Object Protocols for Distributed Systems

Prakash Krishnamurthy and Paolo A.G. Sivilotti
Department of Computer and Information Science
The Ohio State University
2015 Neil Avenue, Columbus, OH 43210-1277 USA
$\{$*krishnam, paolo*$\}$*@cis.ohio-state.edu*

1. Problem & Motivation

Safety properties play a critical role in the specification of software systems. Intuitively, they specify that “bad things” do not happen. An example would be to specify that a file is open before an attempt to read on the file is made. They specify properties of the system whose violations can be detected (e.g., calling read before opening the file). Hence, if proper validation infrastructure is provided, they help us develop robust software. One important and often employed technique for specifying safety is through the specification of protocols. They are intended to capture the valid sequence of operations permissible on a component. In this work, we address using protocols for specifying safety in distributed, concurrent systems.

Information systems have become increasingly distributed. Frequently, due to the complexity of interactions in distributed systems, failures and partial errors are hard to detect and fix. Also, static verification of distributed systems is often impractical and prohibitively expensive. Hence for developing reliable and robust distributed systems, the development of tools and techniques to provide efficient support for specifying and tracking of safety properties becomes important. Our work aims at developing a practical approach to efficiently specify and monitor protocols in distributed systems.

2. Background & Related Work

The specification of distributed systems, with safety and progress is a well established approach. Safety properties have been included in systems in a number of ways. One way is to specify them as pre and post conditions on the interface methods. Another important way of specifying safety is to specify properties at the object level as opposed to method level. For example, specifying the valid sequences of method calls on the object is done at the object level and does not belong to any particular method. As such a specification identifies the protocol, that the clients need to conform to while using the component, it is typically referred to as a protocol specification or simply protocol.

Applications of protocols include specification and validation of operations, verification of behavioral subtyping (by defining subtyping based on protocol conformance), static code checking etc., In this paper we will primarily concentrate on application of protocols for specification and validation of operations and include support for composing component specifications.

Although, in theory, protocol specifications could be based on the semantics of the messages exchanged, existing approaches to protocol specifications are primarily based on just the method names [6]. They support run time verification by modifying source code and/or compiler. However our approach differs from this body of work in its support for parameters and protocol state. Our approach is also easily integrated with CORBA based component development.

Specification languages for supporting message parameters exist [3, 1] . In [1], Florijn defines the more general problem of applying protocols based on message context where the protocols could be based on the properties of the message (owner, time-stamp etc..). Our approach is similar to the approach in [1]. However, important differences exist in our treatment of protocols. We explicitly include support for both *sends* and *receives*, allow local states in protocols and support specification on method parameters. Also unlike [2, 4] concurrent clients and multiple concurrent protocol instances are readily supported without the need to bind the component to a specific channel/process.

3. Approach & Uniqueness

Although expressing protocols based on the messages exchanged has been a fairly standardized practice - the definition of what the message is has been subject to varying interpretations. Tool support for real world systems have generally based their specification on just the name of the method being invoked rather than on the actual content. Protocols instances are distinguished, where required, based on the client involved and not on the message content.

The expressivity of such protocols is often restricted as decisions cannot be based on the content. One simple observation that can be made is that a component's actions are not only determined by the method name but are usually influenced by the content of the message too. Hence we propose a method of capturing object interactions in a distributed environment (at run-time) by extending the basic protocol specifications by supporting parameters (data exchanged) too. We call such protocols *parameterized protocols*.

Our design decisions for a protocol system for distributed, concurrent systems is based on the following observations.

1. Components in a distributed system interact primarily through message passing. Hence capturing the message content in protocols would help strengthen the specification on the components.

2. Frequently different clients could share the same protocol with the component (e.g., a task queue) or the component could be operating on different protocols with its clients (e.g. a file component supporting read-only and read-write protocols) at the same time. Hence support for clients to interleave executions is required.

3. In describing distributed systems we frequently resort to specifying an order on the message flow across the components in the system. Thus to identify the causal relationships we need a way to compose specifications. Hence the need to capture both *sends* and *receives* and not just the valid method invocations on the component.

4. The validation of the protocol cannot be independently performed on the client at compile time as the involvement of other clients and message delays could still cause protocol violations. Hence efficient run-time support for protocol validation needs to be provided.

To support message context we extend the protocol specification to include explicit support for handling parameter values. This gives us a way to distinguish between multiple protocol instances based on the parameters of the method. Thus support for multiple protocol instances is achieved. For addressing the message flow problem, a CORBA interceptor based approach for automatic tracking of both *sends* and *receives* is discussed.

We extend the protocol specification technique of labeled transition systems introduced by Nierstrasz [5] and describe the design of the framework for tracking such specifications. Automatic validation of protocols in the context of CORBA based components is addressed.

A simple and efficient tool design for tracking such protocols is presented for CORBA based components.

4. Brief Description

As discussed in [5] allowed sequences of operations cannot always be expressed as a single regular expression. Consider the example of a stack component, where a pop operation is only acceptable if the stack is not empty. To model this protocol as a deterministic finite state machine - we would need an infinite number of states. As this makes harder to reason about satisfiability, it has been proposed in [5] to model this by a non-deterministic finite state machine.

Following the idea of Nierstrasz we use a labeled transition system over the methods of the interface to describe the protocol. Non-determinism is handled by adding guards to protocol expressions like in [6] and by allowing protocol states to have attributes.

The protocols are more than just a sequence of transition definitions. They are allowed to have internal variables akin to abstract state of the component. Such states are specific to protocol instances and transitions can update and refer to these variables.

To identify various protocol instances we allow transitions to refer to the incoming parameter values too. This feature can be used to distinguish multiple protocol instances and thereby allowing multiple clients to interact with the component.

As the protocols above are very much like parameterized classes in the fact that they provide for dynamic specialization based on the parameter, we call such protocols *parameterized protocols*. As multiple instances of the same protocol can be up and running at the same time. Hence overlap of protocol executions (interleaved execution) is supported.

To illustrate our protocol specification consider a Stack component supporting the interface show in Figure 1. The non-deterministic finite state automata for defining the permitted sequence of operations is given in Figure 2. Figure 3, illustrates an example application for defining the interface with our language. The example specification highlights the basic features of our specification system and illustrates practical application of our specification mechanism. First, by parameterizing on stack name, we find that the component can export multiple stacks, also it allows more than one client to share the same stack. The guards (on init and size) are used to ensure that the system is in a state where it can accept the message and the state attribute (for empty) resolves the non-determinism when a pop operation is called.

```
interface Stack {

  // Allocates memory for stack
  void create (char* name, int size);

  // Standard Operations
  void push (char* name, int val);
  int pop (char* name, int val);

  // Deallocate
  void delete (char* name);

 }
```

Figure 1. Stack Component Interface

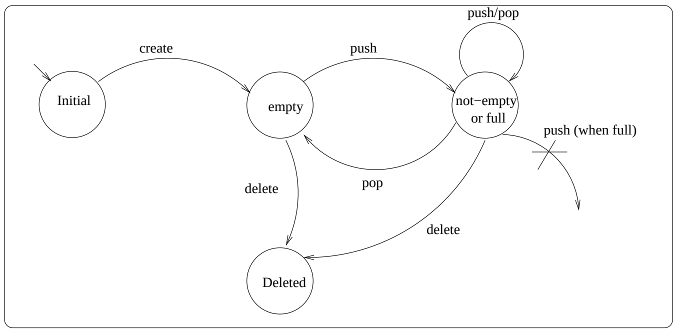

Figure 2. Stack component specification as a NFA

The salient features of our approach include :

Based on abstract state: Our specifications are based on the abstract state of the component as opposed to the concrete implementation state.

Explicit support for parameters: Specifying protocol transitions based on parameters is supported

Support for local state: Protocols have been extended by providing support for local state while at the same time maintaining testability

Support for multiple protocol instances: Default support for multiple instantiations of protocols to co-exist at the same time is provided.

Support for associating states with predicates: Protocol states can be associated with predicates, thus providing a way to capture non-determinism.

Support for guards in transitions: Guards to selectively enable/disable transitions is supported improving expressive power of the specification.

```
// Parameterized Protocol Specification for the Stack component
protocol stack param {
 state start empty := (size = 0) ;//Start State
 state final not_existent ; //Final State
 state not_empty ; //Associating states with boolean expr

 local int max_size;       // local state
 local int size := 0;      // local state with initializer
 local bool init := false;
 local string name;        // instance identifier

 // initializing the parameter
 <empty> create (name := #1, max_size := #2) <empty, init = true>;

 // Parameter Check, Local variable Usage
 <empty> (init = true) ˆ push (name = #1)  <size++, not_empty> ;

 // Capturing non-determinism with guards
 <not_empty> (size < max_size) push (name = #1) <size++, not_empty> ;

 // Move to state without any out transitions
 // => instance can be garbage collected
 <*> (init) ˆ delete (name = #1) <not_existent>
}
```

Figure 3. Illustration of *Parameterized Protocol* **Specification for a Stack Component**

Support for capturing both message receives and sends: Explicit support for both message *sends* and *receives* is provided. The message sends are typed by the expected interface of the component at the other end.

As an application of our specification mechanism we have prototyped a tool for CORBA based component development. Here the specifications are made at the CORBA IDL level and a test harness is generated for validation of the incoming requests. A CORBA interceptor based approach for intercepting incoming and outgoing messages is currently being developed.

5. Results & Contributions

Safety is a fundamental part of the behavior of distributed systems and protocols are an important way for expressing safety properties in distributed systems. In this work we have discussed the problem of specifying protocols in distributed systems. An approach to specification with parameterized protocols using a labeled transition system with state support is proposed and explored. Run time tracking of the protocol specification is discussed and we present a tool for efficient run time tracking of protocols for CORBA based component development.

References

[1] G. Florijn. Object protocols as functional parsers. In W. Olthoff, editor, *Proceedings ECOOP'95*, pages 351–373, Aarhus, Denmark, 1995. Springer-Verlag.

[2] C. A. R. Hoare. Communicating sequential processes. In R. M. McKeag and A. M. Macnaghten, editors, *On the construction of programs – an advanced course*, pages 229–254. Cambridge University Press, 1980.

[3] C. Laffra and J. V. D. Bos. A concurrent object-oriented language with protocols delegation and constraints, 1991.

[4] R. R. Milner. *A calculus of communicating systems*, volume 92. Springer-Verlag Inc., New York, NY, USA, 1980.

[5] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[6] G. B. Sergey Butkevich, Marco Renedo and M. Young. Compiler and tool support for debugging object protocols. In *Eighth international symposium on Foundations of software engineering for twenty-first century applications*, pages 50 – 59, 2000.