2009 33rd Annual IEEE International Computer Software and Applications Conference

ProtoTalk: A Generative Software Engineering Framework for Prototyping Protocols in Smalltalk

Ali Razavi
Software Engineering Laboratory
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
arazavi@swen.uwaterloo.ca

Kostas Kontogiannis
Department of Electrical and Computer Engineering
Natl. Technical University of Athens
Athens, Greece 15780
kkontog@softlab.ntua.gr

Abstract

Network protocols are complex systems implemented by collections of equally complex software components. In many cases, the realization of such protocols requires extensive prototyping and experimentation with different alternative implementations. In this paper, we present ProtoTalk, a generative, domain-specific software framework that utilizes model driven software engineering principles for prototyping state and message driven protocols with emphasis on telecommunication and network protocols. The framework allows first, for modeling a variety of common protocol features by using mappings from state machines, sequence diagrams and packet encodings to ProtoTalk models, and second, for the consequent automatic generation of prototype Smalltalk code from the aforementioned ProtoTalk models. In this respect, the paper attempts to shed light on the use of generative model driven programming techniques within reflective object oriented programming languages and environments. As a proof of concept, we have specified in ProtoTalk and consequently generated in Smalltalk, several core features of the Session Initiation Protocol. [1]

Keywords

Software Engineering, Domain Specific Frameworks, Generative Programming, Protocol Development

1. Introduction

Network and telecommunication protocols encompass software intensive components that are usually intricate and expensive to design and implement. Object oriented programming on the other hand, has been established over the years as a robust technology to design, implement and deploy highly reusable and maintainable software systems. Although we believe that object oriented methodology is effective for improving the quality of protocols' code, there is still a significant gap between the specification of protocols in standards, and the programming constructs that implement such specifications in such languages as C++, Java and Smalltalk.

This paper is the summary of our attempt to investigate the use of generative programming techniques in dynamic and interactive object-oriented environments for the important domain of protocols. More specifically, we introduce ProtoTalk, a generative domain specific framework based on the Smalltalk programming language, for the domain of telecommunication and network protocols. ProtoTalk is implemented in the Squeak [1] dialect of Smalltalk, and utilizes the extensive collection of reflective facilities offered by Squeak to generate Smalltalk source code that implements the corresponding ProtoTalk specification.

Smalltalk is a dynamically typed object oriented programming language that typically features an interactive environment. Squeak is a multi-platform and open-source implementation of Smalltalk that is almost entirely built upon itself [1]. Since Smalltalk offers a broad range of different intercession and introspection facilities [2], [3], [4], it is often advocated as an apt candidate for hosting domain specific languages [3].

ProtoTalk promotes the reuse of domain knowledge by encapsulating that knowledge in its code generators. Furthermore, it aims to support interactive and incremental development, refinement and rapid prototyping of a wide spectrum of protocols from standards and specifications, in an expressive and flexible fashion. We envision ProtoTalk to be used by protocol designers for experimenting with various design choices and alternatives, before commencing a full-fledged implementation in a detailed and statically typed language, such as C or C++, which typically requires extensive time investment. Most telecommunication and network protocol standards include various degree of enforcement for different aspects of protocols. More specifically, these standards usually begin with precise definition of terms, such as *MUST*, *SHOULD*, *RECOMMENDED* and *MAY*, purported towards denoting the degree to which the said standards demand support for a certain concept. Many of the features of protocols are either optional or depend on the usage context, therefore the decision of whether (or how) they should be supported is left to the implementors. In this respect, ProtoTalk can be used for the experimentation and prototyping of various aspects of a protocol specification standard.

The rest of the paper is organized as follows: Section 2 discusses the rationale behind the decisions made in the design of ProtoTalk by summarizing the domain analysis that has been conducted. The design and architecture of ProtoTalk is discussed in Section 2. Section 3 presents the principal techniques used in the implementation of ProtoTalk. Section 4 demonstrates the usage of ProtoTalk for prototyping part of

1. This work was conducted at the University of Waterloo and was funded by CA Labs.

the Session Initial Protocol (SIP). Related work is reviewed in Section 5. Section 6 concludes and outlines the avenue for future work.

.

2. ProtoTalk

2.1. Background

Squeak is an open-source, multi-platform and interactive implementation of the Smalltalk programming environment. In the design of the proposed framework, expressiveness and flexibility were amongst the major concerns. In this respect, ProtoTalk provides a set of generic classes which offers protocol developers the advantages of object orientation and the use of an advanced dynamic development environment, such as the one offered by Squeak. This framework can be used to describe a wide spectrum of layered protocols, and for generating their corresponding Smalltalk implementations. Figure 1 depicts the dataflow of ProtoTalk specifications and the architectural dependencies of ProtoTalk's components with those of Squeak.

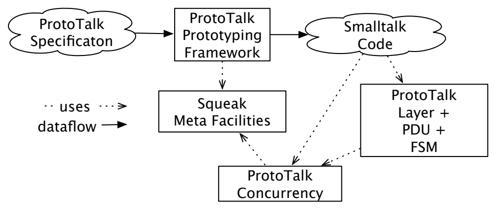


Fig. 1. ProtoTalk Architecture

By examining various telecommunication and network systems, one reckons that these systems share several common concepts, and are usually structured around a reference architectural style, the *layered architecture* [5]. Within each layer, policies and services offered to the layer above are generally implemented using reactive software entities, which run concurrently in an event-driven fashion. Event handling is often realized by the *implicit invocation* architectural style [5], which is often implemented using a variant of the *Observer* design pattern [6]. Furthermore, the behavior of protocols is commonly described using some variant of state machines, often combined with message sequence charts.

2.2. Layered Architectures in ProtoTalk

As discussed in the previous section, protocols are decomposed into hierarchical layers. This common theme is adopted and elaborated upon in ProtoTalk by allowing several protocol entities coexist in each layer of the system. Figure 2 illustrates the general layout of multiple layers in ProtoTalk. Each layer can have multiple *Protocol Entities*. Layer stacks in ProtoTalk are not merely static models of the system; to the contrary, they are part of the dynamic representations of the protocols. Each layer is an independent subsystem that provides a set of services through its *Service Access Point* or *SAP*, and uses its required services via its *Service Request Point* (*SRP*). One layer's *SAP* is attached to the *SRP* of the layer immediately underneath it.

Services offered by each layer are provided by internal *Protocol Entities*. Figure 3 shows a reference architecture for protocol entities in ProtoTalk. Protocols usually have some specific data units that have to be passed to the other communication end using the lower layers' services. Each protocol entity in ProtoTalk has a subsystem called *Packet Processor*. This subsystem is responsible for marshaling and de-marshaling of incoming and outgoing data units from the upper and lower layers. Packet Processors also monitor incoming control packets of the protocols in order to capture the raised events. Protocols are reactive entities that respond to such events by conducting different actions. The architectural style used for managing events in each protocol is the *implicit invocation*. Each protocol entity has a central dispatcher and can have several event sources, each associated with one or more event handlers. An event handler registers its interest in an event via *Protocol Dispatcher*. Upon occurrence of an event, the Protocol Dispatcher notifies the event handlers that have registered interest in the occurred event.

Services encompass the main functionality of protocols. Each service has a service interface that advertises its usage signature to other layers. Each protocol entity also relies on variety of external services for its operation, and therefore specifies a list of *required service interface*, items of which should each lexically match an advertised service from the lower layer during the generation of the layer stack. This sort of configuration is akin to the Lego toy blocks, in the sense that each block has an interface that can be attached only to those other components that can fill its gaps. When stacking layers, in case a required service is missing in a lower layer, ProtoTalk is able to dynamically and automatically modify the source code of the lower layer to add the skeletons for the missing service. An important aspect in communication protocols is the scheme used for establishing connections. Protocols are usually classified as either *connection-oriented* or *connection-less* [7]. Connection oriented protocols need to establish and maintain a dialog session with other peers. The subsystem responsible for management of sessions in ProtoTalk is called `Session Manager`. Each protocol can establish multiple sessions with different peers, using various underlying services.

2.3. Layers and Protocols definition

Layers and Protocol Entities in ProtoTalk are created from generic prototypes. These prototypes are configured for a specific application using Smalltalk's intercession features. Figure 4 depicts the UML diagram for the abstract layer and protocol entity classes. Using meta programming facilities of Smalltalk [2], these prototypes are incrementally transformed

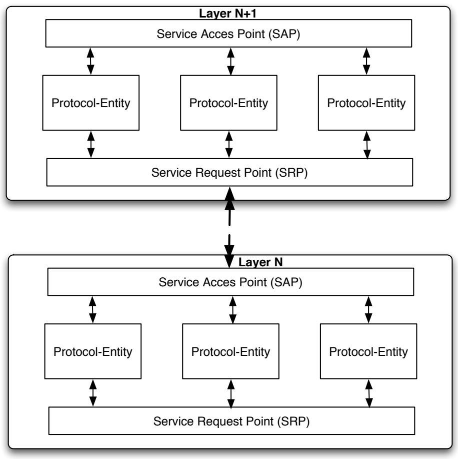


Fig. 2. Layered Architecture for Protocols

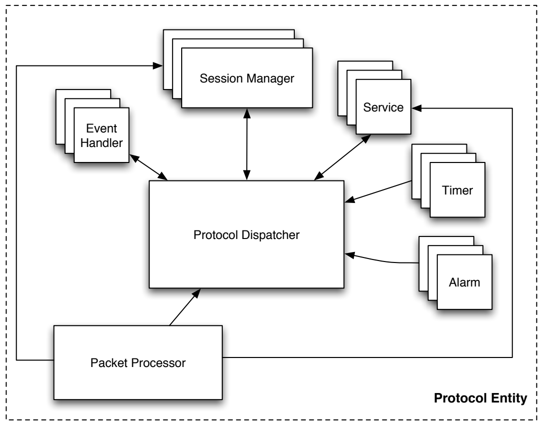


Fig. 3. Protocol Entity Architecture

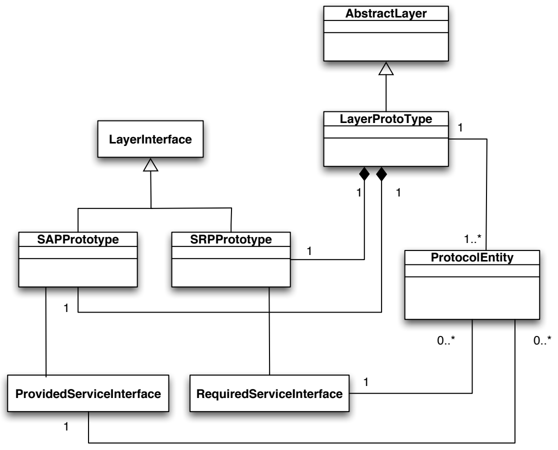


Fig. 4. Protocols and Layers' Classes

into refined customized classes, present at the execution time of the system. As the figure illustrates, the main interface for each protocol is provided by a `ProtocolEntity` object. A layer consists of several such `ProtocolEntitys`. The following piece of code exemplifies how these generic prototypes are used to create a protocol stack in ProtoTalk.

```
|newLayer newProtocol|
newLayer _ LayerPrototype new: #NewLayer
            onTopOf: oldLayer.
newProtocol _ ProtocolEntity new.
"configure the protocol by adding PDUs,
 Events, Sessions and Services"
newLayer addProtocolEntity: newProtocol useService:
    (newLayer service ofType: #Servicetype).
```

Each prototype class in ProtoTalk contains the required code for incremental self-configuration. For example, when the user instantiates a `LayerPrototype`, what actually happens in the background is that a new class is generated using the provided name, and is configured to run over the supplied lower layer. So the `newLayer` variable is an instance of the newly generated class, which is a subclass of `LayerPrototype`. Further details of the generation techniques used for implementing ProtoTalk is discussed in Section 3.

2.4. Concurrency and Synchronization

Concurrency is at the heart of any telecommunication system. The majority of specification and implementation methodologies for protocols provide means for expressing concurrent functional units. These range from low-level primitives provided by operating systems' APIs such as processes and threads as units of granularity for concurrency, to semaphore and monitors as synchronization building blocks. However, concurrent systems described in this level tend to become quite complex and difficult to understand and debug. There has been extensive research about expressing concurrent units in a level of abstraction higher than that of mainstream operating systems' APIs. A survey on several methods of denoting concurrency and distribution in various programming paradigms is presented in [8]. The concept of *Actors* in this regard is a specially intuitive approach to associate high-level concurrent abstractions with the object-oriented programming paradigm [9], [10]. The Actor Model is essentially a systematic way of merging the concept of execution context and abstract data types into one encapsulated unit. The general behavior of Agha's actor model is illustrated in Figure 5. The client context sends an asynchronous message to the *Active Object* which possesses its own thread of execution. The internal scheduler embedded inside the active object sends the received messages to the actual *behavior* object in a first-in, first-out basis. This scheme implies that actors are atomic objects which can process queued messages one at a time, but can also be run in concurrence with the senders of the messages.

Smalltalk provides several primitives for dealing with concurrency. This includes `Process` and `Semaphore`, as basic constructs, and `ProcessScheduler` as the underlying mechanism for scheduling processes. Processes in

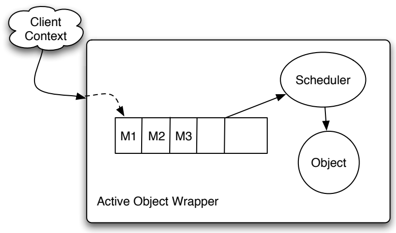


Fig. 5. Agha Actors' Architecture

Smalltalk are created upon invocation of the `fork` method on `BlockContext` instances. For the details of concurrent programming in Smalltalk see [11], [12], [3]. So far, there has been a rich spectrum of research work related to enhancing Smalltalk's concurrency mechanism with higher-level constructs. Some of the approaches are just class libraries on top of a standard Smalltalk implementation, while others are deployed on *ad hoc* modified virtual machines. A survey of existing solutions for supporting concurrency and distribution in Smalltalk can be found in [13]. The most well-known ones which we reviewed in the context of this work are: CST [14], ConcurrentSmalltalk [15] and Actalk [16].

CST is developed at MIT and is more suitable for distributed objects than concurrent ones. The syntax also resembles LISP syntax more than Smalltalk, which we think is not as expressive as Smalltalk's native syntax for our application. It also requires a modified virtual machine which is preferred to be avoided in the implementation of ProtoTalk.

ConcurrentSmalltalk developed at SONY labs [15], is one of the first attempts to bring the concept of asynchronous message passing to Smalltalk. It uses a customized virtual machine for introducing CBox as a primitive for receiving the return values of an asynchronous method invocation.

Actalk, developed by Briot [16], is an attempt to implement Agha's actor model using Smalltalk processes which run on top of a standard virtual machine. The concurrency model implemented in ProtoTalk, although in some ways is similar to Actalk, differs from it in at least one fundamental way. Actalk is primarily based on introducing concurrency via inheritance which requires the active objects to share a common superclass; hence results in an inheritance hierarchy for active objects, parallel to that of "normal" objects. Although there are some advantages to this approach, its main drawback is that it discourages (and somewhat hinders) exploitation of the extensive library of passive objects offered by Smalltalk. In contrast, ProtoTalk supports *metamorphosis* of normal objects into active ones upon need, by utilizing the proxy design pattern [6]. More specifically, as Figure 6 illustrates, `ActiveObjectProxy` instances can be used to *wrap* passive objects dynamically, and at runtime, so as to enhance them with concurrency constructs. This creates a twofold usage scheme for objects, i.e. active and passive, and thereby separates the concurrency concerns with the normal functionality that each object provides. `ActiveObjectProxy` queues received messages in a `MessageQueue`. They will be later scheduled to be applied on the wrapped `Object` by an internal scheduler, an instance of `MessageScheduler`, as shown in Figure 6.

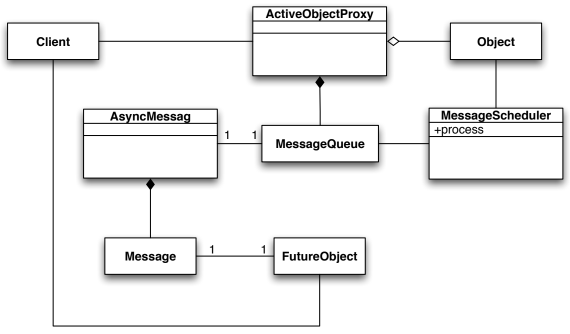


Fig. 6. Actors Implementation in ProtoTalk

Another difference between Actalk and ProtoTalk is the required explicit calling of the `respond:` methods in Actalk for getting the result of an asynchronous method invocation. In contrast, ProtoTalk utilizes the notion of *future objects* [8]. Therefore, it does not require such a method, and the sender instantly receives a `FutureObject` from the corresponding `ActiveObjectProxy`. The `FutureObject` sent to the client has a `valid` method which indicates whether the `FutureObject` is still being evaluated or is ready for use. The sender can keep on executing in its own context until it reaches an expression that requires the value of the `FutureObject`. On that point, if the future's evaluation is not complete, it suspends. The following code demonstrates how these active objects are created and used in ProtoTalk:

```
|actor passiveObject future|
passiveObject _ SamplePassiveObject new.
"Wrap the passive object by active object proxy"
actor _ ActiveObjectProxy new object: passiveObject.
"Send an asynchronous method"
future _ actor aMessage.
"do other stuff"
future value. "wait for the evaluation"
```

2.5. Protocol Data Unit Specification

Protocol Data Unit (PDU) specification is an important, yet cumbersome aspect of protocol design. ProtoTalk adopts a generative approach to assuage this task. A PDU aggregates other pre-constructed PDUs as its fields. As such, it complies with the *composite* design pattern [6]. The following piece of code exemplifies the usage of `PDUPrototype`, ProtoTalk's entry point for defining PDUs.

```
|pdu aPDU|
pdu _ PDUPrototype new: #PDUType.
pdu  addField: ThirtyTwoBits
        named: #myFiel setTo: 16rAAAAAAAA.
pdu  addField: Bit named: #anotherName
       after: #fieldName setTo: 1.
pdu  addField: Bit named: #yetAnotherName
       before: #anotherName.
```

```
aPDU _ pdu new.
aPDU fieldName. "Print It>>" 16rAAAAAAAA
aPDU howManyBits. "Print It>>" 34
aPDU yetAnotherName: 0. "Setting a field"
aPDU asBits.
 "Print It>>" 1010101010101010101010101010101010
aPDU field: #fildName. "Print It>>" aFieldName
aPDU fieldAt: 1 asBits. "Print It>>" 1
```

When a `new:` message, piggy-backing a symbol (e.g. `#PDUType`), is sent to the `PDUPrototype` class, this class creates a new subclass of itself named after the argument of the `new:` method. It thereafter instantiates the new class and returns it back. The subsequent messages sent to the instance belong to a specific category of messages called `prototyping`. They are responsible for reconfiguration and refinement of their *classes*. More specifically, prototype *objects* in ProtoTalk are able to modify the code of their own *class* by adding new methods and changing their underlying code. This is a common theme in other generative parts of ProtoTalk, such as state machine or message sequence descriptions. More detail on the mechanics of self-configuring objects in ProtoTalk is discussed in Section 3.

The `prototyping` methods of `PDUPrototype` objects generate Smalltalk code for various functionalities, including: integrity and size checking, and marshaling/de-marshaling of fields into actual packets, examples of both are listed in the above code. In ProtoTalk, constraining fields' values, e.g. constant-value fields, or data fields values of which solely depend on those of other fields, are possible. An important application of constant-value fields is in defining packet preambles while the definition of checksum fields is an example for which the dependent-value kind of fields is useful. Furthermore, it is possible to add more syntactic sugar to PDU definition in ProtoTalk. The following listing achieves essentially the same functionality of `addField` method, by using the + and < operators defined in `PDUPrototype`.

```
pdu is: #Method + ' ' + ' ' + #RequestURI
        + 'VIA' + #Address.
```

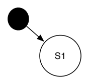
2.6. State Machine description

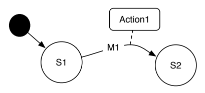
Many of the components described as the subsystems of *Protocol Entities* are realized using finite state machines. Components such as event handlers, session managers, services and protocol dispatchers, all embed a state machine inside. State machine definitions in ProtoTalk are designed to be expressive, extendible and concurrent. Basic classes involved in state machine descriptions are `FSMPrototype`, `State` and `Action`. Instances of the latter class are wrappers of Smalltalk's block closures. State transitions are realized using messages. Table 1 illustrates a typical step-by-step definition of a state machine in ProtoTalk. In each step, the `at:on:switchesTo:` message adds a new class (if it does not exist) that is provided as the argument of `switchesTo:`. It also inserts a new method with the name of the argument of `on:` and generates the code under the new method for switching to the newly created state.

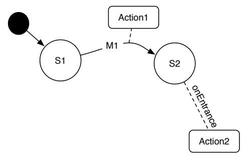
TABLE 1. Incremental specification of a FSM

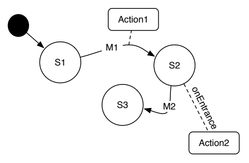
ProtoTalk Code	Generated State Machine

`|fsm|`
`fsm _ FSMPrototype new`
`      name:#SrvFSM.`	

`fsm StartsAt:#S1.`	

`fsm at:#S1 on:#M1`
`    switchesTo:#S2`
`    does:#Action1.`
	

`fsm onEntranceTo:#S2`
`    does:#Action2.`
	

`fsm at:#S2 on:#M2`
`    switchesTo:#S3.`
	 Action1
	M1
	S1
	S2
	S3
	M2
	onEntrance
	Action2

2.7. Message Sequence Definition

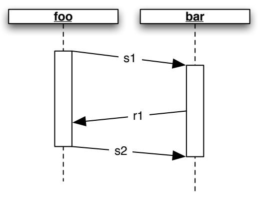


Fig. 7. A Sample Message Sequence Chart

Message sequence charts are widely used in protocol specifications for denoting specific scenarios between communication parties. ProtoTalk provides syntactic support to define message sequences in an incremental and declarative fashion. The entries are in the form of predicates that, similarly to other constructs in ProtoTalk, are sent to prototype classes. These classes refine themselves, and generate other classes as needed, when they receive such messages. Figure 7 depicts a sample interaction scenario, ProtoTalk denotation of which comes in the following.

```
foo _ ProtocolEntityPrototype new state:#Idle.
bar _ ProtocolEntityPrototype new state:#BeforeS1.
foo   sendsMessage: #s1 to: bar.
bar sendsMessage: #r1 to: foo.
foo sendsMessage: #s2 to: bar.
```

3. ProtoTalk Implementation

3.1. ProtoTalk Concurrency

Concurrency plays a key role in ProtoTalk. It adds actor's functionality on top of Squeak's concurrent programming facilities. `ActiveObjectProxy` is the wrapper for passive objects' behaviors. Upon instantiation, this object creates a `Process` that runs an instance of `MessageScheduler` inside. At runtime, instances of `ActiveObjectProxy` trap the received messages by the means of `doseNotUnderstand:` method, and form an `AsyncMessage` object for each message. This message is queued at the end of a `SharedQueue` shared between the sender's context and that of the scheduler. The client is immediately responded to by a `FutureObject` which is subsequently updated by the results of the requested evaluation, once it is complete.

Smalltalk's processes (and similarly those of Squeak) are switched cooperatively inside one priority level, thus a higher priority process will always preempt the lower priority ones [11]. Even though this might be an apt scheme for multimedia applications or graphical user interfaces that for which Smalltalk was originally designed, having a time-sharing system for implementation of actors that need to be used seamlessly, is almost mandatory. In order to keep this scheme consistent with the rest of the system, we developed a time-sharing scheduler that only handles ProtoTalk's active objects. This *ad hoc* scheduler runs at the highest priority level while scheduling actors' processes at a lower priority. The so called scheduler, `ProtoTalkScheduler`, uses round robin algorithm as its current scheduling policy.

3.2. Generative parts of ProtoTalk

ProtoTalk is a generative framework. As such, it extensively harnesses Smalltalk's reflective facilities in order to generate code incrementally. In this section we review the techniques used in the implementation of generative packages of ProtoTalk. As described in Section 2, many of the ProtoTalk features are based on generation of new classes at runtime. Moreover, the customization of these generated classes does not only happen at compile-time; it, nonetheless, is an incremental process that is often carried out at runtime. This is enabled by the highly dynamic and flexible meta-facilities that Smalltalk is famous for [3], [4], [2].

`ClassBuilder` is frequently used in ProtoTalk for crafting new classes. Once these generated classes are created, ProtoTalk uses Squeak's intercession abilities to incrementally populate them with methods and fields. The following piece of code shows an example of how these facilities are exploited in ProtoTalk. This pattern is at the core of most generative features of ProtoTalk, such as layer, PDU, FSM and sequence chart definitions. An object whose class is altered, uses Smalltalk's `become:` primitive to adapt itself to the updated class [4].

```
 (ClassBuilder new) superclass: PDUPrototype
  subclass: pduName instanceVaraibleNames: ''
  classVariableNames: '' poolDictionary: ''
  category: 'ProtoTalk-PDU'.
pduName addInstVarName: fieldName.
pduName compile: ('init' + (fieldName) asString +
  '    ' +  fieldName asString + '_' +
  fieldType asString + 'new')
  classified: 'initialization'.
^ Smalltalk classNamed: (pduName asString)
```

Although adding new methods to the classes is sufficient for most cases, there are still situations in which the source code of an existing method needs to be modified. This can be accomplished using Smalltalk's ability to reify the source code associated to a method as a piece of text. That text can be parsed using Smalltalk's built-in parser and can be modified either at the source or the abstract syntax tree level. The following snippet shows how ProtoTalk uses this feature:

```
pduInitSrc _  self class sourceMethodAt: #initialize.
self class compile: (pduInitSrc + '  ' +
 'self' + ' init' + (fieldName asString))
             classified: 'initialization'.
```

4. Application Development in ProtoTalk

To evaluate efficacy and expressiveness of ProtoTalk, we used it to developed a partial prototype of the Session Initiation Protocol (SIP)'s proxies, as specified in RFC 3261 [17]. In comparison to our previous experience in design and development of network protocols, we found several advantages in using ProtoTalk. As expected, data units and layers were noticeably easier to define in ProtoTalk than in such statically-typed languages as C++. This, as mentioned, is mostly the result of encapsulation of domain knowledge in the form or reusable code generators in ProtoTalk. The layer adaptation facilities in ProtoTalk supports reconfiguration of protocols for investigating different deployment scenarios. For example, with just a few edits, one can configure the SIP prototype to operate on top of an application layer protocol such as HTTP, instead of its conventional underlying layer, TCP.

Session Initiation Protocol is a standard regulated by Internet Engineering Task Force (IETF) specified in RFC 3261. The main intent behind the design of SIP is to provide a standard protocol for creation, modification, management and termination of sessions amongst participants operating in networks such as the Internet. The established sessions can carry various communication payloads including voice and internet telephony applications, multimedia communication and network games. SIP relies on entities it refers to as *proxies*, which act as mediators between SIP communication ends. A SIP client usually contacts a Proxy in order to initiate a session with another client. Proxies can operate in stateless mode, which does not require maintaining any information about sessions and clients. The typical interaction between clients and two stateless proxies is depicted in Figure 8.

The following ProtoTalk specification generates stub code for the sequence diagram of Figure 8. In addition to the classes and messages, ProtoTalk can generate code for the state machines implicated by these message sequence

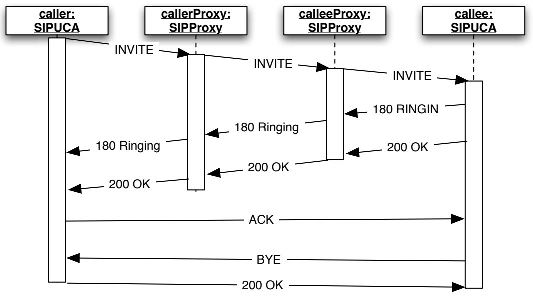


Fig. 8. SIP Proxy operations in stateless mode

specifications. More specifically, ProtoTalk inserts properly-aligned wait-states , pertaining to the messages exchanged in each step of the communication, into the generated state machines' code. For example, `caller` goes into the `waitForRinging` state in the anticipation of a `Ringing` signal from `callerProxy`, after it sends an `Invite` message to `callerProxy`.

```
caller _ UCA new state:#OffHookState.
callee _ UCA new state:#OnHookState.
callerProxy_ StateLessProxy new state:#Forwarding.
calleeProxy_ StateLessProxy new state:#Forwarding.
caller       sendsMessage:#Invite  to:#callerProxy.
callerProxy  sendsMessage:#Invite  to:#calleeProxy.
calleeProxy  sendsMessage:#Invite  to:#callee.
calleeProxy  sendsMessage:#Trying  to:#callerProxy.
callerProxy  sendsMessage:#Trying  to:#caller.
callee       sendsMessage:#Ringing to:#calleeProxy.
calleeProxy  sendsMessage:#Ringing to:#callerProxy.
callerProxy  sendsMessage:#Ringing to:#caller.
callee       waitsFor:#OffHookMsg.
callee       sendsMessage:#OK  to:#calleeProxy.
calleeProxy  sendsMessage:#OK  to:#callerProxy.
callerProxy  sendsMessage:#OK  to:#caller.
caller       sendsMessage:#ACK to:#callee.
```

The SIP RFC comprises a myriad of message types. ProtoTalk's PDU definition features can markedly comfort the effort involved in prototyping of this protocol by facilitating denotation of SIP data units. A portion of the PDU definitions for SIP is demonstrated in the following listing:

```
sipPDU _ PDUPrototype new name: #SIPPDU.
sipPDU is: #Request or: #Response.
Request is: #RequestLine + #MessageHeaders +
 (String crlf) + #MessageBody.
RequestLine is: #Method + ' ' + #RequestURI + ' ' +
 #SIPVersion + (String crlf).
Method is: 'INVITE' or: 16r41434B "Hex for ACK"
 or: 'OPTIONS' or: 'CANCEL' or: 'REGISTER' or: 'BYE'.
MessageHeaders hasMany: #MessageHeader.
MessageHeader is: #Accept or: #AcceptEncoding
 or: #AcceptLanguage or: #AlertInfo or: #Allow
 or: #AuthenticationInfo "..."
```

The generated classes are instantiated in the following code which demonstrates an example interaction with the defined SIP messages in Squeak's Workspace:

```
aSipPDU _ SIPPDU new: #Request.
aSipPDU requestLine method: (Method new: #INVITE).
aSipPDU requestLine requestURI: 'sip:bob@biloxi.com'.
aSipPDU messageHeaders add:
 (MessageHeader new: #ACCEPT). "..."

aSipPDU asString.
"Print It>>" 'INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP bigbox2.site3.atlanta.com;
branch=z9hGbK77ef4c2312983.1
```

5. Related Works

Object oriented and component oriented techniques have been previously acknowledged as effective methodologies for specification, design and implementation of telecommunication protocols [18], [19], [20], [21]. An architecture framework modeling of telecommunication software is presented in [22]. There has been several attempts in raising the level of abstraction in different phases of protocol development, ranging from the specification to the verification phase [23].

Specification Description Language (SDL) [24], designed and proposed by International Telecommunication Union (ITU), is a de facto standard in protocol specification. SDL is employed by many telecommunication standardization bodies and enjoys an extensive degree of tool support from various vendors. Promela++ [25] is a superset of C which provides relatively more abstract constructs for denoting network protocols. Morpheus [26] is a concurrent domain-specific programming language that mostly focuses on the architectural description of protocols. Abdullah in [27] presents a method for expressing protocol state machines using a format based on XML. Using XSLT, appropriate C code can be generated from an XML specification of a state machine. Barbeau and Bordeleau present a generative programming approach based on feature modeling to develop C++ implementation of protocol stacks from feature models [28].

6. Future Work and Conclusion

In this paper, we introduced ProtoTalk, a generative framework on top of Squeak Smalltalk, designed to facilitate iterative and incremental prototyping of network and telecommunication protocols. More specifically, ProtoTalk provides features for the denotation of reconfigurable, layered protocol stacks, and thereafter populating them with protocol entities. In addition, ProtoTalk provides facilities for the specification of state machines, message sequences and protocol data units. Finally, it supports transparent concurrency at the object level of granularity using the notion of active objects. The dynamic, incremental and interactive nature of ProtoTalk first allows for rapid prototyping of protocols from standards and specifications, and second provides a test bed for experimenting with many design alternatives that should be eventually resolved in the final realization of the protocol.

As a proof of concept, we have used ProtoTalk for partially prototyping Session Initiation Protocol's (SIP) proxy units as specified in RFC 3261. As future work, we envision utilizing VMMaker, a customizable Squeak-based toolkit that allows for translating Smalltalk programs into functionally equivalent C code. This can be an appealing feature for ProtoTalk, since it combines the performance of the C language and the expressiveness and productivity of Smalltalk for protocol design. We believe that in addition to the discussed benefits of ProtoTalk for incremental prototyping of protocols, it also sheds light on the potential utility of reflective techniques, as well as the aptitude of the Smalltalk language, for hosting domain specific frameworks and languages.

References

[1] "Squeak official website," http://www.squeak.org.

[2] C. S. Ali Razavi, "Smalltalk vs. Ruby: Reflective facilities in contrast," http://swen.uwaterloo.ca/ arazavi/mplsa.pdf, 2004.

[3] F. Rivard, "Smalltalk: a reflective language," in *REFLECTION'96*, April 1996, pp. 21–38.

[4] B. Foote and R. E. Johnson, "Reflective facilities in Smalltalk-80," in *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM, 1989, pp. 327–335.

[5] D. Garlan and M. Shaw, "An introduction to software architecture," Pittsburgh, PA, USA, Tech. Rep., 1994.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[7] A. S. Tanenbaum, *Computer networks: 2nd edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[8] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[9] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming," *ACM Comput. Surv.*, vol. 30, no. 3, pp. 291–329, 1998.

[10] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[11] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. [Online]. Available: http://portal.acm.org/citation.cfm?id=273

[12] R. Steigerwald and M. Nelson, "Concurrent programming in Smalltalk-80," *SIGPLAN Not.*, vol. 25, no. 8, pp. 27–36, 1990.

[13] Y. Gao and C. K. Yuen, "A survey of implementations of concurrent, parallel and distributed Smalltalk," *SIGPLAN Not.*, vol. 28, no. 9, pp. 29–35, 1993.

[14] W. J. Dally and A. A. Chien, "Object-oriented concurrent programming in CST," in *Proceedings of the third conference on Hypercube concurrent computers and applications*. New York, NY, USA: ACM, 1988, pp. 434–439.

[15] Y. Yokote and M. Tokoro, "The design and implementation of concurrent Smalltalk," in *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM, 1986, pp. 331–340.

[16] J.-P. Briot, "Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 enviroment." Nottingham, UK: Cambridge University Press, 1989, pp. 109–129.

[17] IETF, "Request For Comment 3261: Sip: Session Initiation Protocol," http://www.ietf.org/rfc/rfc3261.txt, June 2002.

[18] A. Beugnard, "Communication services as components for telecommunication applications," in *Proc. Objects and Patterns in Telecom Workshop (in ECOOP00)*, Sophia Antipolis, France, June 2000.

[19] G. Koutsoukos, J. Gouveia, L. F. Andrade, and J. L. Fiadeiro, "Managing evolution in telecommunication systems," in *Proceedings of the IFIP TC6 / WG6.1 Third Int. Working Conf. on New Developments in Distributed App. and Interoperable Systems*. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., 2001, pp. 133–140.

[20] S. Böcking, "Object-oriented network protocols," in *Proceedings of the INFOCOM '97*. Washington, DC, USA: IEEE Computer Society, 1997, p. 1245.

[21] A. A. Hanish and T. S. Dillon, "Communication protocol design to facilitate re-use based on the object-oriented paradigm," *Mob. Netw. Appl.*, vol. 2, no. 3, pp. 285–301, 1997.

[22] G. Fregonese, A. Zorer, and G. Cortese, "Architectural framework modeling in telecommunication domain," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. New York, NY, USA: ACM, 1999, pp. 526–534.

[23] R. Lai and A. Jirachiefpattana, *Communication Protocol Specification and Verification*. Norwell, MA, USA: Kluwer Academic Publishers, 1998.

[24] H. D. Ellsberger, J. and A. Sarma, *SDL–Formal Object-Oriented Language for Communication Systems*.

[25] A. Basu, G. Morrisett, and T. Von Eicken, "Promela++: a language for constructing correct and efficient protocols," *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 455–462 vol.2, Mar-2 Apr 1998.

[26] M. B. Abbott and L. L. Peterson, "A language-based approach to protocol implementation," *IEEE/ACM Trans. Netw.*, vol. 1, no. 1, pp. 4–19, 1993.

[27] I. S. Abdullah and D. A. Menasc, "Protocol specification and automatic implementation using XML and CBSE," in *Proc. of the Int. Conf. on Comm., Internet and Information Tech.*, 2003.

[28] M. Barbeau and F. Bordeleau, "A protocol stack development tool using generative programming," in *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*. London, UK: Springer-Verlag, 2002, pp. 93–109.