o = sat (\(l,v,u,t) -> l == "change" && u == o)

mod m = many (sat (\(l,v,u,t) -> l == "transfer" && u == m)
```

The flaw in this protocol is obvious immediately. We want the constraint in the `work` parser to be affected by a change in owner that is handled by the moderator parser. While parameter passing can be used to pass the initial owner, it is insufficient for communication between two parallel parsers. What is needed in this case is some kind of state (e.g. an environment: a mapping from names to values) that is shared between these two parallel parsers.

Implementing the shared state means that we have to generalize the "input state" handled by parsers. Instead of assuming a list of items, it now becomes a more complex entity consisting of the items and an environment. This composite state object is then passed from one parser to another. Again, the implementation of this is beyond the scope of this paper, but defined in [Florijn94]. Here we focus on how the mechanisms provided there are used to model this particular case.

The shared state is a modelled as an environment, a mapping from labels (strings) to values (of type Value). It is manipulated through two functions that are modelled as parsers. The first, `eadd`, adds a label, value tuple to the environment. The second, `elup`, searches the environment for a given label and returns the corresponding value. Using these two parsers, we can re-model the protocol for the shared drawing object as follows:

```
drawing = [ (i,w,m) | i <- ini,(w,m) <- work `par` moderate]

ini     = [ m | m <- lmsg "moderator", o <- lmsg "owner",
                _ <- eadd ("moderator", value m),
                _ <- eadd ("owner", value o)]

work     = many (oneof [ readop, changeop])
readop   = lmsg "read"
changeop = atomic ([c | o <- elup "owner",
                        c <- sat (\(l,v,u,t) ->
                                l == "change" && u == tv o)])

moderate = many transfer
transfer = atomic  ([c | m <- elup "moderator",
                         c <- sat (\(l,v,u,t) ->
                                   l == "transfer" && u == tv m),
                         o <- eadd ("owner", value c)])
```

Note that we use atomic here to guarantee that accessing and updating the environment is an atomic action whose constituents cannot be interleaved with other actions.

5.4 Discussion

One of the main advantages of the parallel combinator is to combine different sub-protocols into one. This could prove useful, for example, in the case of multiple inheritance. Suppose we want to combine the stack service defined earlier with a locking behaviour where a locked stack can not accept any messages besides an unlock message [Matsuoka93]. The locking behaviour can be described – essentially – by the following protocol:

```
locking = many dolock
dolock  = lmsg "lock" `seq` lmsg "unlock"
```

Of course, more elaborate checks could be added, e.g. to ensure that the unlock is performed by the same sender as the lock message.

Combining the locking service with the stack service on the protocol level now is simple (assuming that `stack` refers to the protocol of the stack service):

```
lockedstack = stack `par` atomic (locking)
```

Note that the `atomic` combinator is needed here to ensure that no intermediate stack operations are performed during locking.

There are two points worth mentioning here. First, the locking behaviour is pure coordination, so, no actual class definition for locking is necessary; adding in the protocol definition suffices. Obviously, this also holds for some of the examples above. The moderated drawing protocol, for example, includes messages used purely for coordination; they do not have to correpond to interface methods of the underlying object. Second, combining protocols is not always this easy, especially if more elaborate synchronization is needed between the protocols that are combined. While the shared state between parallel protocols offers the basic mechanisms to handle this, more study is needed as to how easy combinations can be made.

Finally, the examples given in this section (and also in [Florijn94]) suggest that our protocol formalism could be used to describe cooordination among multiple objects, e.g. clients, a server and work-allocators or moderators. In fact, this is one of the main motives of our research; we want to define and implement a flexible coordination system for hybrid processes in which human activities are combined with automated steps.

6 Incremental protocol checking

In the previous sections we have seen how message-parsers can be used to model even complex object protocols in a fairly straightforward and elegant way. The parsers are executable: when run with a list of input messages a message parser will return a list of successful parses, each of which consists of the result produced and the remainder of the input that was not handled. The non-determinism in the grammar is thus handled by attempting all possible parses. However, if we want to consider parsers as an implementation tool, this is not acceptable. We need a situation where for each incoming message it is immediately clear whether or not it is acceptable. In

other words, we want to make our parsers deterministic and incremental.

In this section we show that this can be done. We give a simple implementation that doesn't cause changes to any of the protocols defined earlier. That is, while the implementation of parsers is changed, the way in which they are used and combined remains unaffected because we use monads. The key to the solution is that if there is no input available, a parser should return a continuation, a construct that can be evaluated whenever new input becomes available and that continues the parsing process. These continuations are collected and handled by a special driver function that handles the incoming input. Note that we apply incremental checking here to parsers without parallellism, but a similar approach also works for parsers with parallellism.

6.1 Parsers with continuations

We adapt the basic type of parsers given earlier to accommodate continuations. The new definition is:

```
type Parser s a = s -> [ PStat a s ]

data PStat  a s  = Done s a
                 | Maybe s (Parser s a)
```

A parser thus not directly returns a result, but a value of type `PStat`. Such a value is either a final result (constructor `Done`) or a continuation (constructor `Maybe`). A result is a combination of the value and the resulting (input) state. The continuation is a combination of the input state and a parser that still has to be performed. Such a value is only returned if we reach the (temporary) end of the input stream. Whenever new input becomes available, the continuation can be reinvoked.

A few of our basic parsers (introduced in section 2) have to be adapted to accommodate the new definitions. Since their type signatures remain the same these have been omitted. Also we introduce a new parser maybe:

```
okay v  s = [Done s v]
fail s = []

(p1 `into` p2) xs = [ bi |
                      pi <- p1 xs,
                      bi <- case pi of
                              Maybe i' p' -> maybe (p' `into` p2) i'
                              Done  i' a  -> (p2 a) i' ]

(p1 `orelse` p2) xs = p1 xs ++ p2 xs

maybe :: Parser s a -> Parser s a
maybe p s = [Maybe s p]
```

The essential part is the definition of `into` since `into` is used to compose parsers out of smaller ones. It shows how the result of applying the first parser affects further handling. A `Maybe` result means that the whole `into` construct is not completed, but should be retried at a later point. A `Done` result means that the result of `into` is the same as the result of applying the second parser on the adapted input. This can again be a `Done` or a `Maybe`.

Given these function definitions we can define our new parsers as monads. In fact we can use the same definitions as used before. Since all the protocol parsers defined

in the previous sections are monad expressions, this means that these definitions still work, but now with a different implementation.

6.2 Input representation

`Maybe` values are created when the input stream is temporarily empty, i.e. when no messages are currently available. To abstract from a particular representation of the input state we define a new constructor class for it. A constructor class defines a class of types by defining a number of functions that should be implemented on types that are instances of the class (as with the monad definitions before). In this particular case we introduce functions that check whether the input is currently empty, that deliver the next element (and the adapted state) and that add items to the input. Also, we introduce the list as an instance of this class.

```
class ParsingState s where
        next ::  s a -> (a,s a)
        unput :: a -> s a -> s a
        isempty :: s a -> Bool

instance ParsingState [] where
        next (i:is) = (i,is)
        unput a xs = append xs a where
                  append (x:xs) i = append xs i
                  append [] i = [i]
        isempty [] = True
        isempty x = False
```

Using these definitions we can give a new implementation of `item`, the basic parser that returns the next item in the input or that returns a continuation if the input is temporarily empty. Of course, now the input is a type in the `ParsingState` class:

```
item ::  ParsingState t => Parser (t a) a
item s = let (x,y) = next s
          in  if (isempty s)
              then maybe item s
              else okay x y
```

6.3 Incremental checking

The final part in providing incremental checking is the implementation of a driver routine that handles incoming messages and the set of continuations and determines whether the next message is acceptable or not. The basic strategy is to derive a first set of continuations based on the empty input. Then, an incoming message is added to the input states of all continuations and these continuations are evaluated. If this does not result in any new continuations, the parser may have completed successfully or not. In the first case, the original parser could be restarted. In the second case, the message is refused and the previous set of continuations is used for the next message.

In the implementation below we represent the decisions as text strings that are returned. Also we assume that the whole list of incoming messages is available to the driver routine which handles them one at a time. In this way we could check for an "end of input" condition indicating that no further messages will arrive (these checks have been omitted here, so we assume an infinite list of input items). It is trivial to

adapt this function to handle real input, e.g. typed by a user or coming from a file. We would then have to add a conversion from text strings to the message tuple. This is not relevant for the point we try to make, so it is omitted here.

```
iparse p s input = doiparse (parseconts (parse p s)) input
  where doincrparse ps inp =
        let -- ps is the current set of continuations
            -- h is the input item handled next
            -- t is the remainder of the input
            -- res is the set of results from applying the
            --     continuations in ps to this input item

           (h:t) = inp
           res = concat (map spa ps)
                 where spa (p',s') = parse p' (unput h s')
        in
         if (null (parseconts res))  -- no continuations?
         then if (null (parseresults res)) -- no termination?
              then "Msg refuse" ++  (doiparse ps t)
              else "Terminated; should restart protocol"
         else "Msg accept" ++ (doiparse (parseconts res) t)
```

The definition is straightforward. It relies on a few auxiliary functions that actually run a parser on an input state or that select particular types of responses (results or continuations) which have consumed all the input. We give the definitions here for sake of completeness:

```
parse p xs = [ pi | pi <- p xs]

parseresults ps = concat (map x ps)
                  where x (Done s r)    | isempty s = [(r,s)]
                                        | otherwise = []
                        x _ = []

parseconts ps = concat (map x ps)
                  where x (Maybe s p)   | isempty s = [(p,s)]
                                        | otherwise = []
                        x _ = []
```

7 Conclusions

This paper has explored techniques for modelling complex service protocols independently from any particular service implementation. A protocol provides clients of a software component with information as to when and how particular interface operations can be perfomed. On the other hand, a protocol can be of use for a service implementer, if checking of protocol conformance is automated.

We propose a grammar-oriented approach, where protocols are separate entities in a service definition, and are defined as grammars. Dynamic checking of protocols is then possible by generating suitable recognizers for the grammar. To explore the constructs that are needed we have defined protocols as (functional) parsers that impose constraints on (sequences of) incoming messages. The available constructs make it easy to model context-sensitivity and to constrain properties of messages, like the sender or a time-stamp. The non-determinism provided by parsers has been

extended with parallellism and shared state. As demonstrated in section 6, incremental checking of protocols can be provided, making the technique, at least in principle, suitable for dynamic checking in actual systems.

The examples we have given show that fairly complex protocols can be modelled in an elegant and straightforward way. The advantages of treating protocols as independent, but executable, specifications have also become obvious. By changing the protocol we can change the visual behaviour of a class (or even an individual object) without any changes to the implementation. In fact, the behaviour of a component that is purely dedicated to coordination (such as the ownership of a shared drawing, or the locking behaviour) does not require a service implementation, but can be handled directly and exclusively with protocols. Similarly, combinations of protocols (using the `par` combinator), occuring e.g. in multiple inheritance, can be modelled. So, in this way, some of the inheritance anomalies can be avoided. Going even further – and in fact one of the main motives behind our research – the approach allows us to coordinate the behaviour of multiple objects.

While modelling protocols in this fashion is attractive, there are still several open ends to be explored and problems to be dealt with. First, while Gofer provides an excellent framework to experiment with these ideas, it is nothing more than that. Ultimately we want a high-level language or notation for describing protocols. This paper suggests some of the constructs that would have to be included, like context-sensitivity and dependencies among parallel parsers that communicate via the shared environment. Besides adding syntactic sugar, defining a protocol language also implies finding a balance between expressiveness and efficiency. The availability of non-determinism combined with the parallel combinator makes the current protocol formalism powerful, but dynamic checking – at least in our naive implementation – too expensive for practical applications where decisions on message acceptation have to be made within a couple of milliseconds. We are currently looking at more efficient implementation schemes, but it is possible that restrictions to the formalism, e.g. by enforcing that the set of message-selectors accepted by two parallel parsers must be disjunct, are needed to make a highly efficient implementation possible. In a similar vein, the desire to provide static checking could lead to further restrictions regarding the expressive power.

Since the protocol is fully independent from the implementation, protocol state changes are only caused by messages. State changes caused by the implementation should thus be modelled as messages sent to the object itself. One example would be a file object that should be closed automatically if the client hasn't accessed it within a given time frame. The object should send the close message to itself at the appropriate time. It remains to be seen, however, to what extent this separation of concerns can be maintained. Can all preconditions be expressed in the protocol, or should the implementation of an operation be allowed to refuse a message that is accepted by the protocol? Implementing this would be trivial: we can include the result of handling an accepted message in the decision whether or not a state-change in the incremental checking algorithm.

One obvious way to extend the scope of our formalism is to include more message properties. In addition to the class of the senders we could include for example the location (i.e. machine) of the sender, to enforce a particular communication strategy in a distributed system. In a similar fashion it might be reasonable to make certain properties of the current object (like the identity or the location) available to model constraints.

For a more direct integration with object-oriented languages we need a more explicit notion of protocol inheritance. The idea would be that a sub-protocol could give new definitions of certain parse functions, add additional messages that can be accepted or impose additional constraints. Another interesting extension would be to introduce reflection and to make the protocol (and its current state) available for querying by clients. Similar to the `respondsTo:` method in Smalltalk, a client could ask the protocol of an object whether a particular message currently is acceptable or not. Obivously, there is a strong relationship between protocols for dynamic checking and reflective object-oriented systems. Our protocol can be seen as a function based on the reification of the message delivery primitive. By the same token, a protocol could be extended with provisions to handle certain meta-messages itself like finding out who performed the last message send, or how many messages haven been handled by the object.

Acknowledgements

The work described here is a part of the Ariadne project where we investigate computer support for hybrid office processes, which are complex information-processing activities involving contributions by people and tools. Doaitse Swierstra initiated this project and suggested the use of functional parsers to model processes. Jeroen Fokker provided an introduction to combinator parsing. Erik Meijer pointed the way in the introduction of the parallel combinator and the use of monads. Many thanks go to the anonymous referees for pointing me to the existing work in this field.

References

[Aksit92] M. Aksit, L. Bergmans and S. Vural, "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach", in *Proceedings ECOOP'92*, Utrecht, the Netherlands, Springer Lecture Notes in Computer Science vol. 615, O. Lehrmann Madsen (Ed), 1992.

[Baeten90] J. Baeten (Ed.), *Applications of Process Algebra*, vol. 17, Cambridge Tracts in Theorethical Computer Science. Cambridge University Press, 1990.

[Bos89] J. van den Bos and C. Laffra, "PROCOL A Parallel Object Language with Protocols," in *Proceedings of the 1989 OOPSLA Conference,* New Orleans, Louisiana, September 1989.

[Cameron94] Robert D. Cameron, "Extending Context Free Grammars with Permutation Phrases," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, March-December 1994.

[Florijn94] Gert Florijn, "Modelling Office Processes with Functional Parsers", Technical Report, Utrecht University, November 1994.

[Hutton92] Graham Hutton, "Higher-Order Functions for Parsing," *Journal of Functional Programming*, vol. 2, no. 3, July 1992.

[Jones93] Mark Jones, "A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorfism," in *Proceedings FPCA'93,* 1993.

[Laffra92] C. Laffra, "PROCOL: a Concurrent Object Language with Protocols, Delegation, Persistence and Constraints," Ph.D. thesis, Erasmus Universiteit, Rotterdam, the Netherlands, May 1992.

[Matsuoka93] S. Matsuoka and A. Yonezawa, Highly Efficient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Languages, in *Proceedings of the 1993 OOPSLA Conference,* Andreas Paepcke (Ed.), Washington D.C., USA, 1993.

[Meyer88] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Meseguer93] J. Meseguer, "Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming", in *Proceedings of ECOOP'93,* Springer Verlag, Lecture Notes in Computer Science, 1993.

[Nierstrasz93] Oscar Nierstrasz, "Regular Types for Active Objects," in *Proceedings of the 1993 OOPSLA Conference,* Andreas Paepcke (Ed.), Washington D.C., USA, 1993.

[Perry89] D.E. Perry, "The Inscape Environment," in *Proceedings 11th International Conference on Software Engineering,* Pittsburgh, Pennsylvania, May 1989, pp. 2–12.

[Wadler90] P. Wadler, "Comprehending Monads," in *ACM Conference on LISP and Functional Programming,* Nice, France, June 1990.

[Wadler92] P. Wadler, "The Essence of Functional Programming," in *19th Annual Symposium on Principles of Programming Languages,* Santa Fe, New Mexico, January 1992.

[Weide91] B.H. Weide, W.F. Ogden, and S.H. Zweben, "Reusable Software Components," in *Advances in Computers*, M.C. Yovits, Ed. Academic Press, 1991.