From running code to event-graphs: a pragmatic approach to derive WCRT of protocol compositions*

João Carlos Rodrigues
INETI
joao.carlos@ineti.pt

Luis Rodrigues
U. de Lisboa
ler@di.fc.ul.pt

July 19, 2004

Abstract

A flexible way of building modular communication stacks relies on the use of protocol composition. In order to derive the worst-case response time of a protocol composition, one needs to capture its event-graph: the event-graph consists of the set of all events processed by each component and how these events are related.

This paper describes a protocol composition framework that simplifies the task of deriving the event-graph from the protocol implementation. The framework, called *RT-Appia*, takes a pragmatic approach: instead of requiring the use of domain specific code analysis tools, or dedicated compilers, it simply requires protocol programmers to make explicit which events are processed and produced by each layer, and how these events are related. The interest of the approach is that the same data structures are used not only to simplify the task of computing the worst case response time of the protocol composition, but also to optimize the performance and for debugging the resulting implementation.

*Sections of this report will be published in the Proceedings of the 5th IEEE International Workshop on Factory Communication Systems, Vienna, Austria, September 2004.

1 Introduction

With the increase of processing power and network bandwidth it is possible to build sophisticated distributed hard real-time systems. Many of these systems benefit from communication services that enforce strong consistency properties such as ordering and agreement at the communication level. The construction of such communication systems using the composition of several micro-protocol objects is an approach that has been applied with success in the non real-time arena [8, 4, 14]. This encourages the re-use of protocol components and allows the applications to configure stacks tailored to their needs. To benefit from this approach in hard real-time systems, one must be able to derive the timing behavior of a protocol composition.

In a previous paper [18], we have shown how the worst-case response time of a protocol composition (WCRT) can be computed from a *composition event-graph*. An event graph is a structure that captures all the events processed by each layer of a protocol composition. Furthermore, the event-graph also captures the causality among these events. These relations are useful as they can reduce the pessimism of the timing analysis. In particular, by identifying the offsets between related events, one can use the timing analysis approach developed by Tindell in [16] to compute the WCRT of a protocol composition.

One important practical problem is how to derive, in an automated form, the event-graph from the protocol implementation. In fact, we are interested in developing a tool that offers the programmer the possibility of coding each layer only once, in such a way that both the executable and the information required for timing analysis can be derived from the same source code. Furthermore, the programmer should not be required to learn a new language, or be required to annotate the source code with non-standard directives. Ideally, all the information required for the operation of the framework should be expressed in a programming language with a large user base, such as, for instance, C++.

This paper presents the *RT-Appia* protocol composition and execution framework. The framework has been implemented in C++ and supports the development of real-time communication stacks based on the composition of micro-protocols. Individual micro-protocols are described as software components that subscribe and produce events; interactions among adjacent protocols are modeled by the exchange of these events. When implementing a micro-protocol, the programmer provides information about the type

of events it is interested to process, and which events may be generated in response to those input events. Using this information, the tool is able to extract the sequences of events that occur in a given protocol composition. Therefore, without resorting to code analysis techniques, or to other form of pre-compilation, the framework can extract all the information required to derive the event-graph for the protocol composition. The event graph can later be used to feed the timing-analysis tool described in [18].

An interesting aspect of the framework is that the data structures required to derive the event-graph of a protocol composition are also used by the execution environment to optimize the performance of the implementation and make the code more robust by performing, upon demand, runtime checks on the events produced by each layer. The high degree of integration among the development, analysis, and execution environments greatly simplifies the task of the programmer of complex real-time protocol stacks. To illustrate the framework features, we use a group communication protocol designed for the Controller Area Network (CAN) field-bus [13].

The paper is organized as follows: Section 2 motivates our work and references related work. Section 3 introduces a case-study that is used to motivate the mechanisms implemented by *RT-Appia*. Section 4 shows how the event graph can be automatically derived from the protocol implementation by the *RT-Appia* framework. The merits and disadvantages of our approach are discussed in Section 5. Section 6 concludes the paper.

2 Motivation and Related Work

Among others, an important goal of a framework to support the development and execution of communication protocols is to simplify the task of programmers. This can be achieved by offering several complementary mechanisms and services that cover the different aspects of protocol development, including the design, the analysis, the implementation, and the execution of the protocols:

- At the design level, the framework should encourage the re-use of code, by allowing complex stacks to be created from the composition of micro-protocols. This aspect is particularly relevant in the context of real-time applications where, due to memory and power consumption constraints, it is interesting to execute in each node just the protocol layers

required to support the desired functionality. This facet of *RT-Appia* is discussed in Section 4.

- In the context of real-time systems, at the analysis level, the framework should provide support to validate the timing-behavior of the resulting protocol composition. In this paper, we focus our attention on the support provided by *RT-Appia* to simplify the task of validating the correctness of the protocol composition in the time domain.

- At the implementation level, the framework may (and usually does) offer a library exporting a number of functions needed by the protocol programmer, such as buffer management for data messages (including primitives to add and remove headers from messages), timer management, thread management, etc. In *RT-Appia* we also offer such functionality, adapted to the requirements of real-time operation.

- At the execution level, the framework must implement the basic services that support the execution of protocol modules, the communication and synchronization among those modules, etc. A real-time communication framework must also support the reservation of the resources required by a communication channel, to ensure their availability during the protocol execution. The *RT-Appia* not only supports pre-allocation of the objects managed by the communication protocols, but also offers mechanisms that allow to derive the size of the object pools in an automated manner.

x-Kernel [12] is an earlier and influential protocol composition framework that supports a composition model in which protocol stacks are structured as a hierarchical graph of protocols. Protocols communicate by exchanging messages. In order to promote the re-usability of protocols, messages are exchanged indirectly, via the *x*-Kernel, through calls to *push* and *pop* functions. Following the work of *x*-Kernel, many other protocol composition frameworks have been proposed, including Ensemble [8], CORDS [17], Coyote [4], Cactus [9], Bast [7], and *Appia* [14]. In the following paragraphs, we only devote our attention to the systems that have targeted real-time environments.

CORDS is an extension of the *x*-Kernel system that includes resource reservation mechanisms. Resources managed by CORDS include memory allocation, CPU cycles (the ability to select the number of threads and their

scheduling attributes) and network bandwidth. The "unit" of resource reservation is called a *path*. The implementation is supported by the real-time kernel OSF MK (derived from the Mach kernel [1]). The kernel's Integrated Time-Driven Scheduler (ITDS) allows the user to define several scheduling policies. Thread synchronization mechanisms implement the priority inheritance protocol to avoid priority inversion problems.

Cactus is a follow-up of Coyote, a system that augments the flexibility of x-Kernel by allowing each protocol to be decomposed in a set of micro-protocols. Each micro-protocol, in turn, is implemented as a set of event handlers that communicate primarily through the exchange of events but may also share memory. Cactus attempts to preserve the additional flexibility of Coyote while providing support for real-time operation. Cactus is built on top of CORDS, and each composition of micro-protocols, also called a *composite protocol*, is implemented as a CORDS coarse grain protocol. Cactus illustrates some of the advantages in customizing distributed real-time communication services through the composition of finer-grain micro-protocols. However, the worst-case execution time of a composite protocol may be difficult to estimate because of the need to know which events will be raised within each micro-protocol event handler. This information can be provided explicitly by the protocol designer or else it requires an additional tool to identify it. In [10], a conservative estimate is used and all the events raised are included in the computation even if they are raised in separate conditional branches of an event handler.

None of the previous frameworks provides support for the analysis of hard real-time communication systems, namely for the schedulability analysis of protocol compositions. In contrast, this kind of support is one of the important features of *RT-Appia*. Therefore, in this paper we describe the most important *RT-Appia* mechanism that simplify the integration of the schedulability analysis in the development process.

3 A Case-Study

3.1 A Simple Protocol Composition

To motivate our work, we use a very simple protocol composition consisting of a stack of just three layers as depicted in Figure 1. The bottom layer corresponds to the driver to the CAN network. The upper layer corresponds

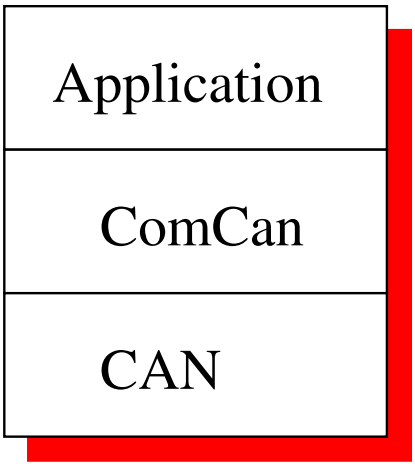

Figure 1: A simple stack with three layers.

to the application. The intermediate layer, "Committed Can" (*ComCan*) is a simple reliable broadcast protocol for CAN, which is, in essence, a simplified version of the protocols described in [15]. The algorithm executed at each of these layers is depicted in Figure 2 using pseudo-code. In the next paragraph we briefly describe the operation of this protocol composition.

Reliable multicast is enforced by a combination of the bare CAN properties, augmented by the ComCan algorithm. The rationale of the algorithm is the following. It was shown by Rufino *et al.* [15] that reliability of a broadcast in CAN is not ensured by the network if the sender of a message fails during the transmission. Therefore, the "ComCan" algorithm only delivers a message when it is sure that the sender did not crash during the transmission. This information is provided by the sender, through the transmission of a control message called the COMMIT (hence, the name of the protocol). The COMMIT is retransmitted by all nodes, to ensure the delivery of the original DATA message, in the event of the sender failure during the transmission of the COMMIT message. The reader may notice that this protocol is not very efficient (the protocol presented here has been simplified for sake of conciseness). A more efficient, but also more sophisticated, version of a reliable broadcast for CAN is described in [15].

It is also worth noting that the protocol description presented in Figure 2 is *not* fully modular, as each layer is aware of the adjacent layers (i.e., it

```
Application:
    upon event ⟨ init ⟩ do
        m := payload (); trigger ⟨ ComCan.TxDown, [ m ] ⟩;
    upon event ⟨ Application.TxUp, [ m ] ⟩ do
        // process message m

ComCan:
    upon event ⟨ ComCan.TxDown, [ m ] ⟩ do
        id = newid (); trigger ⟨ CAN.TxDown, [ data, ID, m ] ⟩;
    upon event ⟨ ComCan.TxCnf, [ data, ID, m ] ⟩ do
        trigger ⟨ CAN.TxDown, [ commit, ID ] ⟩;
    upon event ⟨ ComCan.TxUp, [ data, ID, m ] ⟩ do
        buffer: = buffer ∪ {[ data, ID, m ]};
        trigger ⟨ ComCan.comCanTimer, [ data, ID, m ] ⟩ with offset timer;
    upon event ⟨ ComCan.TxUp, [ commit, ID ] ⟩ do
        if {[ data, ID, m ]} ∈ buffer then
            buffer := buffer \ {[ data, ID, m ]};
            trigger ⟨ Application.TxUp, m ⟩;
            trigger ⟨ CAN.TxDown, [ commit, ID ] ⟩;
    upon event ⟨ ComCan.comCanTimer, [ data, ID, m ] ⟩ do
        if {[ data, ID, m ]} ∈ buffer then buffer := buffer \ {[ data, ID, m ]};

CAN:
    upon event ⟨ CAN.TxDown, packet ⟩ do
        //send packet to the CAN controller
    upon event controller confirms transmission of packet do
        trigger ⟨ ComCan.TxCnf, packet ⟩;
    upon event controller received message packet do
        trigger ⟨ ComCan.TxUp, packet ⟩;
```

Figure 2: Algorithms for the three layers.

explicitly triggers events of those layers). As it will be clear in the subsequent section, the composition model supported by *RT-Appia* eliminates this limitation.

3.2 The Composition Event-Graph

By looking at the algorithms of each layer, it is possible to manually construct the event-graph of the protocol composition. The event graph captures a causal of chain of event handlers execution, in response to some external stimulus (typically, the data load imposed on the system). Each node of the graph is identified by the name of the handler and the state carried by the event being processed (namely the message being processed).

The complete event-graph for our example is depicted in Figure 3. The graph can be constructed by capturing the sequence of events from the initial-

ization of the system. It can be seen that, upon initialization (event $E1$), the application triggers the transmission of a DATA message ($E2$). This transmission request will trigger a chain of events in the remaining layers, both in the sending node and in remote nodes. We just describe in detail the first steps of this chain, as the same procedure can be applied to derive the rest. The transmission request triggered by the *Application* layer is processed by the *ComCan* layer, which manipulates the message by adding a control header. Then, the *ComCan* layer forwards the transmission request to the *CAN* layer ($E3$). In turn, the *CAN* layer invokes the CAN controller to transmit the message on the network. This will later trigger a local confirmation ($E4$) and an indication ($E5$) at all nodes (we are describing a broadcast protocol). These two events, in turn, generate further events, as illustrated in the remainder of the Figure. Note that in the event graph, events that carry a DATA message are labelled with the type of message associated with the event. For instance, for the purpose of timing analysis, events $E10$ and $E13$ are different events.

3.3 Using the Event-Graph to Perform the Timing Analysis

In a previous paper [18], we have shown how the event graph can be used to derive the WCRT of the protocol composition. For self-containment, and before we explain how the event-graph can be automatically derived from the code, we briefly highlight the main principles of the technique. The interested reader can refer to [18] for further details.

The WCRT for each received event of the protocol is computed according to a set of timing analysis equations developed by Tindell in [16]. In his work, several schedulability analysis models are presented; we have chosen to use the timing offset model as this is the one that better allows us to capture the chain of events mechanism. This method defines a *transaction* as a set of tasks that execute with given offsets in relation to its initial time. We use this to offset the various tasks that make up a micro-protocol in relation to the user request event. The WCRT of an event is the time taken by the protocol starting at the time of reception of the event that triggered the protocol until it returns to an idle state. This time is composed of the local worst case computation time (WCCT) and the WCRT of the lower level layers (including lower levels computation time and message transmission through

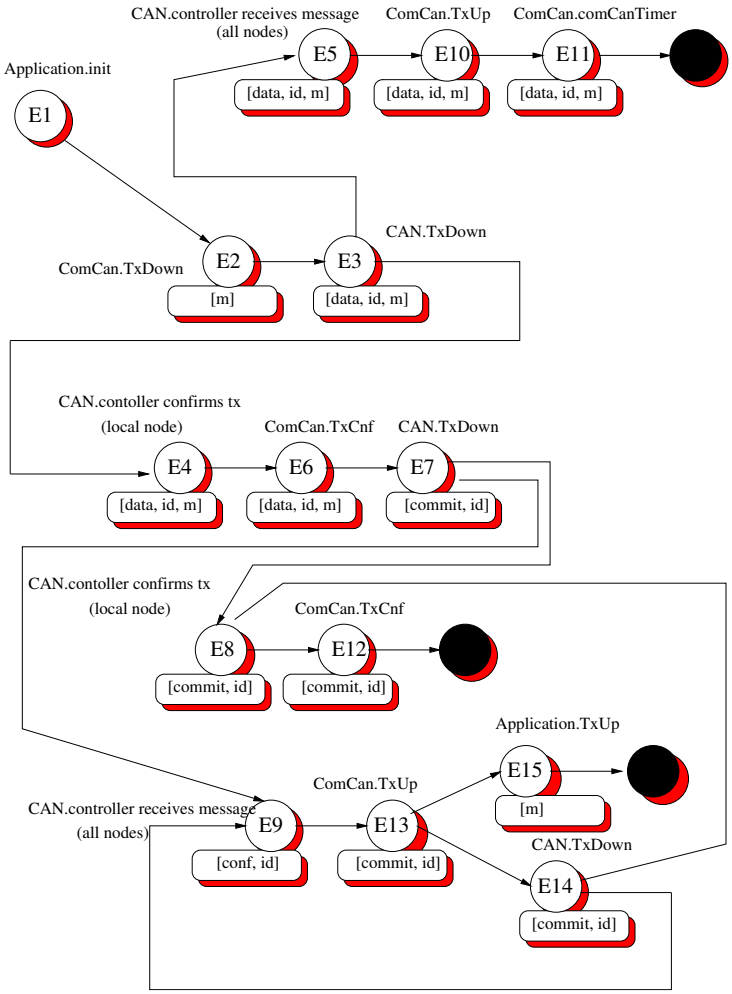

Figure 3: Protocol composition event graph.

the communication channel).

The timing analysis of the communications protocols is performed in two phases. In the first phase, the WCRT of the processing spent by the protocols themselves is computed. In the second phase, the time spent during messages transmission through the communications network must also be calculated. However, the results of the second phase depend on, and affect, the results of the first phase, requiring several iterations before converging on a final value.

4 Deriving the Event-Graph Automatically from Code

We now explain how the *RT-Appia* framework allows the programmer to code the protocol stack in such a way that both a running implementation and the event graph can be derived from the same source. In order to do so, we introduce the relevant *RT-Appia* mechanisms. Later in the text, we make a brief overview of the remaining functionalities of *RT-Appia* which are not directly related with the capture of the event-graph but are nevertheless worthwhile mentioning to provide a clearer view of the overall framework. *RT-Appia* has been implemented in Visual C++ and the resulting prototype runs both on the WindowsNT operating system and on the ETS kernel[1].

4.1 Channels, Layers and Sessions

We start by introducing the composition mechanisms of *RT-Appia*. Many of these mechanisms are inherited from the *Appia* system [14], a protocol composition framework developed in Java, without support for real-time operation, upon which *RT-Appia* was based.

In *RT-Appia* each stack is composed of one or more *channels*. Each real-time channel is an ordered sequence of *sessions*, instances of a specific *protocol layer*. The session maintains state that is used by the layer to process events. A layer that implements an ordering protocol may keep a sequence number as part of the session state. In connection oriented protocols, the session also keeps information about the endpoints of the connection. The sequence of layers associated with a given channel defines a *configuration* implemented by the channel. An important aspect of an *RT-Appia* stack is that

[1]ETS is a proprietary kernel of Phar Lap Software, Inc.

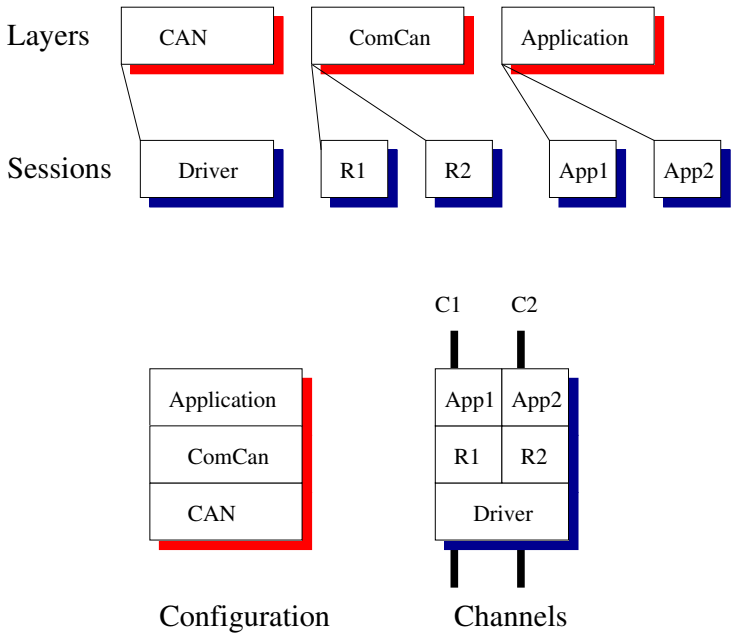

Figure 4: Configuration, layers and sessions.

different channels may share sessions at one or more layers. This mechanism supports the implementation of inter-channel coordination policies. The relation among configurations, layers and session in *RT-Appia* is illustrated in Figure 4.

Communication between sessions is made by the exchange of *events*. Events are object oriented data structures, all descendant of a base class named RTEvent. New events can be created by deriving from a previously defined event class. In order to allow future event refinement, event type tests are always performed on the weakest class satisfying the desired requisites. The goal is to support event specialization using inheritance. This way, legacy protocols, unaware of the new event attributes, will continue to execute correctly. An advantage of *RT-Appia* is the use of an open event model that allows the protocol designer to define the more appropriate set of events to a target application area. Thus it provides an excellent framework to develop specialized and efficient protocol stacks to be used in real-time applications.

Listing 1: Methods accepts and provides.

```
class RTLayer {
public:
  //....
  void accepts (RTEventClass *event_class, Direction dir,
                RTHandler handler, Duration wcct);
  void provides (RTEventClass *event_class, Direction dir);
  //....
};
```

4.2 Events Accepted and Provided

In order to derive the event graph for a protocol composition, one need first to identify which events are processed and generated by each layer. Therefore, *RT-Appia* requires the programmer of a layer to specify the set of events accepted by the layer, and the set of events produced by the layer. For that purpose, the programmer must invoke the accepts and provides methods, whose signature is depicted in Listing 1. These methods must be invoked in the constructor of the object that represents the protocol layer in the *RT-Appia* runtime support (this object is a singleton).

The parameters for the accepts method are as follows. The first parameter is the class of the event being accepted. The *RT-Appia* framework requires the declaration of events to follow a methodology that offers the required reflexive information to support both the capture of event-graphs but also the scheduling of events at runtime. Therefore, each class of events has a singleton object that represents that class [6]. The second parameter is the direction in which the event flows in the stack. The third parameter, is a pointer to the session method that will process the event. Finally, the last parameter is the worst case computation time of the handler, which, in the current version of the framework, must be explicitly declared by the programmer. The provides method only accepts the class of the event and its direction.

Listing 2 shows the declaration of accepted and provided events for our case study (for the *Application* and *ComCan* layers). There are a number of important points that are worth to emphasize. The reader will notice that the same event, of class dataTxEventClass flows in the stack in both directions. Such an event is produced by the *Application* in the down direction. The event is then processed by the *ComCan* layer and forwarded to the *CAN* layer. When the event is received from the network, it preserves its class but

Listing 2: Accepted and provided events.

```
ApplicationLayer:: ApplicationLayer ()
{
  accepts (dataTxEventClass, UP,
           &ApplicationSession::handleDataTxUp, WCCT);
  accepts (initEventClass, UP,
           &ApplicationSession::handleInit, WCCT);

  provides (dataTxEventClass, DOWN);
  //...
}

ComCanLayer::ComCanLayer ()
{
  accepts (dataTxEventClass, DOWN,
           &ComCanSession::handleDataTxDown, WCCT);
  accepts (dataTxEventClass, UP,
           &ComCanSession::handleDataTxUp, WCCT);
  accepts (commitEventClass, UP,
           &ComCanSession::handleCommitUp, WCCT);
  accepts (dataCnfEventClass, UP,
           &ComCanSession::handleDataCnfUp, WCCT);
  accepts (comCanTimerEventClass, DOWN,
           &ComCanSession::handleTimer, WCCT);

  provides (dataTxEventClass, DOWN);
  provides (commitEventClass, DOWN);
  provides (dataTxEventClass, UP);
  provides (comCanTimerEventClass, DOWN);
  //...
}
```

is now propagated in the upwards direction: it is first processed by the *CAN* layer, forwarded by the *ComCan* layer and consumed by the *Application*. Using this pattern, the programmer of each layer does not need to be aware of exactly which adjacent layers will be used in the final protocol composition. This make easy to add or remove layers to adapt the protocol stack to specific requirements.

On the other hand, using the information maintained in the RTChannel, the runtime can check exactly which sessions accept each event. With this information, *RT-Appia* constructs an *event route*, an ordered sequence of sessions that process an event. The event route is used at runtime to avoid delivering an event to layers that are not interested in processing that event. Therefore, the information captured using the accepts and provides methods is useful, not only for the timing analysis, but also to support an efficient execution of the protocol composition.

We would also like to highlight how the composition model of *RT-Appia* allows to express the fact that the *ComCan* protocol processes two different types of messages: DATA messages and COMMIT messages. This is achieved by deriving a specialization name CommitEvent from the DataTxEvent. Using this technique, we can distinguish both events at the level of the *ComCan* layer (associating different handlers to each event) without changing a single line of code at the level of the *CAN* layer (which processes both messages in the same way, by handling just the base class). This feature, supports the reuse of protocol implementations in different contexts, an important property of protocol composition frameworks.

4.3 Event Triggers

The knowledge of the set of accepted and provided events is not enough to construct the event graph. In fact, we also need to capture the causal relation between accepted events and provided events. In order to make this relation explicit, *RT-Appia* requires the programmer to declare *triggers*. A trigger indicates that a given provided event is generated in response to an associated accepted event. The signature of the available triggers is presented in Listing 3.

There are two types of triggers: local triggers and network triggers. Local triggers capture the communication between sessions in a protocol stack in a single node. Network triggers capture the communication among different nodes using a given network. The parameters for both triggers are the same, with a few exceptions described later.

The first parameter, source, is a set of nodes where the trigger takes effect. Most triggers take effect at all nodes (a reserved value, ALLNODES, is used for this purpose). However, this parameter offers the opportunity to specify that some triggers may only be active at given nodes. As we will see later with our example, this feature is particularly useful to model the load of the system (since not every node may be allowed to transmit messages).

The next four parameters are the class and direction of an accepted event and the class and direction of the provided event, respectively. This information is the key element of a trigger, as it allows to associate an accepted event with a provided event. The FORWARD keyword is used to specify that the event provided is the same as the event accepted. In this case, the class of the event is preserved.

Listing 3: Trigger methods

```
const RTEventClass *FORWARD = NULL;
const NodeSet *ALLNODES = NULL;
const int PERIODIC = ~0;
enum TriggerSemantics { Default, OnFirst, OnLast }
enum TriggerLocation { Local, Remote }
enum IdOperation { NewId, SameId, AddSubId, RemoveSubId }
enum SizeOperation { NewSz, FromTrigerId, FromProvidedId }

class RTLayer {
public:
  // ....
  localTrigger        (NodeSet *source,
                       RTEventClass *accepted, Direction acc_dir,
                       RTEventClass *provided, Direction prov_dir,
                       IdOperation idop, int id_desc,
                       SizeOperation size_op, int size,
                       TriggerSemantics semantics,
                       Duration period, int max_activations);

  networkTrigger      (NodeSet *source,
                       RTEventClass *accepted, Direction acc_dir,
                       RTEventClass *provided, Direction prov_dir,
                       IdOperation idop, int id_desc,
                       SizeOperation size_op, int size,
                       TriggerSemantics semantics,
                       NetworkModel *network, TriggerLocation loc);
};
```

The remaining set of parameters are less obvious, but extremely important to preserve the modularity of the system.

The IdOperation is used to distinguish distinct triggers of the same handler in the context of a single chain of events. Consider the case of the *ComCan* layer in our example. This layer, in response to a transmission request, activates twice the data transmission handler of the underlying CAN layer: the first time to transmit the DATA message, and a second time to transmit the corresponding COMMIT message. These invocations must be distinguished in the event graph. The IdOperation parameter addresses this problem. When a trigger is activated it is possible to:

- Create a new instance root identifier (NewId). A root identifier is a 4-tuple that consists of: the node identifier, the layer where the identifier was generated, the class of the triggered event and, finally, a numerical id provided by the programmer (id_desc parameter).
- Preserve the identifier of the accepted event (SameID). This is the most common behavior of an handler that forwards an event after some processing.
- Add a sub-identifier to the identifier of the accepted event (AddSubId). This is used by a layer that generates more than one event in response to a single accepted event (for instance, our *ComCan* layer or a layer that fragments a message). The programmer specifies a numerical value (id_desc parameter) that distinguishes each sub-identifier. Sub-identifiers are stacked on top of the root identifier.
- Remove a sub-identifier (RemoveSubId). This operation is the counterpart of the AddSubId operation. A layer that has previously added a sub-identifier to an event may extract the original identifier. This is useful, for instance, to model the reassembly of an original message from multiple fragments.

The SizeOperation parameter allows the programmer to specify the size of the messages created by the layer. This information is important to have a fine grain timing analysis of the protocol composition, as the message size has an impact on the network transmission. The programmer may specify three different operations: to declare the absolute size of a new message (NewSz), to increase the size of a incoming message (this operation, FromTriggerId, models

the addition of headers to messages), to recover the size associated with some message identifier (this operation, ProvidedId, models the operations such as the reassembly of messages).

The TriggerSemantics parameter captures the fact that, in many protocols, triggers may fire only once for a given message, even if the accepted event is accepted more than once. Consider for instance a layer that retransmits a message but discard duplicates at the reception. Even if a message is received more than once, only one copy is forwarded to the application. In a similar manner, a layer that reassembles a message, may receive several fragments and only delivers a single message upwards, when all fragments have been received. The three semantics currently supported by *RT-Appia* model these different cases. The Default semantics, activates the trigger every time the accepted event is received. The OnFirst semantics, for the same message identifier, discard all duplicate activations of the trigger. The OnLast semantics, for a given message identifier, only considers the last activation (triggers with on last semantics are only evaluated when no more triggers with Default or OnFirst semantics can be activated).

The final parameters distinguish local triggers from network triggers. In a local trigger the programmer may specify a period and a maximum number of activations. This information is used to model timers and periodic workloads in the timing analysis. In the network trigger, the programmer specifies a *network model* object. A network model is a component that captures the timing properties of the real-time network in use. In our prototype, a model of the CAN bus network has been implemented. In a network trigger, one also specifies if the event target is the local endpoint of a channel (for instance, a transmission confirmation) or the remote endpoint(s) (typically, data indications).

Listings 4, 5, and 6 present all triggers defined for our case study. These listings illustrate how the different parameters of the triggers can be used.

The trigger for the application layer, in Listing 4, illustrates how a new message is created. In this case, the message is sent in response to the init event (this event is automatically generated by the *RT-Appia* runtime). Note that the message is only sent by Node1, that a new identifier is created, and that the size of the message is specified in the event trigger. Note also that a periodic workload could be captured by specifying a period for this trigger (we opted to omit this behavior in the example to simplify the graphs).

The triggers for the *ComCan* layer, partially depicted in Listing 5, show how COMMIT messages can be matched with the associated DATA messages

Listing 4: Application layer triggers

```
NodeSet* data_sources = new NodeSet (Node1);

ApplicationLayer:: ApplicationLayer ()
{
  //...
  // start transmission after init event
  localTrigger   (data_sources,          // on node 1
                  initEventClass, UP,    // trigger event
                  dataTxEventClass, DOWN, // created event
                  NewId, 1,              // with new identifier
                  NewSz, PAYLOAD_SZ,     // with new size
                  Default,               // default semantics
                  0, 0);                 // not periodic
}
```

Listing 5: ComCan layer triggers (partial)

```
ComCanLayer::ComCanLayer ()
{
  //...
  // send commit message
  localTrigger   (
      ALLNODES,                  // trigger at every node
      dataCnfEventClass, UP,     // trigger event
      commitEventClass, DOWN,    // created event
      AddSubId, 2,               // same identifier as tx confirmation
      NewSz, CommitMessageSize,  // commit message has fixed size
      OnFirst,                   // sends the commit only once
      0, 0);                     // not periodic
  // forward data message to the application when commit is received
  localTrigger   (
      ALLNODES,                  // trigger at every node
      commitEventClass, UP,      // trigger event
      DataTxEventClass, UP,      // created event
      RemoveSubId, 0,            // same identifier as tx request
      FromProvidedId, 0,         // size of original data message
      OnFirst,                   // discards duplicates
      0, 0);                     // not periodic
  // retransmit the commit message
  localTrigger   (
      ALLNODES,                  // trigger at every node
      commitEventClass, UP,      // trigger event
      commitEventClass, DOWN,    // created event
      SameId, 0,                 // same identifier as confirmation request
      NewSz, CommitMessageSize,  // commit message has fixed size
      OnFirst,                   // sends the commit only once
      0, 0);                     // not periodic
}
```

Listing 6: CAN layer triggers

```
CANLayer:: CANLayer ()
{
  //...
  // deliver the message to the local node (loopback)
  networkTrigger (
      ALLNODES,              // trigger at every node
      dataTxEventClass, DOWN, // trigger event
      FORWARD, UP,           // created event
      SameId, 0,             // same identifier as request
      FromTriggerId, 0,      // not relevant
      Default,               // default semantics
      canModel, Local);      // uses the CAN network model to derive offsets
  // deliver the message to the remote nodes
  networkTrigger (
      ALLNODES,              // trigger at every node
      dataTxEventClass, DOWN, // trigger event
      FORWARD, UP,           // created event
      SameId, 0,             // same identifier as request
      FromTriggerId, 0,      // same size as request
      Default,               // default semantics
      canModel, Remote);     // uses the CAN network model to derive offsets
  // confirm the transmission of the data message
  networkTrigger (
      ALLNODES,              // trigger at every node
      dataTxEventClass, DOWN, // trigger event
      dataCnfClass, UP,      // created event
      SameId, 0,             // same identifir as data request
      NewSz, 0,              // not relevant
      Default,               // default semantics
      canModel, Local);      // uses the CAN network model to derive offsets
}
```

through the use of sub-identifiers. For instance, when a COMMIT message is sent, it inherits the identifier of the original DATA message, and a sub-identifier is added to distinguish it from the original request. Later, when a data indication (dataTxEventClass, UP) is generated in response to the reception of the COMMIT message, the sub-identifier is removed and the original size of the corresponding DATA message is recovered using the FromProvidedId as the value for the SizeOperation parameter. The same example, illustrates the use of the TriggerSemantics parameter. By selecting the OnFirst value, one eliminates the duplicate generation of redundant COMMIT events and data indications.

Finally, the triggers for the CAN layer illustrate the use of the networkTrigger. The canModel object is used by the timing analysis tool to derive the offsets of the local confirmation and remote indication(s), in response to a

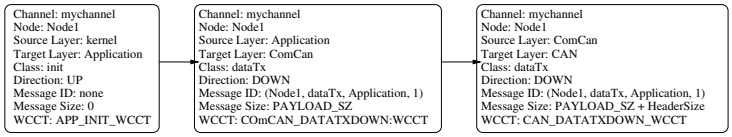

Figure 5: Event descriptor graph (first 3 nodes).

data transmission request to the CAN network.

Triggers were introduced in *RT-Appia* to derive the event graph in an automated manner. However, they are also used to support the debugging and validation of the protocol execution. In fact, it is possible to configure the implementation such that the generation of events at runtime is confronted with the specification of triggers. If an handler provides an event not captured by the trigger information, a runtime exception is generated.

4.4 From Triggers to the Event Graph

The event graph is constructed in a iterative manner, by generating the init event at all nodes and checking for triggers. If a trigger is fired, new events are added to the graph and triggers are checked for these events. This procedure is iterated until no new events are created.

Each node of the graph is an event descriptor. The descriptor contains the following information: The channel in which the event is generated; the node in which the event is triggered; the layer that has triggered the event (source); the layer that will accept the event (target); the event class, the event direction; the identifier of the message associated with the event, if any; the size of the message associated with the event if any; and the WCCT of the associated handler.

Figure 5 illustrates the first three nodes of the event graph, rooted at the init event triggered at Node1 as extracted from the triggers coded in the layers. The reader may check that a complete event graph, equivalent to the one we have derived manually and depicted in Figure 3 can be derived in an automatic manner using this procedure.

5 Discussion

The version of *RT-Appia* described in this paper is implemented in Visual C++. The prototype runs both on the WindowsNT operating system and on the ETS kernel. The current version can be executed on embedded systems with a 32-bit x86 compatible processors.

One of the most challenging aspects of the *RT-Appia* design was the definition of the trigger mechanisms, in order to automate the extraction of the event graph. We aimed at achieving the following goals:

- *Preserve modularity,* in the sense that, when a layer is specified, the programmer should not be aware of configuration parameters such as: the layers that will be executed above and below, the size of the messages being exchanged, the exact subclasses of the events accepted (he/she should only be aware of its base class), etc.

- *Support a wide range of protocols,* including protocols that fragment and reassemble messages, protocols that generate control messages associated with DATA messages (such as acknowledgements or commit messages, as illustrated by the *ComCan* algorithm), protocols that retransmit messages a maximum number of times (bounded loops), protocols that eliminate duplicates, etc.

We believe that these goals have been meet. It should be noted however that the expressive power of trigger declarations is limited and cannot capture the behavior of all protocols. For instance, multi-participant protocols with a rotating coordinator, where the decision of which node reacts to a given broadcast message depends on the session state, cannot be expressed with our trigger mechanisms. However, to extract the model from the actual code in an automated manner is a very hard task. Furthermore, it should be noted that our goal is not to perform full protocol validation, but only to capture worst-case execution patterns. This broadens the applicability of our tool, despite its theoretical limitations (in most cases, the lack of detail only introduces an acceptable level of pessimism in the analysis). Therefore, we believe that *RT-Appia* provides a valid pragmatic tradeoff between complexity and power.

One concern regarding our approach is that the programmer is required to perform an additional task (to specify the triggers) in order to allow the event graph to be extracted. However, it is important to stress that most of state-of-the-art tools for automatically validating protocol properties, such as

Kronos [19] and UPPAAL [2], require a model of the program to be extracted manually. In fact, tools to derive the model automatically from code are a topic of active research [3, 5, 11]. Therefore, *RT-Appia* imposes a burden on the programmer that is comparable to other existing approaches.

At first sight, it may seem that the specification of the triggers is almost as hard as the extraction of the complete event graph (as the construction of the graph from the triggers is somehow trivial). However, we need to reiterate that the specification of triggers is local to a layer, i.e., when the programmer specifies the triggers she does not need to be aware of the other layers in the stack. One of the challenges of the work described here was to design a trigger interface able to support such locality. The fact that such interface was feasible was not obvious when we started this work. Our work shows that it is possible to achieve this goal: For instance, in our example, the triggers for the CAN layer are specified for DATA messages only; this does not prevent upper layers from defining new subclasses, such as the COMMIT message (and the framework is able to support the exchange and keep track of those new messages).

Finally, we discuss how *RT-Appia* helps to keep the information captured by the triggers synchronized with the actual code of the event handlers. As we have noted before, one advantages of the *RT-Appia* approach is that the trigger information is also used at runtime. In particular, *RT-Appia* automatically builds a set of control data-structures that can be used at runtime to check the conformity of the execution with the information captured by the triggers. For performance reasons, these runtime checks are optional but, when activated, generate exceptions whenever a mismatch is detected (for instance, if a handler generates an event not listed in any trigger). Therefore, *RT-Appia* offers a much tighter integration among the model and the implementation, specially if compared with tools where the model is described in a language different from the programming language (such as Kronos and UPPAAL).

6 Conclusions

In this paper we have described a system that simplifies the task of performing the timing analysis of real-time protocol compositions by extracting the *event graph* automatically from the protocol implementation. This approach has the advantage of requiring only a compiler for the language used in the

implementation (in the current prototype, C++). More sophisticated tools, such as the ones based on code analysis, may be much harder to develop and maintain. Furthermore, the programmer is not required to annotate the code with any kind of pre-compilation directives or comments. Therefore, our tool is not only simpler to use, but also efficient in coding effort. Finally, the information provided to support the extraction of the event graph is used by the implementation to optimize the execution of the protocol composition, and to perform a number of runtime checks that validate the correspondence between the computed event-graph and the actual flow of the running implementation.

Acknowledgements We are grateful to A. Casimiro, A. Lopes and J. Rufino and the anonymous reviewers for commenting earlier versions of this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for Unix development. In *Proceedings of the Summer Usenix*, July 1986.

[2] T. Amnell, G. Behrmann, J. Bengtsson, P. DArgenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. Larsen, M. Moller, P. Pettersson, C. Weise, and W. Yi. Uppaal - now, next, and future, modeling and verification of parallel processes. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *LNCS 2067*, pages 100–125. Springer Verlag, 2001.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, pages 103–122. LNCS 2057, Springer Verlag, May 2001.

[4] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, Nov. 1998.

[5] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, R. V. Willem, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

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

[7] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proc. of the 18th Intl. Conference on Distributed Computing Systems*

(ICDCS-18), pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[8] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[9] M. Hiltunen, X. Han, and R. Schlichting. Real-time issues in cactus. Technical Report AZ85721, Department of Computer Science, University of Arizona, Tucson, 1996.

[10] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing qos attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, June 1999.

[11] G. Holzmann and M. Smith. An automated verification method for distributed systems software based on model extraction. *Trans. on Software Engineering*, 28(4):364–377, Apr. 2002.

[12] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engeneering*, 17(1):64–76, Jan. 1991.

[13] ISO, editor. *ISO International Standard 11898 - Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for high-speed communication*. ISO, nov 1993.

[14] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, Apr. 2001. IEEE.

[15] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Digest of Papers, The 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 150–159, Munich, Germany, June 1998. IEEE.

[16] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS221, Department of Computer Science, University of York, Jan. 1994.

[17] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the 2nd IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Laguna Beach, CA, Feb. 1996.

[18] J. Ventura, J. Rodrigues, and L. Rodrigues. Response time analysis of composable micro-protocols. In *Proceedings of the 4rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 335–342, Magdeburg, Germany, May 2001. IEEE.

[19] S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, (1/2), 1997.