in: D. Batory and C. Consel (Eds.), Proceedings of Generative Programming and Component Engineering (GPCE), Lecture Notes in Computer Science 2487, Springer. 2002.

A Protocol Stack Development Tool Using Generative Programming

Michel Barbeau and Francis Bordeleau

School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa (Ontario), Canada K1S 5B6,
barbeau | francis @scs.carleton.ca,
WWW home page: http://www.scs.carleton.ca

Abstract. Traditional protocol implementation approaches capture the structural aspects of protocols in a common base that can be used across layers. However, they are usually not very good at capturing the behavioral aspects. Two important implementation problems result, namely, reprogramming similar behavior and configuration of crosscutting concerns. In this paper, we present an approach to solve the problems of reprogramming similar behavior and absence of systematic configuration mechanisims for crosscutting concerns in communication systems. Our approach is based on generative programming, has been implemented in C++ and has been validated with several protocols. We also sketch an approach for run-time reconfigurable protocol stacks.

1 Introduction

Accross layers, protocols have several architectural elements in common. Indeed, they are often based on a common model, which is customized for each protocol. Traditional protocol implementation approaches can capture the structural aspects of protocols in a common base that can be used across the layers. However, they are usually not very good at capturing the behavioural aspects. As a result, the same behaviour model has to be reprogrammed and debugged from protocol to protocol. We call this problem *reprogramming similar behaviour*.

Each individual protocol implementation framework includes a set of good solutions to concerns that cross all the protocols of an implementation, e.g. message representation and event handling. However, it is actually very hard to combine a good solution for one concern from one framework with a good solution for another concern from another framework. It is unfortunate because it might be exactly the combination that would make the application at hand effective. We call this problem *absence of systematic configuration mechanisms for crosscutting concerns*.

In this paper, we present an approach to solve the problems of reprogramming similar behaviour and configuration of crosscutting concerns in communication systems. Our approach is based on generative programming and has been implemented in C++. It has been validated with several protocols, including a packet radio protocol, a wireless LAN protocol, IP, a dynamic source routing protocol,

a satellite transport protocol and a simple network management protocol. Some of our work is available on-line [1].

Traditional approaches for the implementation of communication systems and candidate approaches are reviewed in Section 2. Our approach based on generative programming is presented in Sections 3 and 4. In Section 5, we discuss a limitation of our current work and how we plan to address it in the future. We conclude with Section 6.

2 Related Work

Protocol implementation approaches can be categorized according to the placement of the code relative to the OS space. There are three approaches: kernel level, user level and a mix of both.

With the kernel-level approach, all the code is placed in the kernel space. This approach is especially efficient for network-level and link-level protocols in which the amount of data and control traffic is large. A lot of work is normally required to port a kernel-level protocol stack from one OS to another. The TCP/IP stack of both Linux [2] and BSD Unix [14] are kernel level.

With the user-level approach, the kernel of the OS is not changed. The protocol stack runs as one or several user processes. On Unix, for example, user-level protocols communicate with the OS through the socket abstraction. The deployment and debugging of communication systems are made easier. The start and interruption of the communication systems are also often easier to manage. It is also easy to port an implementation from one OS to another. This approach may cause, however, performance penalties because protocol data units have to be transferred between the user-level and kernel-level and may not work for cases in which a certain level of control of the OS is required, but not available through system calls. Hence, a mixed approach is often required. The x-kernel framework [6] and NOS TCP/IP stack [13] are both user level. Elements of mobility support in IPv6 have been implemented with a mix approach in [15].

All the aforementioned implementation approaches are based on traditional programming methodologies, except for x-kernel, which is an object based framework. Object-oriented approaches can help to solve the problems of reprogramming similar behaviour and configuration of crosscutting concerns. Use of the object-oriented approach to implement network management applications has resulted to SNMP++ [8]. A pattern language that addresses issues associated with concurrency and networking is presented in [12]. An object-oriented network programming approach is described in [11] and [10].

Other related work is the Ensemble project. It proposes an approach in which stacks of micro-protocols can be assembled to fulfill needs for group communications [3]. There is a support for dynamic adaptation [9].

In this paper, we introduce a novel protocol implementation approach based on generative programming, which can be combined with other approaches to obtain a higher degree of reuse of design and code. To the best of our knowledge,

it is the first attempt to use the generative programming approach, including feature modeling, for developing protocols.

3 Feature Modeling of Protocols

Generative programming [4] is domain engineering and modeling of a family of software as components that can be assembled automatically. Domain engineering consists of three steps: domain analysis (including feature analysis), domain design and domain implementation. A model is presented consisting of software components that can be assembled to build communication systems, i.e. protocol stacks.

Domain Analysis

According to [6], a family of telecommunications protocol stacks can be modeled using the following abstractions: Session, Protocol, Participant, Map, Message, Event and Uniform Interface.

A Session is the abstraction of an end-point of a communication channel, i.e. a point-to-point channel is terminated at each end by a session. A Protocol is an abstraction that represents specific message formats and communication rules. There is a many-to-one relationship between Session and Protocol; each session is associated with a protocol and each protocol can contain several sessions. For instance, the local end-point of a host-to-host channel is abstracted as an IP session and is embedded and managed by an IP protocol. A session object is dynamically created when a new channel is opened. Protocols and sessions contain together the elements required for the interpretation of messages: coding, decoding and taking decisions. They maintain the local state of the channel: window size, number of next message expected, number of last acknowledged message, etc.

The Participant abstraction represents the identity of an application, a protocol or a session. For example, in the TCP/IP architecture the identity of an application is a port number and an IP address.

The Map abstraction models partial functions. It is used to map external keys made of addresses (e.g. a source IP address and a remote IP address pair) to an internal value (e.g. the session corresponding to the abstraction of that channel).

The general model for a protocol layer is pictured in Figure 1. Each layer is made of a protocol. It is a rather static entity (created at start and persisting until the end). A protocol contains both an active map and a passive map. The active map keeps track of sessions managed by the protocol. Each entry maps a local address and remote address pair (e.g. A1 and A2), that we call an ActiveId, to a pointer to a session which models the entry of a communication channel between a local entity that has address A1 to a remote entity that has address A2. The passive map keeps track of enabled local addresses on which the protocol is eager to open new channels. With TCP, for instance, the passive map

would contain the port numbers on which servers are listening for client-initiated sessions. The establishment of the channel is initiated by a remote entity. Such an enabled address is called a PassiveId. The enabling is normally performed by a higher-level protocol. The number of times this action has been performed is kept in the Ref count field and the Protocol field is a pointer to this higher-level protocol.

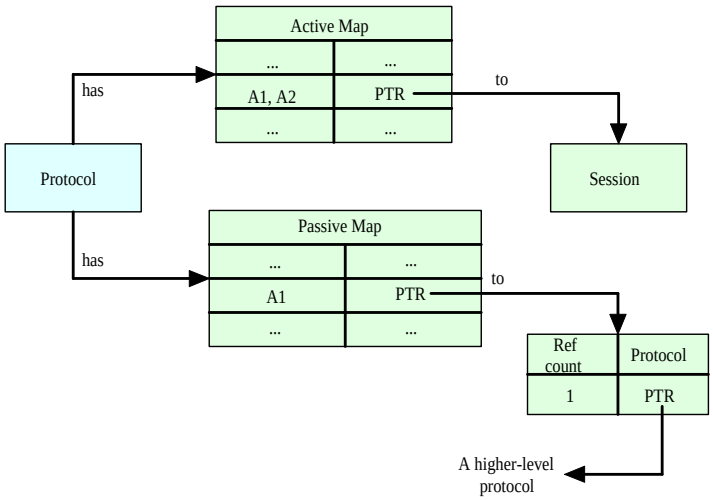

Fig. 1. General model of a protocol layer.

The Message abstraction represents protocol data units. It is a string of bytes of some length. This abstraction is used to pass data up and down in a protocol stack. It provides operations for adding/stripping headers and fragmentation/reassembly of messages.

Scheduling of events, captured by the Event abstraction, is an activity shared by several protocols, e.g. transmission of advertisements and scheduling of retransmissions.

The notion of Uniform Interface captures the operations common to all protocols and sessions. Interaction with protocols and session can be modelled in a layer independent model using a common collection of operations. For example, every layer provides an operation for opening a channel (xOpen) and operation for pushing data through data channel (xPush). It also makes protocols and session definitions easily substitutable.

Feature Analysis

Feature analysis is the identification of the commonalities and variability among the members of a family software. A feature diagram is a result of feature anal-

ysis. A feature is a property having importance and is a point where choices can be made to define a particular instance of a concept. A feature diagram is a representation that captures the features of a concept. Figures 2 and 3 picture the feature diagrams for the Session and Protocol concepts. The root of each diagram represents a concept and leaf nodes represent its features.

The Session concept has the Message mandatory open feature (the filled circle indicates a mandatory property, open brackets indicate an open property) and the EventManager optional feature (indicated by a non-filled circle). For example, either a single buffer model, as in Linux, or directed acyclic graph of buffers, as in x-kernel, can be chosen for the Message feature. For the EventManager feature, either a delta list model of a timing wheel model of event manager can be selected.

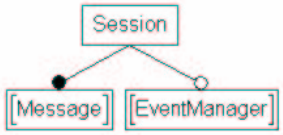

Fig. 2. The Session concept with internal features.

The Protocol concept has the Message, Session, Participant, EventManager, ActiveOpen and PassiveOpen features. The Session feature is optional because the Protocol-Session model does not fit certain cases such as high-level protocols or low-level protocols that abstract hardware devices. A possible choice for the Participant feature is a stack model of identifiers, as in x-kernel, or a linked-list of identifiers. A Protocol variant is defined using a Session variant, when applicable. The choice for the Message feature of the Protocol variant must match the value of the same feature in the Session variant.

The Active/Passive Map (with the accompanying ActiveId/PassiveId) is modelled as a sub feature of feature Active/PassiveOpen. The ActiveOpen feature is optional because as we mentioned before certain protocols are not providers of communication channels and don't fit into that model. The PassiveOpen feature is optional because enabling addresses is needed only for nodes running servers, which may not be required, for instance, by handheld computers. If the PassiveOpen feature is selected, then the ActiveOpen feature must be selected as well.

Domain Design

Domain design is accomplished using a conceptual tool called the GenVoca grammar [4]. A GenVoca grammar defines the architecture of a domain as a hierarchy of layers of abstraction. Each layer groups together features or concepts, which

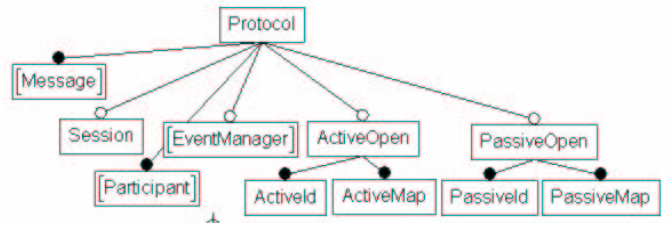

Fig. 3. The Protocol concept with internal features.

are independent of each other. Each layer is defined as a refinement of the layers below. This hierarchy reflects the dependencies between the features and concepts, which are present in the feature analysis. In the hierarchy, we go from the more specialized (top) to the more general (bottom). Features or concepts that appear in each layer are called components and are mapped to classes, in a C++ implementation. A GenVoca grammar for protocol stacks is listed hereafter. There are seven named layers of abstraction L1 (top) to L7 (bottom):

```
Protocol: (L1)
   PassiveOpen[POWithOptAO] | ActiveOpen[Config] | ProtocolUI
POWithOptAO: (L2)
   ActiveOpen[Config]
Config: (L3)
   ProtocolUI EventManager ActiveId ActiveMap PassiveId PassiveMap
ProtocolUI: (L4)
   Protocol[ProtocolUIConfig]
ProtocolUIConfig: (L5)
   Message Session Participant
SessionUI: (L6)
   Session[SessionConfig]
SessionConfig: (L7)
   Message EventManager
```

Mapping of the feature diagrams to the GenVoca grammar is done according to [4]. The Message and EventManager features of the Session concept, are collected together in the bottom layer (L7), also called a configuration repository. This configuration repository is used to define the SessionUI layer (L6), which reflects the Session concept.

For the Protocol concept, we make a distinction between the essential features that are required to interact with a protocol and additional features that are needed to define the internal of at least one protocol. We came to this distinction after several iterations. It separates the protocol declaration (an interface) from the protocol definition (an implementation). To interact with a protocol, only the Message, Session and Participant features are required. They are collected together in a repository (L5). It is used to define the layer above, named ProtocolUI (L4), which is the abstraction of an interface to a protocol.

The layer labeled Config (L3) is the configuration repository for a protocol definition and includes a dependency on the ProtocolUI layer and all additional internal features. The ActiveOpen feature is mapped to the POWithOptAO

layer (L2), which depends on the Config layer. The PassiveOpen feature depends on the ActiveOpen feature. This is reflected by a dependency from the L1 layer to the POWithOptAO layer. The Protocol layer (L1) groups three different alternatives that can be taken to define a protocol, namely, PassiveOpen parameterized with POWithOptAO (provides a basis for protocols that have sessions with client/server behavior), ActiveOpen parameterized with Config (provides a basis for protocols with sessions with client behavior only) or ProtocolUI (for protocols that don't embed sessions, e.g. top level protocols and certain drivers). Alternatives are separated by vertical bars.

4 Implementation

This section is about the implementation of the components uncovered in Section 3. This application is rather complex. In this paper it is possible to address a limited number of key elements. The full source code is available on-line [1].

Components with parameters are represented as C++ class templates. In other words, the GenVoca grammar is a generator of hierarchies of classes. With the GenVoca grammar translated to C++, the assembly of a base class for a new protocol class is as follows, informally:

```
PassiveOpen<ActiveOpen<Protocol<Config> > >
```

It results from the composition of a series of components. The Protocol component is configured with the Config repository, which is a structure (defined with the `struct` statement) with type members (defined with the `typedef` statement). The result is the actual parameter of an ActiveOpen component which is in turn the actual parameter of a PassiveOpen component.

In our model, such a generated type definition is used as a base for the definition of a specific protocol. In this base, all the architecture related code is present. What remain to be defined are specific message interpretation rules of a specific protocol. Here is an outline of the class template representing the protocol component.

```
template <class Generator> class Protocol {
public:
   typedef typename Generator::Config Config;
   // retrieve all the needed types from "Config"
   typedef typename Config::Session Session;
   typedef typename Config::Message Message;
   typedef typename Config::Participant Participant;
   // Constuctor
   Protocol() {}
   // Destructor
   virtual ~Protocol() {}
   // Does an active open of a session
   virtual Session *xOpen(Protocol *hlp, Participant **p)
   { return NULL; }
   // Does close an active session
   virtual Outcome xClose(Session *s)
   { return OK; }
   // Other operations of the uniform interface ...
   // Layer interconnection operations
   Protocol * getmyHlp() { return myHlp; };
   void setmyLlp(Protocol * aLlp) { myLlp = aLlp; };
   Protocol * getmyLlp() { return myLlp; };
private:
```

```
   // My high-level protocol
   Protocol  * myHlp;
   // My low-level protocol
   Protocol  * myLlp; };
```

The Protocol class template has a parameter named `Generator` with a field named `Config`. It is also called a repository and its role is to pass to the template the choices needed to obtain a specific variant. The Protocol class template declares the operations of the uniform interface. They are declared virtual because they can be overridden in derived classes. The field `Generator::Config` is exported under the data member named `Config`, which makes it accessible to other components down in an inheritance hierarchy.

The `xOpen()` operation does an active open of a session. It takes in input a reference to a high-level protocol opening the session (`hlp`), a pair of participant addresses (`p`), `p[0]` is local and `p[1]` is remote. In output, it returns a reference to a session. The `xClose()` operation closes an active session. It takes in input a reference to an active session (`s`). In returns OK/NOK to indicate success/failure to close the session. Declarations of other similar operations are omitted. Then there are operations for connecting protocol layers together, which are required to assemble a protocol graph.

A generator takes a specification of a uniform protocol interface (which is normally realized by all protocols of a protocol stack) and returns the finished interface. It is defined as follows:

```
template < class SessionClass, class ParticipantClass = Part,
   class MessageClass = SingleBuffMessage >
class PROTOCOL_UI_GENERATOR {
   // define a short name for the complete generator
   typedef PROTOCOL_UI_GENERATOR <
      SessionClass, ParticipantClass, MessageClass >
   Generator;
   // create configuration
   struct Config {
      typedef SessionClass Session;
      typedef ParticipantClass Participant;
      typedef MessageClass Message; };
public:
   // assembly
   typedef Protocol<Generator> RET; };
```

A generator is another class template that encapsulates the rules defining how components can be assembled together. It knows what goes with what and under which constraints. It takes a specification of what needs to be assembled. It verifies its consistency and can provide default values. The generator of the uniform interface (UI) of protocols takes a protocol description and returns a finished protocol UI base class. During this generation, the actual classes of the Message, Participant and Session features are selected. The result of the assembly is returned as the RET type member. A common interface to all protocols, within a stack, is generated as follows.

```
typedef PROTOCOL_UI_GENERATOR < SessionUI >::RET ProtocolUI;
```

It is the creation of an instance of the Protocol class template, which is renamed ProtocolUI. A value (SessionUI) for the feature Session is specified. Other features take default values. We now explain key elements of the implementation of a component in the hierarchy, namely, the ActiveOpen component which source is outlined hereafter:

```
template <class Generator>
class ActiveOpen: public Generator::Config::Base {
public:
   // export the name "Generator::Config" under the name "Config"
   typedef typename Generator::Config Config;
   // retrieve all the needed types from "Config"
   typedef typename Config::Base Base;
   typedef typename Config::Session Session;
   typedef typename Config::Message Message;
   typedef typename Config::Participant Participant;
   typedef typename Config::ActiveId ActiveId;
   typedef typename Config::ActiveMap ActiveMap;
   typedef typename Config::Header Header;
   typedef typename Config::EventManager EventManager;
   static const int hdrSize = sizeof(Header);
   typedef typename Config::ProtocolUI ProtocolUI;

   // active open of a session
   virtual Session *xOpen(ProtocolUI *hlp, Participant **p) {
      ActiveId key;
      Session * aSession;
      // execution of a protocol specific preamble (key construction!)
      if (xOpenPreamble(&key, p) != OK) {
         // execution of the preamble failed
         return NULL;
      }
      // is this session in the active map
      if ((aSession = activeMap.resolve((char *)&key)) != NULL) {
         // add a reference
         aSession->addRef();
         // session is in the active map, return it
         return aSession;
      }
      // session not in the active map, create it!
      aSession = createSession(&key);
      if (aSession == NULL) {
         // creation of session failed, return NULL
         return NULL;
      }
      // store the new binding in the active map
      if (activeMap.install((char *)&key, aSession) == NULL) {
         delete aSession; // recover the allocated memory
         return NULL;
      }
      // end with success
      return aSession;
   }
   // definition of other operations ...  }
```

The ActiveOpen class template has one parameter named `Generator`, which supplies as sub fields of the `Config` field the `Base` base class name and choices of features. Note that the format of headers of protocol data units is passed within the configuration as the `Header` field. The ActiveOpen class template is responsible of keeping track of sessions that are active. It creates and deletes entries in the active map. Each entry is a pair made of a key (structure specific to each protocol) and a reference to an active session.

The definition of the `xOpen()` operation is detailed. The `xOpen()` operation creates a session. Arguments are a high-level protocol `hlp`, a reference to the caller of the operation and a local participant and a remote participant contained in `p`. Firstly, an active id key is constructed by calling the `xOpenPreamble()` operation. The active map is consulted to determine if there is already a session associated to the key. If a session exists, then its reference count is incremented.

Else, a new session `s` is created by calling the `createSession()` operation. If creation of `s` succeeds, then a new entry is created in the active map binding the key and `s` together. This operation depends on the implementation of the `xOpenPreamble()` and `createSession()` operations, which are done in a derived class.

The generator of a protocol is given below:

```
template <
   class BaseClass, class ActiveIdClass, class ActiveMapClass,
   class EnableClass, class PassiveIdClass, class PassiveMapClass,
   class HeaderClass, class EventManagerClass, class ProtocolUIClass,
   ActiveopenFlag activeopenFlag = with_activeopen,
   PassiveopenFlag passiveopenFlag = with_passiveopen >
class PROTOCOL_GENERATOR {
   // define a short name for the complete generator
   typedef PROTOCOL_GENERATOR <
      BaseClass, ActiveIdClass, ActiveMapClass,
      EnableClass, PassiveIdClass, PassiveMapClass,
      HeaderClass, EventManagerClass, ProtocolUIClass,
      activeopenFlag, passiveopenFlag > Generator;
   // parse domain specific language
   enum {
      isActiveopen = activeopenFlag==with_activeopen,
      isPassiveopen = passiveopenFlag==with_passiveopen };
   // assemble feature ActiveOpen
   typedef IF<isActiveopen,
      ActiveOpen<Generator>,
      BaseClass
   >::RET ActiveopenBase;
   // assemble feature PassiveOpen
   typedef IF<isPassiveopen,
      PassiveOpen<ActiveopenBase>,
      ActiveopenBase
   >::RET ProtocolBase;
   // create configuration
   struct Config {
      typedef BaseClass Base;
      typedef typename Base::Session Session;
      typedef typename Base::Participant Participant;
      typedef ActiveIdClass ActiveId;
      typedef ActiveMapClass ActiveMap;
      typedef EnableClass Enable;
      typedef PassiveIdClass PassiveId;
      typedef PassiveMapClass PassiveMap;
      typedef HeaderClass Header;
      typedef EventManagerClass EventManager;
      typedef typename Base::Message Message;
      typedef ProtocolUIClass ProtocolUI; };
public:
   typedef ProtocolBase RET; };
```

The generator reflects the rules of the GenVoca grammar of Section 3. The `BaseClass` parameter is a base class and the assembly is defined as an extension of it. Logically, it is protocol UI. The `activeopenFlag`/`passiveopenFlag` selects the inclusion of the ActiveOpen/PassiveOpen feature. The result, returned as the RET type member, is a base class which can be used through an extension relationship to define a specific protocol.

Configuration of the WLAN Protocol

A user level Wireless LAN protocol is used an example. The WLAN protocol is a user-level abstraction of an 802.11 interface [7]. A WLAN address is defined

by the `WLANAddr` structure. An address is six bytes long, stored in the data data member.

```
#define WLAN_ADDR_LEN 6
struct WLANAddr {
   // address
   unsigned char data[WLAN_ADDR_LEN]; };
```

The configuration code is as follows:

```
// (1) format of a WLAN header
struct WLANHeader {
   WLANAddr destAddr;
   WLANAddr srcAddr;
   unsigned short type;
#define IP_TYPE x0800
};
#define WLAN_HEADER_LEN 14 // bytes

// (2) definition of a key in the active map
struct WLANActiveId {
   WLANAddr localAddr;
   WLANAddr remoteAddr; };

// (3) configuration of the Active Map
struct WLANActiveMapConfig {
   static const int KEYSIZE = sizeof(WLANActiveId);
   typedef SessionUI ValueType;
   static const int HASHSIZE = 50;
   typedef Map<WLANActiveMapConfig> WLANActiveMap; };

// (4) definition of a key in the passive map
struct WLANPassiveId {
   WLANAddr localAddr; };

// (5) define enable structure
struct WLANEnable {
   unsigned int refCnt;
   ProtocolUI *hlp; // ptr to the high-level protocol };

// (6) configuration of passive map
struct WLANPassiveMapConfig {
   static const int KEYSIZE = sizeof(WLANPassiveId);
   typedef WLANEnable ValueType;
   static const int HASHSIZE = 50;
   typedef Map<WLANPassiveMapConfig> WLANPassiveMap; };

// (7) generation of the base class used to derive the WLAN protocol
typedef PROTOCOL_GENERATOR <
   ProtocolUI,
   WLANActiveId,
   WLANActiveMapConfig::WLANActiveMap,
   WLANEnable,
   WLANPassiveId,
   WLANPassiveMapConfig::WLANPassiveMap,
   WLANHeader,
   SimpleEvent,
   ProtocolUI,
   with_activeopen,
   with_passiveopen >::RET WLANBase;
```

The WLANHeader structure defines the format of the header of frames of the WLAN protocol. It consists of a destination address, `destAddr`, a source address, `srcAddr` and a `type` field. The value of the `type` field determines the protocol to which the payload of the frame is associated.

The WLAN protocol has an active map and a passive map. The active map keeps track of sessions managed by the protocol. The passive map keeps track of local addresses with which the protocol is willing to create new sessions. The `WLANActiveId` structure defines the format of a key installed in the active map. It consists of a local address, `localAddr`, and a remote address, `remoteAddr`. Each key installed in the active map is paired with a pointer to an object of a class extending the SessionUI class, more specifically a class derived from SessionUI that defines a WLAN session. The active map is defined as an actualization of the Map template. The configuration passed to the template defines the size of a key, type of values pointed by a pointer paired with a key and hash size. The actualization is named `WLANActiveMap`.

The WLANPassiveId structure defines the format of a key installed in the passive map. It consists of a local address, `localAddr`. Each key installed in the passive map is paired with an instance of the `WLANEnable` structure. It consists of and integer counting the number of invocations to the open enable operation that have been performed using address `localAddr` and a pointer to the high-level protocol that was the first to invoke the open enable operation on this address, `hlp`. The passive map is defined as an actualization of the Map template. The configuration passed to the template defines the size of a key, type of values pointed by a pointer paired with a key and hash size. The actualization is named `WLANActiveMap`.

Finally, the WLANBase base class, used to derive the WLAN protocol, is produced using the `PROTOCOL_GENERATOR` generator. The actual arguments of the generator are the name of the parent class of WLANBase (ProtocolUI), the type of a key in the active map (`WLANActiveId`), the type of the active map (`WLANActiveMap`), the type of values pointed by entry stored in the passive map (`WLANEnable`), the type of a key in the passive map (`WLANPassiveId`), the type of the passive map (`WLAPassiveMap`), the type of the frame header (`WLANHeader`), the type of the chosen event manager (`SimpleEvent`), the class defining the uniform protocol interface (ProtocolUI), a value forcing the selection of the ActiveOpen feature (`with_activeopen`) and a value forcing the selection of the PassiveOpen feature (`with_passiveopen`). The ActiveOpen feature provides all the mechanisms required to maintain the active map while the PassiveOpen feature provides all the mechanisms required to maintain the passive map.

Class Diagram

The class diagram for the WLAN protocol is pictured in Figure 4. Each rectangle represents a class while each arrow represents an inheritance relation from a derived class to a parent class. The ProtocolUI class is an abstract class resulting from the actualization of the class template implementing the Protocol concept. It represents a uniform interface common to all protocols. The ActiveOpen and PassiveOpen classes represent the actualizations of the class templates implementing the ActiveOpen and PassiveOpen features. The WLANBase class is a synonym for the actualization of the PassiveOpen class template (equivalent to

typedef of the C language). The WLANProtocol class represents the implementation of the WLAN protocol and augments WLANBase with aspects that are specific to the WLAN protocol. WLANBase has pointers to instances of the WLANSession class (represented by the relationship ended by a diamond on the side of the container).

The SessionUI concept is an actualization of the class template implementing the Session concept. It represents a uniform interface common to all sessions. The WLANSession class represents the implementation of a session (i.e. a channel end-point) managed by the WLAN protocol. It implements the operations declared in the SessionUI class.

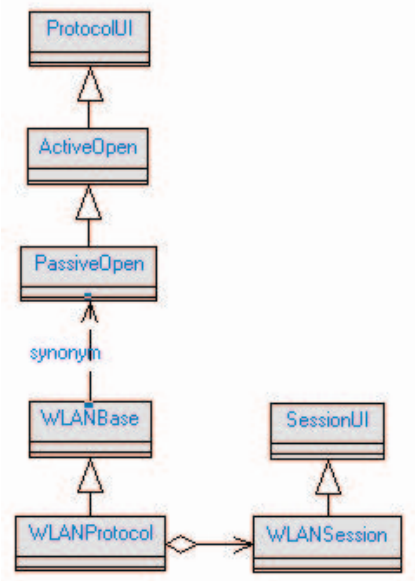

Fig. 4. Class diagram for the WLAN protocol.

5 Towards Dynamic Re-configurability

The main objective of generative programming is to develop families of systems. Until now, the concepts of generative programming have been almost exclusively used to generate systems at compile-time [4] by means of a generator. The role of the generator consisting in producing a system from a feature specification, described using a GenVoca grammar, input by a user. The generator is also responsible for validating feature specifications and detecting invalid feature combinations. Once the feature specification is validated, the generator produces the requested system by assembling predefined components.

While the current usage of generative techniques offers great potential in the context of software engineering by allowing for the development of families of systems, it remains limited with respect to other crucial issues like system run-time reconfiguration.

It is our opinion that the use of generative techniques should not be limited to static compile-time system configuration, but should also be used to develop systems that could be dynamically configured and reconfigured at run-time, which is considered to be a very important issue in todays software industry. A software system might have to be reconfigured for different reasons. For example, the user wants to change the set of features of the system, or because the current systems operation has degraded to the point where it no longer satisfies the required quality of service (QoS). The system might also have to change because the context in which it evolves has changed. To use the analogy of the car industry discussed in [4], what we often need to build in software engineering is a car that could reconfigure itself to move the driver seat from the left side of the car to the right side of the car when the car moves from France to England.

In our research project, we are currently working on the development of a reconfigurable protocol stack using generative techniques. The goals is to develop a system architecture that would allow the protocol stack to be initially configured and reconfigured at run-time without having to shutdown and restart the system. This would also allow for the online upgrade of the system.

Proposed Architecture

A high-level architecture of our reconfigurable protocol stack is given in Figure 5. The key element of this architecture is the Configuration Agent. The role of the Configuration Agent is in part similar to the role of a generator in generative programming. Initially, the Configuration Agent receives the specification of a desired set of features, described using a GenVoca grammar, it validates the set of features and then if the feature specification is valid, it configures the system by loading the requested components and linking them together in a communication topology that allows satisfying the user feature specification.

However, unlike a generator, the Configuration Agent also has the capability of dynamically reconfiguring the system at run-time. The triggering of the reconfiguration could be external to the system, e.g. an explicit command input by a user, or internal, e.g. as a result of decreasing performance internally observed by the system. The use of internally triggered reconfiguration requires the existence of some internal mechanisms that can determine when reconfiguration must occur. For example, a system might have a performance-monitoring component that would inform the Configuration Agent to reconfigure when the system performance reaches certain thresholds.

Upon reception of a reconfiguration request, the Configuration Agent must determine if the system is a proper state for reconfiguration. Two parameters must be considered at this point: the type of reconfiguration and state of the system. For this reason, the Configuration Agent needs to maintain, or have ways of obtaining, information concerning the state of the system. To continue with the

car analogy, it is important that the Configuration Agent of the car verifies the state of the car before initiating reconfiguration. If the reconfiguration request is about moving the driver seat from left to right, the Configuration Agent must first ensure that the car is stopped and that nobody sits in the front seats before initiating the reconfiguration. However, other types of reconfiguration, like switching from two-wheel-drive to four-wheel-drive, might not require stopping the car.

If the system is not in a proper state for reconfiguration, the Configuration Agent must either reject the reconfiguration request or bring the system to an acceptable safe state before initiating the reconfiguration.

During reconfiguration, the Configuration Agent is responsible for controlling and synchronizing the reconfiguration of the different components and for bringing the system back to its operational state once the reconfiguration is completed.

Protocol Layer Modifications

In order to develop a reconfigurable protocol stack, the different layers that compose the system would also have to be modified to allow for layer reconfiguration. The required layer modifications are twofold: layer interface and layer implementation. The layers of the protocol stack described in the previous sections have two main standard interfaces: the upper-layer interface and the lower-layer interface. The definition of a reconfiguable protocol stack requires the addition of a third standard interface to the different layers for the reconfiguration aspect. These interfaces are the ones used for the communication between the Configuration Agent and the different layers of the communication protocol stack. While the configuration messages defined in the configuration interfaces are standard for all layers, the behaviour executed upon reception of a configuration message is layer dependent.

The implementation of the different layers would also have to be modified to allow for reconfiguration. This can be done by recursively using the concept of a Configuration Agent inside each layer.

Tradeoffs

The main tradeoff between the conventional protocol stack presented in Section 3 and the reconfigurable one described in this section is the one of size (footprint) versus flexibility. The addition of the reconfigurability aspect to the system results in a more complex and larger system, but has the huge advantage of being more flexible.

In the next generation of our protocol stack, reconfigurability will be considered as a feature. Therefore, our generator will have the capability of i) generating a conventional protocol stack, as described in Section 3, for cases where the footprint is a main concern and ii) generating a reconfigurable protocol stack for cases where reconfigurability is a main concern.

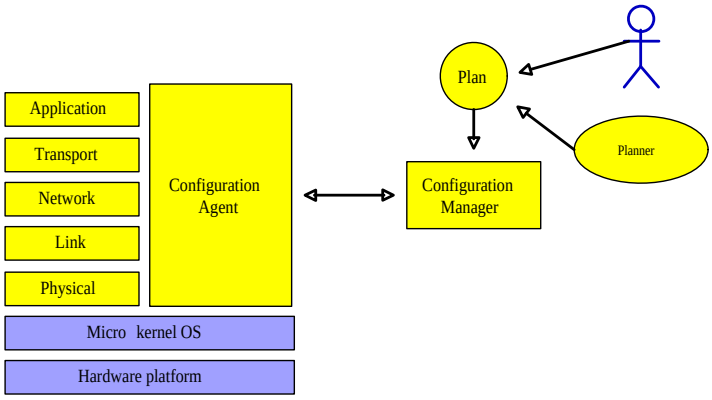

Fig. 5. Architecture of the reconfigurable protocol stack.

6 Conclusion

Our work consists of a unique combination of well established techniques for protocol implementation and generative programming. Our hypothesis is that by combining these techniques we provide a higher degree of configurability, but not at the expense of performance, correctness or resource usage. Our current implementation of the feature diagrams of Figures 2 and 3 offers 18 different alternatives (there are two alternatives for Message, three for EventManager and three for Protocol). We claim configurability superiority for crosscutting concerns, with respect to several protocol development environments. Regarding performance, correctness and resource usage, apart from the usage of our implementation to develop several protocols, systematic evaluation of the implementation of some of the crosscutting concerns has been conducted, namely, the Message and EventManager features [16]. The performance so far seems to be comparable with the performance obtained in other contexts. It is premature for us to claim superiority of performance, correctness and resource usage. Further work needs to be conducted to clarify exactly our position on that side.

References

1. Barbeau, M.: Software Engineering for Telecommunications Protocols. Web page, www.scs.carleton.ca/∼barbeau/Courses/SETP/index.html
2. Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: Linux Kernel Internals - Second Edition. Addison Wesley Longman (1998)
3. Birman, K., Constable, R., Hayden, M., Kreitz, C., Rodeh, O., van Renesse, R., Vogels, W.: The Horus and Ensemble Projects: Accomplishments and Limitations. Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX '00). Hilton Head, South Carolina (2000)

4. Czarnecki, K., Eisenecker, U.W.: Generative Programming Methods, Tools, and Applications. Addison Wesley (2000)
5. Czarnecki, K. Eisenecker, U.W.: Components and Generative Programming. Proceedings of the 7th European Engineering Conference held jointly with the 7th ACM SIGSOFT symposium on Foundations of software engineering (1999) 2–19
6. Hutchinson, N.C., Peterson, L.L.: The x-Kernel: An Architecture for Implementing Network Protocols. IEEE Transactions on Software Engineering **17** (1) (1991) 64–76
7. LAN/MAN Standards Committee of the IEEE Computer Society: Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-Speed Physical Layer Extension in the 2.4 Ghz Band. IEEE Std 802.11b (1999)
8. Mellquist, P.E.: SNMP++: An Object-Oriented Approach to Developing Network Management Applications. Prentice Hall (1998)
9. van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: Building Adaptive Systems Using Ensemble. Software–Practice and Experience **28** (9) (1998) 963–979
10. Schmidt, D.C., Huston, S.D.: C++ Network Programming: Systematic Reuse with Frameworks and ACE. Addison-Wesley Longman (2002)
11. Schmidt, D.C., Huston, S.D.: C++ Network Programming: Mastering Complexity Using ACE and Patterns. Addison-Wesley Longman (2001)
12. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. Wiley and Sons (2000)
13. Wade, I.: NOSintro - TCP/IP Over Packet Radio - An Introduction to the KA9Q Network Operating System. Dowermain Ltd. (1992)
14. Wright, G.R., Stevens, W.R.: TCP/IP Illustrated - Volume 2. Addison Wesley (1995)
15. Xu, Y.: Implementation of Location Detection, Home Agent Discovery and Registration of Mobile IPv6. Master Thesis, School of Computer Science, Carleton University (2001) (Available at: www.scs.carleton.ca/∼barbeau/Thesis/xu.pdf)
16. Zhang, S.: Channel Management, Message Representation and Event Handling of a Protocol Implementation Framework for Linux Using Generative Programming. Master Thesis, School of Computer Science, Carleton University (2002) (Available at: www.scs.carleton.ca/∼barbeau/Thesis/zhang.pdf)