Frameworks for protocol implementation

Ciro de Barros Barbosa[1]
Luís Ferreira Pires
Marten van Sinderen

Centre for Telematics and Information Technology
University of Twente
Enschede, the Netherlands
e-mail: {barbosa, pires, sinderen}@cs.utwente.nl

Resumo. Este artigo reporta o desenvolvimento de um catálogo de molduras ('frameworks') para a implementação de protocolos. Molduras são estruturas de software desenvolvidas para um domínio de aplicação específico, e que podem ser reutilizadas para vários sistemas concretos dentro desse domínio. Através do uso de molduras nós visamos aumentar a eficiência do processo de implementação de protocolos. Nós assumimos a premissa de que quando protocolos são implementados diretamente de suas especificações é possível se aumentar a correção e o desempenho do processo de implementação, e a capacidade de manutenção do sistema resultante. Nós argumentamos que molduras devem ser definidas de acordo com os conceitos que suportam as técnicas usadas para especificar protocolos. Conseqüentemente nós acoplamos o desenvolvimento de molduras para implementação de protocolos à investigação dos diferentes modelos de projeto usados na especificação de protocolos. Esse artigo apresenta a forma de trabalho que estamos utilizando no desenvolvimento de molduras e exemplifica essa forma de trabalho com um exemplo de moldura.

Abstract. This paper reports on the development of a catalogue of frameworks for protocol implementation. Frameworks are software structures developed for a specific application domain, which can be re-used in the implementation of various different concrete systems in this domain. By using frameworks we aim at increasing the effectiveness of the protocol implementation process. We assume that whenever protocols are directly implemented from their specifications one may be able to increase the correctness and the speed of the implementation process, and the maintainability of the resulting system. We argue that frameworks should match the concepts underlying the techniques used for specifying protocols. Consequently, we couple the development of frameworks for protocol implementation to the investigation of the different alternative design models for protocol specification. This paper presents the approach we have been using to develop frameworks, and illustrates this approach with an example of framework.

1 Introduction

The need to speed up the implementation process and to facilitate the maintenance and extension of the system to be built has inspired the development of software design methods in the 90's. Examples of these methods are object-oriented analysis and programming [2, 9], design patterns and frameworks [6, 7, 10].

Frameworks are software structures developed for a specific application domain, which can be re-used in the implementation of various different concrete systems in this domain. Experience with the use of frameworks has shown that they can increase the effectiveness of the implementation process [7, 10].

Our work considers the implementation of protocols from their specifications. Different alternative notations can be used to specify a protocol, whereas each of these notations is based on a specific design model. A design model consists of a set of design concepts and the rules for combining them.

1. sponsored by CNPq - Brasília/Brazil.

Examples of design concepts used for specifying protocols are interactions (synchronous or asynchronous), processes and interaction means (gates or channels).

In our research we intend to apply frameworks to support the implementation of protocols in a correct, efficient and effective way. These frameworks should be compatible with the design concepts underlying the techniques used for specifying protocols. Consequently, we couple the development of frameworks for protocol implementation to the investigation of the different alternative design models for protocol specification.

The main long term objective of our work is to develop a catalogue of frameworks for protocol implementation. When developing frameworks for a specific design model we have observed that a multitude of alternative implementation decisions could still be taken, all resulting in valid frameworks. These implementation decisions concern, for example, the techniques for implementing parallel processes, interactions and interfaces. This implies that our catalogue will get the form of a table, indexed with design models and implementation decisions.

By constructing frameworks for different design models we should be able to develop a comprehensive design method for protocol implementation. This method offers flexibility to implementers, since it is not limited to a single design model. Once enough experience has been obtained with the development of frameworks for different design models, we intend to investigate the commonalities between these frameworks. This opens the possibility of developing more general frameworks, which can be applied for different design models and alternative implementation decisions in a flexible way.

Although our approach is general enough to be applied to the implementation of protocols in different implementation environments, we have decided to limit our experiments to SUN workstations, running the Solaris operating system, and using the C++ programming language, for practical reasons.

This paper is further structured as follows: Section 2 presents our approach to develop frameworks, Section 3 discusses the development of the Events Manager framework, Section 4 introduces a protocol stack example used to illustrate the use of this framework, Section 5 discusses the implementation of this protocol stack using the Events Manager framework and Section 6 presents some conclusions and suggestions for further work.

2 Approach

This section discusses the motivation for developing a catalogue of frameworks and the methods applied in the development and usage of a framework.

2.1 The catalogue

In our research we strive for results that can be applied in practical situations. We have based the development of our catalogue of frameworks on the following observations:

1. protocols are presented in different alternative forms. There are many different design models with which one can specify a protocol.
2. in essence protocol specifications consist of concurrent and interacting processes, representing interacting protocol entities or protocol functions. There are many alternative implementation solutions that can be used to implement concurrent and interacting processes.

When we combine these two observations we conclude that it is possible to build a catalogue of frameworks for implementing protocols by systematically considering different alternative design models and different alternative implementation solutions. This means that this catalogue can be seen as a two-dimensional matrix indexed with design models along one dimension, and implementation solutions along the other dimension.

In our catalogue, a framework for protocol implementation consists of a set of classes that implement the common aspects of all protocols that can be described according to a specific design model. This also implies that when we use a framework to implement a specific protocol, the final implementation will consist of the framework complemented with classes that define the specific functions of the protocol. In this respect a framework resembles a program with a 'hole', which has to be 'filled in' with specific functions in order to be executed.

Figure 1 depicts our catalogue of frameworks. The framework icon in Figure 1 reflects our metaphorical view of a framework.

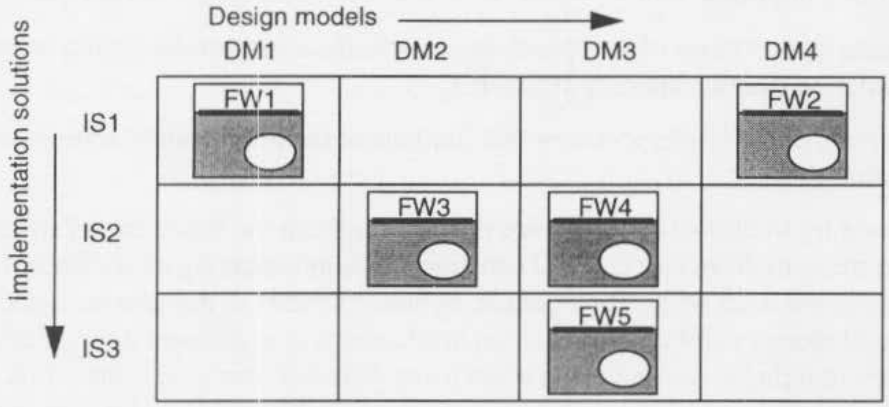

Figure 1. Catalogue of frameworks

2.2 Developing a framework

When developing a framework we have to determine the design model that the framework is expected to support and the implementation solution that will be applied for building the framework.

Design models

Design models represent properties of a system. In order to be supported by a framework a design model should have some form of execution model, either formally defined by an operational semantics or defined by simulation tools that are able to execute the model. These design models are normally called *constructive models* [8].

Furthermore, we consider design models that are capable of representing the structure of the system in terms of its components and the behaviour of these components. These characteristics make a design model applicable to protocol specification.

After identifying design models we intend to develop a classification of these models, possibly in terms of a hierarchy. For the time being we have identified two families of design models, namely the synchronous event-based and asynchronous event-based models. Inside these families we can identify sub-families. For example, the synchronous event-based family has the multi-way and the two-way synchronization sub-families. These sub-families can be again decomposed, by considering some other aspects of the behaviour models, such as the representation of input-output events. An example of a classification of models can be found in [8]. One of the important research activities in our work is to check whether such a classification of design models is also useful for developing frameworks to support these models.

Implementation solutions

There is a multitude of techniques and components that can be used to implement concurrent and interacting processes. Examples of these techniques and components are:

- replacement of concurrency by sequences of activities controlled by a scheduling function;
- use of available scheduling mechanisms that simulate concurrency inside an operating system process (e.g., threads);
- mapping of different concurrent processes of the specification onto different operating system processes.

In our approach we intend to preserve as much as possible the specification structure in the implementation structure, in order to facilitate correctness assessment and maintenance. Based on this intention we observe that two important issues should be solved when developing a framework:

1. the mapping of processes of the specification (specification structure) onto constructs of the implementation (implementation structure);
2. the constructs of the implementation that implement the interactions between processes of the specification.

Ideally one should try to dissociate the choice of mapping from the specification structure onto the implementation structure from the choice of constructs for implementing interactions. In practice these choices often influence each other. For example, by using threads to implement concurrent processes one can use shared memory and synchronization mechanisms to implement interactions between these threads in a more straightforward way than when using different operating system processes. In future we intend to investigate to what extent these two choices can be considered as orthogonal and can be made separately of each other.

Defining a framework

By using object-oriented design, we can consider a framework as a set of cooperating classes that can be reused to facilitate the implementation of different specific systems in a certain application domain. A framework captures implementation solutions that are common to its application domain. In our case the application domain of a framework is initially limited to a specific design model. The classes of a framework are responsible for supporting the design concepts of the design model using an implementation solution.

The development of a framework follows the object-oriented approach, as presented in, e.g., [2, 9]. We start by performing *domain analysis*, in which the concepts of the design model and their relationships are analysed and represented in conceptual class diagrams. After that we develop a software architecture, in which the classes of the software implementation and their relationships and responsibilities are fully defined in class diagrams and interaction diagrams. The coding of these classes and their deployment is done according to these definitions.

Documenting frameworks

Frameworks in our catalogue should be documented in such a way that their purpose, limitations, structure, etc. can be easily retrieved. The following information should be found in the definition of a framework:

- *Framework name*: an identifier for the framework, preferably related to its main characteristics or classes;
- *Structure*: the classes of the framework and their relationships, represented using class diagrams and interaction diagrams, such as in [9];
- *Participants*: more detailed definition of each class of the framework, in terms of its attributes, methods, responsibilities and cooperation with other classes;
- *Consequences*: limitations of the framework;

- *Usage*: guidelines for using the framework to implement specific protocols.

2.3 Using a framework

Each framework in our catalogue is accompanied with guidelines for using the framework to implement protocols (*Usage* information). Depending on the framework, these guidelines may consist of:

- classes that have to be specialized for each specific protocol implementation;
- template code to be filled in with code for each specific protocol function;
- suggested mappings of behaviour structures of the protocol specification onto pieces of code that use the template code.

By defining these guidelines precisely, for example, by explicitly defining the mapping of behaviours onto code that uses the template, one could implement tools that compile specifications to protocol implementations that use the framework. We do not address the construction of compilers in our research, since we concentrate on the problems related to the development and use of frameworks for protocol implementation. However, our results should give enough basis for the construction of compilers for some of the design models and frameworks being considered.

2.4 Generic components

Some components (modules or classes) can be applied in many different frameworks for protocol implementation. Examples of such components are lists and data packets. The identification of such generic components can be seen as a step towards the integration of frameworks.

3 The Events Manager framework

This section discusses the development of the Events Manager framework. This framework has been developed according to the approach described before and using methods and notations for object-oriented design [9].

3.1 Domain analysis

The design model considered for the development of this framework consists of a collection of *components*. Each component can be attached to another component, or to an interface with the environment of the system or both. An attachment between two components or an interface between a component and the environment of the system is indistinctly denoted as an *interaction point*. A component can be structured in terms of sub-components, which generates a hierarchy of components. A parent component contains child components; a child component is contained in a parent component.

Figure 2 gives an example of configuration of components and interaction points.

In Figure 2 components A and C are attached to the environment of the system through an interface. Components A and C are also attached to component B. Component B consists of (sub-)components B1 and B2.

The framework supports synchronous interactions between two components; components can only interact if they are attached to each other. Interactions between a component and the environment of the system follow the pattern dictated by the interface between them. In this framework these interactions are asynchronous, in the sense we make use of the buffering facilities available in the socket and pipe packages.

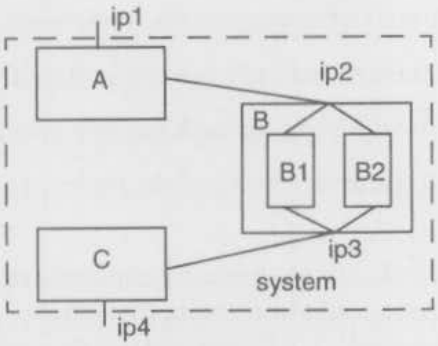

Figure 2. Illustration of the design model supported by the Events Manager framework.

Events are occurrences relevant to the functioning of the system. Such an occurrence can be an interaction between a component and its environment through an external interface (external event), an interaction between two components (internal event) or the expiration of a timeout (timeout event). Service primitives of the protocol designs being implemented become either external or internal events, depending on the mapping of protocol layers and functions onto components. Since service primitives are typed and contain information (parameter values), an external or internal event has to be extended in order to convey the type and information related to a service primitive.

An event may depend on the fulfilment of logical *conditions* involving former events or characteristics of the event itself. These conditions do not involve timing aspects, which are already handled by the timeout events. An *action* consists of an activity that is triggered by the occurrence of an event. Once an event occurs, its actions are executed atomically, i.e., they all run to completion and cannot be interrupted by other events or actions.

Each component has a *behaviour*, which consists of the events allowed by this component, the conditions of these events and the relationships between these events. We assume that the behaviour of a component can be represented in terms of a finite state machine, although other alternative representations are not necessarily disallowed.

Figure 3 depicts the concepts identified above and their relationships in terms of a class diagram [9].

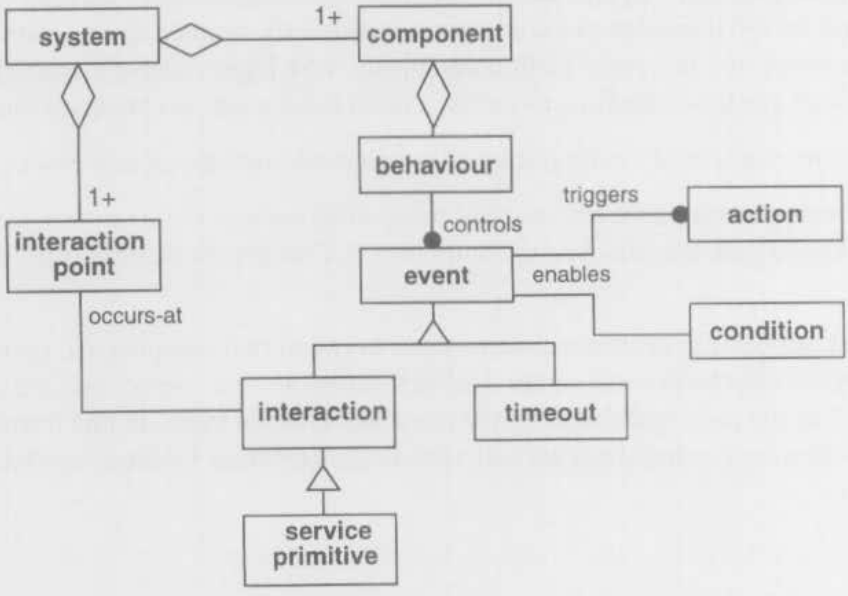

Figure 3. Class diagram of the design model supported by the Events Manager framework.

3.2 Software architecture

The software architecture of the Events Manager framework supports the design model mentioned above. This implies that this software architecture has to incorporate design decisions that support a hierarchy of components that interact through interaction points.

In this framework each component is implemented as an object. The collection of components of a system is altogether implemented in a single operating system process. Interactions between components are implemented by means of method calls, i.e., a component takes the initiative of an interaction with another component by calling one of its methods.

The basic functionality of a component in the framework is represented in an abstract class called eventHandler. This functionality makes it possible for a component to handle events and enables the management of a set of components. Each component is implemented as a concrete class that inherits from eventHandler. This concrete class contains the protocol specific functions of a component.

The execution of events is controlled by an object instantiated from a class called eventsManager. In each system there is only one eventsManager object active at a time. Since the eventsManager object has a central role in this framework we have decided to call it 'the Events Manager framework'.

Abstract class eventPoint contains the information on the mechanisms for the execution of external and timeout events. Each possible implementation of these mechanisms is represented by a concrete class that inherits from class eventPoint. When an concrete component (subclass of eventHandler) is created it generates eventPoint objects for executing its events. Each eventPoint object registers itself to the eventsManager.

Figure 4 depicts the class diagram of the software architecture.

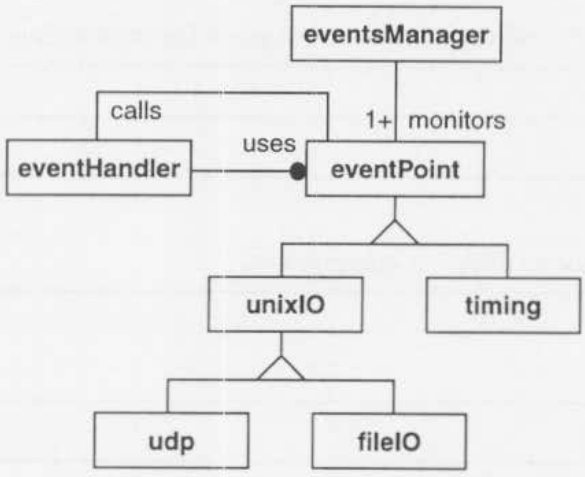

Figure 4. Class diagram of the framework's software architecture.

3.3 Definition of classes

This section defines systematically each class identified in the software architecture of the Events Manager framework, in terms of its responsibilities, collaborators, attributes and methods. This section

consists of a set of tables in the form of 'index cards', one for each class. Attributes and methods of a sub-class are not repeated, unless it overrules or defines methods of its super-class.

Class eventsManager
Responsibility • registers and de-registers event points • monitors the execution of events at the event points • gives control to an event point once an event is detected
Collaborators eventPoint
Attributes • list<eventPoint> _epList // list of monitored event points

Methods
• void WaitEvents() // loop that monitors the occurrence of events
• void AddEP(eventPoint*) // registers an event point for monitoring
• void RemoveEP(eventPoint*) // de-registers an event point for monitoring

Class eventHandler
Responsibility • offers a common reference to different components
Collaborators eventPoint
Attributes
Methods

Class eventPoint
Responsibility • informs its associated event handler that an event has happened
Collaborators eventsManager, eventHandler
Attributes • eventsManager* _em // associated eventsManager • eventHandler* _eh; // associated eventHandler object

Methods

• eventHandler* GetHandler() // gets the associated eventHandler object
• virtual int Type() // gets the type of the event point
• virtual void CallEH(int, char*) // informs the event point that the event has happened, by asking the event point to call its event handler

Class unixIO
Responsibility • specializes class eventPoint for UNIX I/O
Collaborators eventsManager, eventHandler
Attributes • int _descr; // file or socket descriptor

Methods
• virtual int Type() // gets the type of the event point
• int GetFd() // gets the file descriptor of the event point (if any)
• virtual void CallEH() // abstract method for treating interactions

Class udp
Responsibility • maintain the information on the mechanism used for executing an event • specializes class eventPoint for using UDP through a socket
Collaborators eventsManager, eventHandler
Attributes • int _port; // port number

Methods
• int Type() // gets the type of the event point
• int GetData(struct sockaddr_in*, char**) // gets a reference to buffer containing received data
• int SendData(struct sockaddr_in*, char*) // sends data contained in a buffer
• virtual void CallEH() // abstract method for treating interactions

Class fileIO
Responsibility • maintain the information on the mechanism used for executing an event • specializes class eventPoint for using file I/O (e.g., standard I/O or file)
Collaborators eventsManager, eventHandler
Attributes • FILE* _stream // interaction stream

Methods
• int Type() // gets the type of the event point
• int GetData(char**) // gets a reference to buffer containing received data
• int SendData(char*) // sends data contained in a buffer
• virtual void CallEH() // abstract method for treating interactions

Class timing
Responsibility • specializes class eventPoint for handling timeout
Collaborators eventsManager, eventHandler
Attributes • long int _nextto; // time moment of next timeout

Methods
• int Type() // gets the type of the event point
• void SetTimeout(int) // sets a timeout for current time plus the given value (in seconds)
• void TimeLeft() // informs the time left before the next timeout
• virtual void CallEH() // abstract method for treating interactions

3.4 Dynamic behaviour

During the initialization of the system the eventsManager object and some initial components are instantiated. Once initialization is completed, the eventsManager object executes a loop (method WaitEvents) in which it checks for events that have occurred. Once an event is detected, the eventsManager object calls the eventPoint object that corresponds to this event. This eventPoint object then calls the eventHandler object that treats the event. When the eventHandler is ready the control returns to the eventsManager.

Figure 5 depicts the interaction diagram for the execution of an external event in this framework.

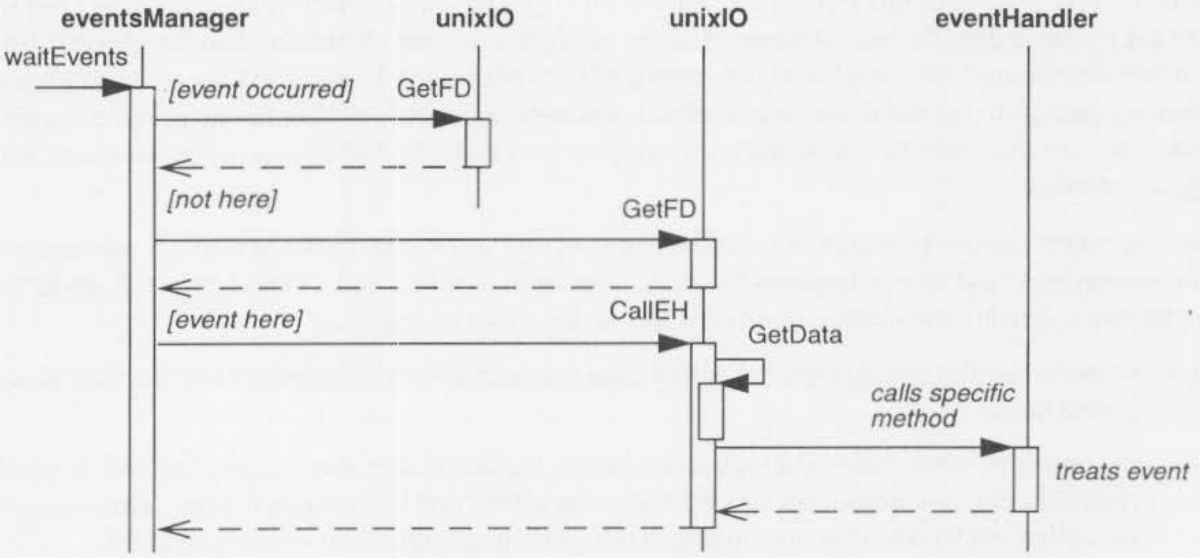

Figure 5. Interaction diagram for the treatment of an external event.

In Figure 5 we represent the situation in which the eventsManager object monitors events at two eventPoints of type unixIO[1]. Once the eventsManager object detects that an event has occurred (by using the select system call), it goes through the list of eventPoints, asking their file (socket) descriptors. In this way the eventPoint (unixIO) in which an event has occurred can be determined. In Figure 5 we suppose that an event happened at the second unixIO object being asked. Once the eventsManager has found an unixIO object in which an event has happened, it calls the method CallEH of this object. This method gets the data from the corresponding file or socket descriptor (method getData) and gives the information on the event (e.g., a service primitive) by calling a specific method that treats the event in the specialization of the eventHandler (a component). This component takes the appropriate measures for treating the event and eventually its method returns, causing callEH to return. At this point a new cycle of event execution can begin.

3.5 Usage Information

In order to implement protocols with the Events Manager framework one has to define the protocol specific functions (components). These components are specializations of the eventHandler class. In the component code, specific methods are defined to treat the different events. Depending on the protocol, it is advisable to structure this code in terms of functions that get an input and generate one or more outputs, such as in Section 5.2, or in terms of a finite state machine, such as in Section 5.3.

The implementation also contains initialization code, in which the eventsManager object, the main component objects and the main eventPoint objects are created, and the waitEvents loop of the eventsManager object is started.

Concrete event points are created as specializations of the fileIO, udp or timing class, either in the initialization code or in a component. In case multiple child components with a common parent share an event point, this event point has to be instantiated in the parent component. In the case of a fileIO or udp object, a specialization may capture the information conveyed in service primitives that occur at this event point.

1. In this text we use the specialization hierarchy as a property for typing. According to our class hierarchy an eventPoint object is of type unixIO or timing, and an unixIO object is of type udp or fileIO.

3.6 Consequences

In the current version of this framework, only inputs from the environment (external inputs) and time-outs are controlled by the eventsManager. Output events and internal events are handled directly by the components themselves. This limits the control of the interleaving between events; once an external input happens, it triggers a chain of related internal events, possibly ending in one or more output events. In case this chain of events takes too long, this may cause loss of data or an inconvenient 'blocking' at interfaces.

Internal events are implemented as procedure calls in this version of the framework. A procedure call can be seen as a kind of synchronous interaction between a calling and a called object. In order to implement a synchronous interaction correctly, we have two alternatives:

1. in the behaviour description, the called object should always be prepared to accept all possible interactions;
2. the return of a procedure call indicates whether the interaction was accepted or not. A negative return (interaction not accepted) means that the interaction has not taken place. This means that the called object was in a state in which this particular interaction was not enabled.

4 A protocol stack example

This section presents the protocol stack that is used to illustrate the application of the Events Manager framework.

4.1 Overview

The protocol stack used in this paper consists of two protocols: the conference protocol and the multicast protocol. The conference protocol supports the conference service, making use of the multicast service supported by the multicast protocol. The multicast protocol operates on top of a connectionless service, such as the service supported by UDP [3].

Figure 6 shows the global structure of our protocol stack.

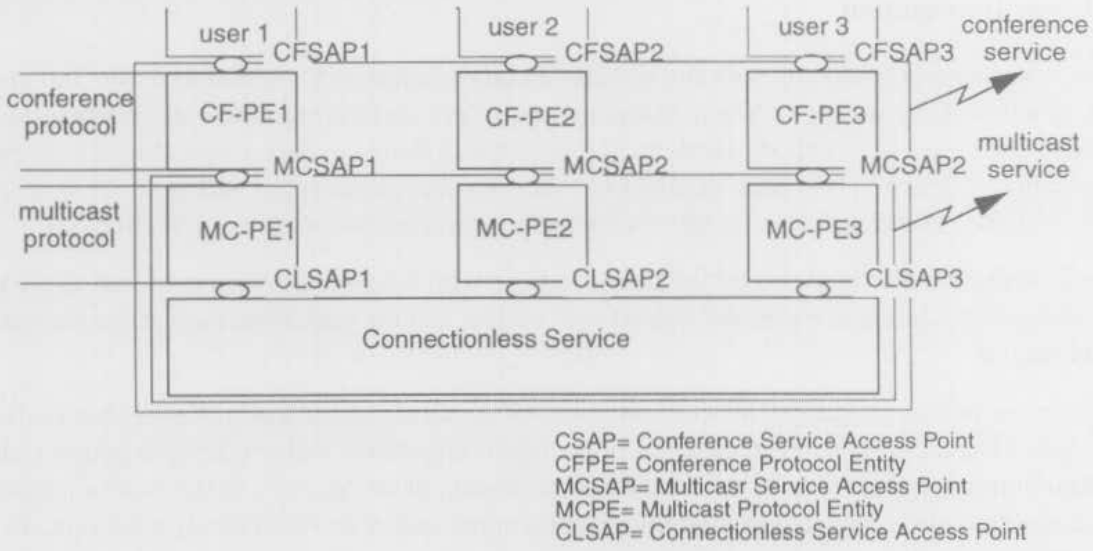

Figure 6. Global structure of our example protocol stack.

The protocol stack defined in this example has been artificially introduced, since it would be possible to define a single protocol that supports the conference service on top of a connectionless service. However, our protocol stack makes it possible to illustrate the use of the Events Manager framework

for more than one protocol. This example enables us to present and discuss some interesting implementation constructs, and it is simple enough so that it does not blur the discussion on the application of the framework. This protocol stack is documented in detail in [4].

4.2 The conference service

The conference protocol supports the conference service. A conference is defined as the context in which a group of users can exchange messages. Every user in a conference can send messages to all other conference partners participating in that conference, and it can receive messages from every other participant. The participants in a conference can change dynamically, since the conference service allows its users to join and leave a conference. We assume that different conferences can exist at the same time, but each user can only participate in at most one conference at a time.

The conference service has the following service primitives:

- *join*: a user joins a conference and defines its user title in this conference. The user title is simply a name that is assigned to a user in a conference;
- *datareq*: a user sends a message to the other users participating in the conference;
- *dataind*: a user receives a message from another user participating in the conference;
- *leave*: a user leaves the conference. Since a user can only participate in one conference at a time, there is no need to identify the conference in this primitive.

Initially, a user is only allowed to perform a *join* primitive. After this, the user is allowed to send messages, by performing *datareq*'s, or to receive messages, by performing *dataind*'s. In order to stop its participation in the conference, a user performs a *leave* at any time after it has performed a *join*.

4.3 The multicast service

The multicast service allows its users to send messages to a set of users in a single primitive interaction. The multicast service has the following service primitives:

- *mc-datareq*: a user sends a message to a set of service users. This set of users is called the *destination set*. A special destination parameter value indicates that the message should be sent to all known service users;
- *mc-dataind*: a user receives a message from another user.

The multicast service is unreliable, which implies that messages sent to a set of users may not arrive at one or more of these users. However, we impose that in case a message arrives it is delivered to its intended destination and it is not corrupted. Messages may not be delivered at a user in the sequence which they have been sent. A special destination set parameter value indicates that data should be sent to all MCSAP addresses reachable from the source MCSAP, except to the source MCSAP itself.

4.4 The conference protocol

The conference protocol is responsible for the administration of conference participants and for the data transfer between participants.

The conference protocol has the following protocol data units (PDUs):

- *join-PDU*: informs the other protocol entities that this protocol entity joins a certain conference. A user title and a conference identifier are conveyed in this PDU;
- *answer-PDU*: answers a protocol entity that has sent a *join-PDU*, and contains the user title of

the answering protocol entity;

- *data-PDU*: contains a message to be delivered to the other conference participants;
- *leave-PDU*: informs the other conference participants that a participant is leaving the conference.

Each protocol entity keeps a set of *conference partners*, which consists of a set of pairs, each pair consisting of a MCSAP address and a user title. The set of conference partners is initially empty when a protocol entity starts its operation, since a protocol entity initially does not participate in any conference.

The normal behaviour of a protocol entity is defined in terms of simple rules as follows:

1. each protocol entity that performs a *join* primitive sends *join-PDU*s to all MCSAP addresses reachable from its MCSAP, using the special destination set parameter value of the *mc-datareq* primitive;
2. a protocol entity that receives a *join-PDU* and is engaged in the conference identified in this *join-PDU* sends an *answer-PDU* to the source of the *join-PDU*, and includes the protocol entity that sent the *join-PDU* in its set of conference partners;
3. a protocol entity that receives a *join-PDU* and is not engaged in the conference identified in this *join-PDU* ignores the *join-PDU*;
4. a protocol entity that receives an *answer-PDU* keeps the MCSAP address and the user title of this *answer-PDU* in its set of conference partners;
5. a protocol entity that performs a *datareq* sends the message of this *datareq* to all MCSAPs of the set of conference partners, through an *mc-datareq*;
6. a protocol entity that receives a *data-PDU* in an *mc-dataind* delivers the message contained in this *data-PDU* to its user by executing a *dataind*. The user title parameter of the *dataind* is obtained by translating the MCSAP source address to the corresponding user title, according to the information contained in the set of conference partners;
7. a protocol entity that performs a *leave* sends a *leave-PDU* to its set of conference partners and clear its set of conference partners;
8. a protocol entity that receives a *leave-PDU* removes the source MCSAP address and the user title of the *leave-PDU* from the set of conference partners.

This protocol includes a limited treatment of exception situations. More details on this protocol can be found in [4].

4.5 The multicast protocol

The multicast protocol entities communicate with each other using the connectionless service provided by UDP. The service primitives of this connectionless service can be modelled as:

- *cl-datareq (destination address, data)*: a user sends data to another user
- *cl-dataind (source address, data)*: a user receives data from another user

Since the *cl-dataind* primitive indicates the source address, there is no need to define any Protocol Control Information in the PDU of this protocol. This also implies that the behaviour of the protocol entities is defined as a mapping of:

- an *mc-datareq* onto *cl-datareq*s to the CLSAPs that correspond to the MCSAPs of the destination set, at the sending side;

- an *cl-dataind* onto an *mc-dataind* at the receiving side.

Each multicast protocol entity must know a set of MCSAP addresses and their corresponding CLSAP addresses. This information makes it possible for a multicast protocol entity to address other multicast protocol entities. In the implementation this information is made available to the protocol entities during initialization.

4.6 Protocol stack operation

Figure 7 illustrates the operation of our protocol stack with a simple execution scenario.

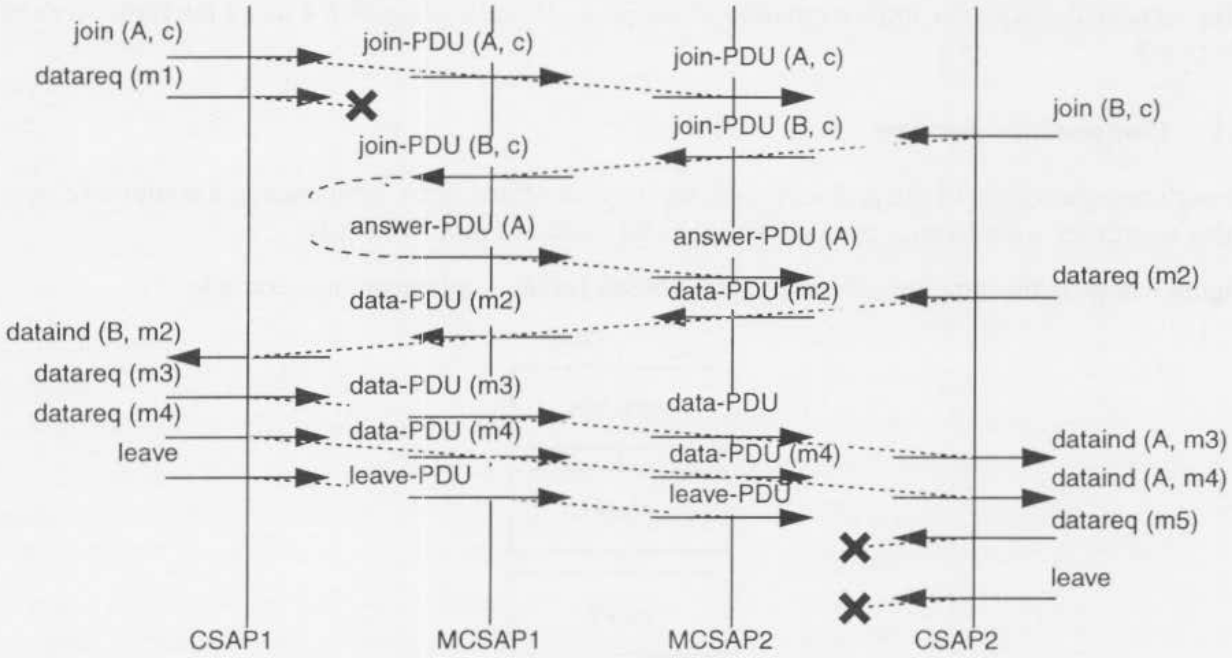

Figure 7. Instance of protocol behaviour

The following situations are shown in Figure 7:

- the user at $CSAP_1$ executes a *join* to conference *c* and with user title *A*. A *join-PDU* is generated, and sent to all MCSAP addresses. Since no other users participate in the conference at this time, this protocol entity receives no *answer-PDU* back, and the set of conference partners of this protocol entity remains empty;
- the user at $CSAP_1$ executes a *datareq* to send message m_1, but since the set of conference partners of the protocol entity is empty, message m_1 cannot be sent and is discarded;
- the user at $CSAP_2$ executes a *join* to conference *c* and with user title *B*. A *join-PDU* is generated, and sent to all MCSAP addresses. Since the user at $CSAP_1$ is the only user that participates in the conference, its protocol entity sends an *answer-PDU* with user title *A* as a response to the protocol entity at $MCSAP_2$. The protocol entity at $MCSAP_2$ updates its set of conference partners, by including $MCSAP_1$ and user title *A* in this set, while the protocol entity at $MCSAP_1$ updates its set of conference partners, by including $MCSAP_2$ and user title *B* in this set;
- the user at $CSAP_2$ executes a *datareq* to send message m_2, which generates a *data-PDU* that is sent to all elements of the set of conference partners of the protocol entity. The *dataind* caused by the arrival of this data-PDU at the receiving protocol entity contains the user title *B* associated with the sending protocol entity at $MCSAP_1$;
- the user at $CSAP_1$ sends messages m_3 and m_4, which are coded in *data-PDUs* that are forwarded to the protocol entity at $MCSAP_2$. These messages are finally delivered to the user at $CSAP_2$;

- the user at $CSAP_1$ executes a *leave*. A *leave-PDU* is generated and sent to all elements of the set of conference partners of the protocol entity at $MCSAP_2$. When the *leave-PDU* reaches its destination, the set of conference partners of the protocol entity at $MCSAP_2$ is updated to an empty set;
- subsequent *datareq* and *leave* primitives are simply discarded, since the set of conference partners of the protocol entity at $MCSAP_2$ is empty.

5 Protocol stack implementation

This section discusses the implementation of the protocol stack of Section 4 using the framework of Section 3.

5.1 Components overview

In our implementation of the protocol stack we have identified three components: a conference user (user interface), a conference protocol entity and a multicast protocol entity.

Figure 8 depicts the structure of components chosen for the implementation example.

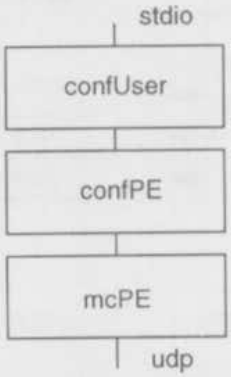

Figure 8. Structure of components for the implementation example.

5.2 The multicast protocol entity

The multicast protocol entity component is defined in the class mcPE, which is a specialization of eventHandler, extended with methods ClDataInd and McDataReq to handle the *cl-dataind* and *mc-datareq* primitive, respectively. The mcPE object is related to an mcUser object, which defines a prototype for a multicast service user. Class mcUser is a specialization of eventHandler and offers a prototype to method MCDataInd. In the conference protocol entity module the *mc-dataind* service primitive is implemented by defining method McDataInd (see Section 5.3).

The mcPE object uses an mcAccessPoint object, which implements an MCSAP. Class mcAccessPoint specializes class udp with knowledge about the multicast protocol. The mcPE object also contains two parts: mcSender and mcReceiver, to send and receive messages on behalf of the service users, respectively. Both parts make use of a peersDirectory object, which maintains addressing information (MCSAP address, IP address and port number) necessary to reach and identify peer protocol entities.

Figure 9 depicts the class diagram of the multicast protocol entity module.

5.3 The conference protocol entity

The conference protocol entity component is defined in the class confPE, which is a specialization of mcUser, extended with methods to handle the conference service primitives. The confPE object is

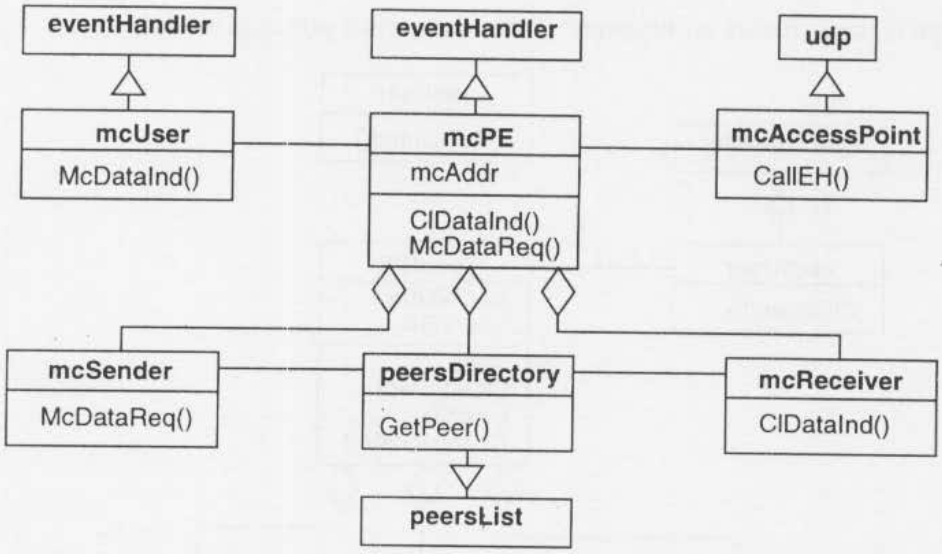

Figure 9. Class diagram of the multicast protocol entity module.

related to a confUser object in a similar way as an mcPE object was related to an mcUser object in Section 5.2.

The behaviour of the confPE object has been implemented by considering it as a finite state machine with two states. Figure 10 depicts this state machine in terms of a state table.

	stateIdle	*stateBusy*
answer-PDU	- ignore-PDU	- update partners
join-PDU		- update partners; - send answer-PDU
leave-PDU		- update partners
data-PDU		[sender known]: - execute dataind [sender unknown]: - send join-PDU
join	- send join-PDU; - change to *stateBusy*	not allowed
datareq	not allowed	- send data-PDU
leave	not allowed	- send leave-PDU; - change to *stateIdle*

Figure 10. State machine for the confPE behaviour.

Based on the state machine depicted in Figure 10 we applied the state design pattern of [6] to develop the software architecture of the conference protocol module. The state design pattern consists of a context, a class representing a generic state, and a class for each state. In our case the context is represented by the confPE class, the generic state is called confState, and classes stateIdle and stateBusy represent their respective states. Classes stateIdle and stateBusy implement the actions to be taken as a result of a certain event when the conference protocol entity is in each of these states.

The confPE object uses a pdu object, which encapsulates the functions to manipulate (create, encode and decode) PDUs. Since the conference protocol entity does not store PDUs and treats one PDU at a time, a single pdu object is enough.

The confPE also maintains a partnersDirectory, which keeps track of the partners participating in a conference. The partnersDirectory class is a sub-class of a partnersList, included to facilitate the manipulation of partners information (MCSAP address and user title).

Figure 11 depicts the software architecture of the conference protocol module.

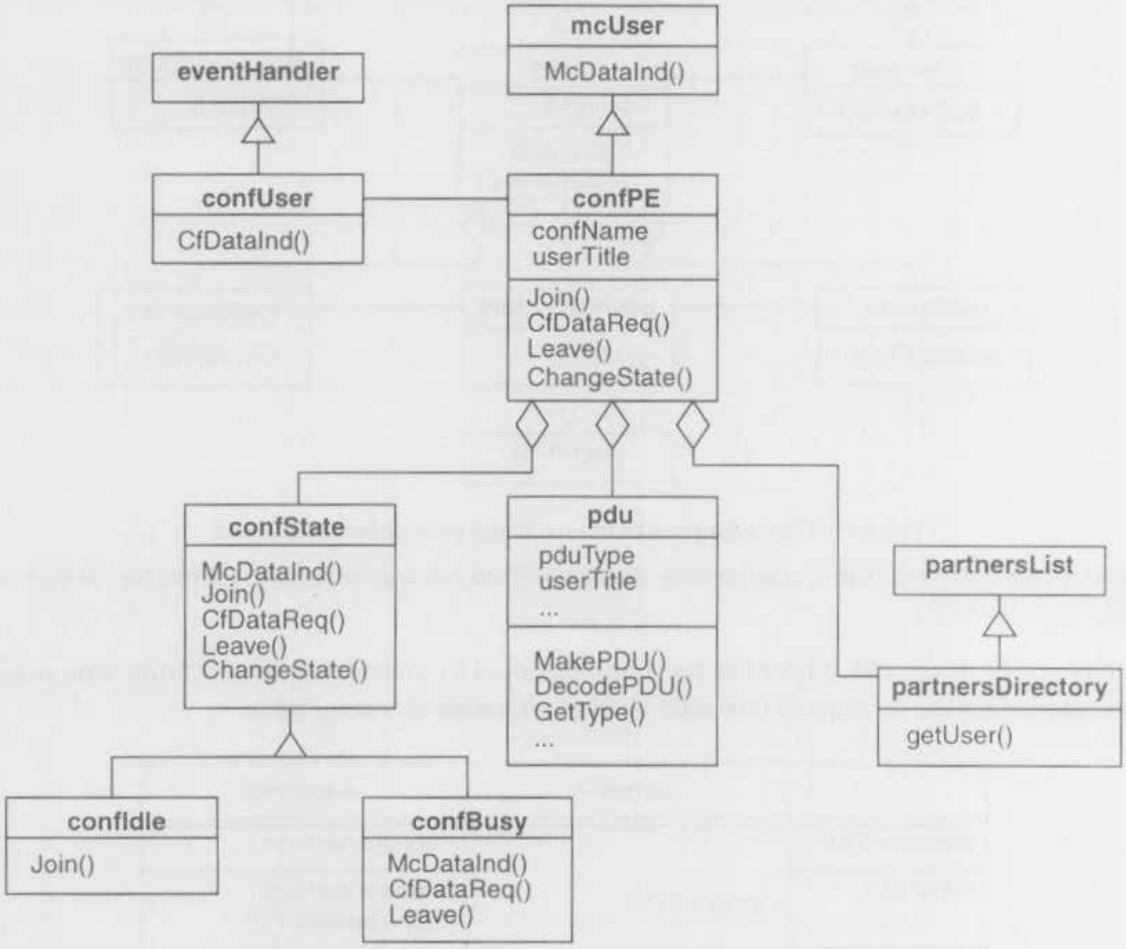

Figure 11. Class diagram of the conference protocol entity module.

5.4 The user interface

This module provides an interface to a human user that uses the service provided by the conference protocol. This module allows a user to join a conference, send and receive messages, and leave the conference. This module is connected to the confPE by having a sub-class of confUser, in a similar way as in the case of the multicast protocol (see Section 5.2). For the sake of conciseness we refrain from giving details on this module.

6 Conclusions

This paper discusses the development of frameworks for protocol implementation. The approach proposed in this paper is illustrated with a simple framework and its application to implement a protocol stack consisting of a conference and a multicast protocol.

In our research we investigate frameworks for protocol implementation. Such a framework supports a design model, and incorporates implementation solutions concerning the mechanisms used to implement concepts of this design model. For example, the design model supported by the Events Manager framework consists of synchronous and two-party internal interactions, asynchronous external inter-

actions and components that can be hierarchically organized. The implementation solutions used in this framework are to implement components as objects, to implement internal interactions as procedure (method) calls, to implement external interactions using sockets or file I/O, and to map the whole system onto a single operating system process. The external events are monitored by an eventsManager object, which implies that this framework implements the scheduling of the handling of events, as opposed to using available scheduling mechanisms.

By varying the design model and the implementation solutions we intend to generate a catalogue of frameworks. A framework consists of some classes that are ready for use, and usage rules in the form of templates. This should accelerate the implementation of protocols, since implementers can choose a framework in accordance with the design model used in the description of the protocol they want to implement. The usage rules indicate how to translate the protocol design to a running implementation.

Although we have fully discussed a framework and the implementation of a protocol stack in this paper, we feel that the most important contribution of this paper is the approach to the development of frameworks presented here. Our future work will consist of the development and documentation of more entries to the catalogue of frameworks, the categorization of these entries and the development of precise (possibly formal) methods to define the mappings from a protocol design to its implementation using the frameworks.

References

[1] A. Ananthaswamy. *Data communications using object-oriented design and C++*. McGraw-Hill, Inc., USA, 1995.

[2] G. Booch. *Object-oriented analysis and design with applications*. The Benjamin/Cummings Publishing Company, Inc., California, USA, 1994.

[3] D. E. Comer. *Internetworking with TCP/IP. Volume I; Principles, Protocols and Architecture*. Prentice-Hall International, Inc., USA, 2nd edition, 1991.

[4] L. Ferreira Pires. *Protocol implementation. Manual for the practical exercises 1997/1998*. Department of Computer Science. University of Twente. Enschede, the Netherlands, 1997.

[5] L. Ferreira Pires. *Protocol implementation. Lecture Notes 1997/1998*. Department of Computer Science. University of Twente. Enschede, the Netherlands, 1997.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design patterns: elements of re-usable object-oriented software*. Addison-Wesley Publishing Company, Inc., USA, 1995.

[7] R.E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10): 39-42, October 1997.

[8] P. W. King. Formalization of protocol engineering concepts. *IEEE Transactions on Computers*, 40(4):387–403, April 1991.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-oriented modelling and design*. Prentice-Hall, Inc., New Jersey, USA, 1991.

[10] D.C. Schmidt, M.E. Fayad. Lessons learned building reusable OO frameworks for distributed software. *Communications of the ACM*, 40(10): 85-87, October 1997.