SOFTWARE DEVELOPMENT AND VERIFICATION OF DYNAMIC REAL-TIME DISTRIBUTED SYSTEMS BASED ON THE RADIO BROADCAST PARADIGM

JAN VAN KATWIJK, RUUD DE ROOIJ, SYLVIA STUURMAN AND HANS TOETENEL*

Abstract. The combination of an increased power of computer systems and a marriage between computing and communication causes an enormous increase in the complexity of applications in almost all domains. This also applies in the real-time and the embedded domains, construction of distributed applications is a major research area. In our research, we develop a framework for the systematic development of distributed real-time control applications. We emphasize the use of pragmatic sound approaches in the design steps of the development process, preferably based on some common architectural style. For analyzing and validating critical elements of design and implementation, we emphasize the use of formalisms, however. Complexity of applications is such that for real verification and validation, proof or model-checking techniques are required. We use formalized abstraction as a technique for obtaining the appropriate templates from design and implementation. These abstractions are then dealt with in a model-checking and analysis tool set. In this paper we show some elements of our approach, in particular we describe the architectural style that we are using, the Radio Broadcast Paradigm, and we demonstrate the viability of our approach by showing a case study.

Key words. distributed computations, real-time computing, software verification, radio broadcast paradigm

1. Introduction. The continuously increasing power of cheap computers and the integration between computer systems and communication mechanisms both have an enormous impact on applications. Applications become more and more distributed, obviously in the areas of business computing, but also in technical and control domains.

From our perspective, measurement and control applications are interesting since they depend, perhaps stronger than other kinds of applications, on the notion of time and on the reliability of the underlying networked systems. In distributed environments, these notions are hard to control, which influences design and implementation of distributed control applications considerably.

Part of the additional care to be given to the design of distributed control applications is based on the uncertainty in the stability of the environment: message passing in a network may not take a constant amount of time, networks may break down or be altered dynamically. Development of an application therefore should take into account an accurate characterization of the execution attributes of the underlying infrastructure. Characterization of the underlying infrastructure, and indicating requirements on the infrastructure is therefore an integrated element in the development cycle of this kind of applications.

The objective of our research is to establish a framework for the systematic development of distributed real-time control applications. In our approach, practical development starts with the selection of an architectural style and an architecture that will fit the needs of the application. The selected architecture is supported by technology implementing the functionality of the architectural components. Such designs are gradually transformed into program code through the coding phase of system development.

The design process is supported by design analysis and simulation feedback. Analysis provides data that enables a more precise verification effort to be done. A classical example in this respect is the validation of timing properties of a system. Standard schedulability analysis provides a technique for analyzing whether or not real-time requirements are met, under the assumption that accurate execution characteristics are available. Here testing and simulation can provide accurate timing data that together with an abstract representation of the process structure of the implementation form the basis for formal analysis and simulation.

In our approach, we use abstraction as a technique for deriving the appropriate templates from design and implementation as an input to formal verification (see Figure 1.1).

In order for our approach to be successful, we need to be able to build models of the technology supporting the architectural models, to be used in validation of the architectural instances.

Analysis and verification can be addressed two ways. The classical way is that testing and analysis of implementations provide data on execution characteristics, after which the application

* Department of Mathematics and Informatics, Delft University of Technology, P.O.Box 356, 2600 AJ Delft, The Netherlands. E-mail: {J.vanKatwijk|R.DeRooij|S.Stuurman|W.J.Toetenel}@twi.tudelft.nl

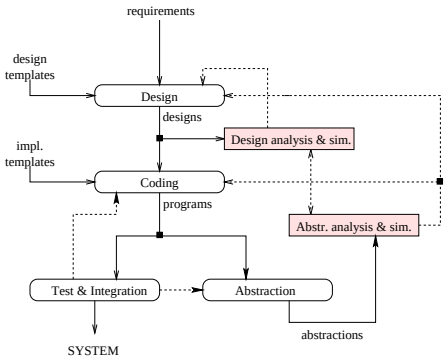

FIG. 1.1. *A Software Development Process*

is analyzed using this data. For many distributed applications this approach is not appropriate, simply because the application runs on different underlying systems with different characteristics. However, an approach in which constraints on the architectural support mechanisms are derived, would be desirable. In such an approach, analysis of the application leads to a characterization of essential constraints in the underlying system that, when met, guarantee the application to operate correctly.

One of the key issues in our research is therefore to build models for both applications and underlying frameworks with which a qualitative and quantitative analysis of execution parameters and attributes in distributed real-time control systems is possible. In order to limit the scope of our research we use Java as our language of choice, and we use a radio broadcast model as basis for interprocess communication. For the latter, we developed a standardized implementation and a number of specification and analysis tools. Part of the research then focuses on the characterization of the radio broadcast model implementation with respect to applications running on it.

In this paper, we describe some elements of our development approach and discuss a (simplified) example of the analysis we pursue. The architecture we have chosen is the Radio Broadcast Paradigm (RBP), while the analysis technique used is parametric model checking. In Section 2 we describe the RBP and briefly discuss its implementation in Java. In Section 3 we describe a case study, a distributed control application. In Section 4 we describe our analysis approach and the results of the application to a simplified model of the distributed control application. Finally, in Section 5, we summarize our results and describe our current work.

2. RBP and Its Implementation.

2.1. Subscription-based Communication. A subscription-based communication model abstracts from the physical communications structure by introducing communications "channels". A process interested in specific information "subscribes" to a channel. The real-world analogy to this action is tuning a radio to a certain frequency. After having subscribed to the channel, the process will receive all messages sent to the channel. Similarly, if a process wants to make information available to other processes, it can "publish" it on a channel, either an already existing one or a newly created one. In principle, the number of available channels is unlimited.

The properties described above make the subscription-based model appropriate for (data--dominated) embedded systems, which deal with a continuous information stream from the environment [3].

In most cases, the subscribers will be able to deal completely locally with the (temporary or permanent) absence of incoming messages. The fact that there are no hard-coded dependencies between processes opens the possibility for on-line change. Processes can be added, removed, and/or replaced without disturbing the information flow. The subscription paradigm thus provides a mechanism for very late binding.

2.2. Existing Models and Implementations. The subscription paradigm is a form of multicast communication, for which there are many implementations [5]. In this section, we briefly

mention a number of existing implementations.

2.2.1. Splice. SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines) is an implementation of the paradigm developed by Boasson [3] of Hollandse Signaal Apparaten. The main application area is embedded systems for command and control systems.

SPLICE extends the basic subscription-based communication model as described above, by providing a conceptual global data space. This data space is partitioned into data sorts (channels). Data sorts are typed; the structure of data elements (their fields and field types) must be specified before the data sort is used. In addition, it is possible to mark fields of the data sort as key fields. The global data space is a virtual one, it does not physically exist.

SPLICE provides a rich query mechanism so that applications can retrieve exactly the data they are interested in. In addition to queries, processes can put a filter on a local database. Whereas queries operate on the information already present in the local database, a filter is used to decide whether a newly received data item is stored in the database. Using filters enables a process to specify a more fine-grained selection of the information it wishes to process, and reduces the size of the local database.

SPLICE does not provide an operation to remove a data item from the global data space. Processes are allowed to remove items from their own local databases if they decide the specific data item is no longer needed, but a global remove-operation is not available.

2.2.2. JavaSpaces. JavaSpaces is a package of Java classes and interfaces, very recently developed by Sun Microsystems [12]. The package is meant to support the design of distributed applications using Java. At the moment of writing this article, only a beta version is available.

The two main concepts of JavaSpaces are "space" and "entry". The term "space" is used for the implementation of a JavaSpaces server. A space holds entries: typed groups of objects, expressed in a Java class. Three (atomic) operations are possible on entries:

- An entry may be *written* into a space. The result of this operation is a copy of the entry object, created in the space.
- An entry may be *read*, after which the space still holds the entry.
- An entry may be *taken*, after which the space no longer holds the entry.

With respect to the *read* and the *take* operations, JavaSpaces offers the possibility of matching against a template: entries in the space may be matched against an object that has some or all of its fields set to specific values (the template). The remaining fields act as wild-cards.

JavaSpaces offers still another kind of connection between clients of a space (or JavaSpaces server) and the space: clients may request a server to *notify* them, when an entry, matching a specific template, is written. Notification is done using a distributed event mechanism. This notify operation makes it possible to use the JavaSpaces package as an implementation of the subscription-based communication model.

Notifications may arrive in different orders on different clients, or may not arrive at all. The JavaSpaces server will make a "best effort" attempt to deliver the notification. The server will retry at least until the notifications request "lease" is expired. This "lease" is an amount of time, negotiated between the server and the client, during the request for notification. This concept of a lease is added to the JavaSpaces package to obtain more robustness, in case of failure of parts of a distributed application.

2.2.3. Other Implementations. A real-time publisher/subscriber model [13] was developed at Carnegie-Mellon as a basis for building replaceable software units. The emphasis is on real-time fault-tolerant systems. Next to the basic publish/subscribe model, a facility is provided to detect whether nodes in the network have failed, and when new nodes in the network have joined. The channels are therefore consistent to all publishers, i.e., all publishers on one channel will see the same subscribers at every time.

The Tibco Rendezvous Information Bus [15] is a commercial implementation of the subscription-based communication model. It is mainly intended for financial applications, such as for example the distribution of stock price information to dealer rooms. One of the interesting features is that when a process subscribes, it can leave part of the channel name unspecified. It will effectively be subscribed to all channels matching the given channel name.

The Java Shared Data Toolkit, a product of Sun Microsystems, is intended for collaborative interactive applications. It provides the concept of sessions which processes can join and leave dynamically. Within a session, processes can send messages to the other processes, as well as have the ability to create shared data structures. The JSDT can be used on top of a number of different communication mechanisms, such as sockets or Remote Method Invocation. Sun Microsystems offers JSDT and JavaSpaces as separate products intended for use in different application areas.

2.3. A Java Implementation of the Radio Broadcast Paradigm. In order to have a facility for doing experiments with the RBP and its implementation, we developed a library implementing the Radio Broadcast Paradigm[6]. Having a private implementation allows us to experiment with different implementation techniques and to instrument the library in order to obtain performance figures.

The library provides a Programming Interface (API), containing two Java classes and one Java interface (Figure 2.1).

```
public class Transmitter {
    public Transmitter(String channel);
    public void transmit(Object object);
    public void close();
}

public class Receiver {
    public Receiver(String channel, Listener listener);
    public void close();
}

public interface Listener {
    public void accept(Object object);
}
```

FIG. 2.1. *Public classes and methods in Radio Broadcast Paradigm library*

An application that wishes to open a channel for publishing data, simply instantiates an object of class *Transmitter*, passing the name of the channel to the constructor.

To open a channel for receiving objects from a channel, the process instantiates an object of the class *Receiver*. The constructor of the *Receiver* class takes the channel name and an object which implements the *Listener* interface as its parameters. This *Listener* object is responsible for dealing with objects received from the channel (the concept of a *Listener* object is similar to the GUI call-back mechanism introduced in JDK 1.1). The only method which must be implemented by an object which implements the *Listener* interface, is the *accept*() method which is called whenever an object is received. It is up to the *Listener* object to handle the object.

The classes *Transmitter* and *Receiver*, and the interface *Listener*, are sufficient to build complete applications. The fact that *Transmitter* and *Receiver* objects may be on completely different machines is completely hidden from the programmer.

The need to define one's own classes implementing the *Listener* interface for even the most basic ways of handling incoming data is rather impractical. Therefore, the library contains a few *Listener* classes for simple processing tasks. They are summarized in Table 2.1.

The behavior of *StateListener*s, *QueueListener*s, *IndexListener*s *Filter*s is fairly intuitive. The *ClassListener* class is more complex, it opens up the possibility of dynamically updating the code in a running system. New parts of the software can be distributed in the architecture-neutral Java byte code format to all processes that have a *ClassListener* defined for a certain channel. The class will be defined and linked dynamically with the Java Virtual Machine in the receiving process. If the class contains any static initializers, these will be evaluated, thereby allowing a class to initialize itself, e.g., by starting a new Thread. The *ClassListener* mechanism is similar to the on-demand loading of classes in Java-based web browsers. However, in our case, loading of classes is *push-based*. It is the transmitting side that decides if and when classes are to be sent. Since the usual

StateListener	keeps last received object
QueueListener	places received objects in a queue
IndexListener	places incoming objects in a table indexed by a key value; new objects overwrite older objects with the same key
Filter	forwards received object to another Listener only if a specified predicate is valid
ClassListener	if incoming object is a class description in Java byte code format, this class is defined and t is linked dynamically with the Java virtual machine

TABLE 2.1
Listener classes

Java dynamic class loading mechanisms are *pull-based*, missing class dependencies can be solved by loading the required class. In our case, missing class dependencies will cause the class defining and linking process to be suspended until the required class is received.

In the example given in Figure 2.2, we show how a *ClassListener* object is used to transfer actual byte code to another object after which the transferred code is executed. In this example, program *HelloProducer* constructs a *ClassDefinition* object containing the byte code for class *HelloPrinter*. *HelloPrinter* is a class that contains a static initializer which will cause the string "Hello, World" to be printed on the screen. *HelloConsumer* is a simple program which only defines a *ClassListener* and then waits indefinitely.

On execution, the code for the class *HelloPrinter* is sent to the *ClassListener* that is contained in objects of the class *HelloConsumer*, and the text "Hello, World" will be printed on the screen.

```
public class HelloPrinter {
  static { System.out.println("Hello, world"); }
}

class HelloProducer {
  public static void main(String[] args) {
    try {
      Object o =
       new ClassDefinition(Class.forName("HelloPrinter"));
      new Transmitter("hello").transmit(o);
    } catch (java.io.IOException e) { }
  }
}

class HelloConsumer {
  public static void main(String[] args) {
    new Receiver("hello", new ClassListener());
    pause(); // Wait indefinitely
  }
}
```

FIG. 2.2. *A simple example in which Java byte code is transferred*

Even though the *transmit*() method takes an *Object* as parameter, only objects which directly or indirectly implement the interface *Serializable* can be transmitted. Objects not implementing this interface are silently discarded. The *Serializable* interface was introduced in Java 1.1 as part of the Object Serialization API. Our implementation of the Radio Broadcast Paradigm library uses these features internally to convert objects to byte arrays before passing them on to receivers. Even

when sending objects within a single Java Virtual Machine, they are serialized and de-serialized.

Consistent serialization has the interesting consequence that any data structure that is transmitted, will become a "deep copy" when received. Consider for example a Vector of objects. Applying the *clone*() method to such a Vector will yield a copy of the vector, however, any reference in the vector refers to the same objects as the references in the original vector. However, sending and receiving the vector through the Radio Broadcast Paradigm library will yield a vector with references to *copies* of the elements of the original vector. This is important when considering the fact that one of the purposes of the Radio Broadcast Paradigm is to decouple processes as much as possible from each other.

2.4. Design Considerations. It is important to realize that the channels provided by the library are *unreliable*. Even though the library makes effort to get objects from transmitters to receivers, delivery of objects is not guaranteed. Similarly, the order in which objects are received is not guaranteed. Even though two objects sent sequentially by the same transmitter will be received in the same order by all receivers, two objects sent by different transmitters may be received in different orders by different receivers. An application cannot make any assumption about such issues.

Besides platform-dependent native-code solutions, the only form of communication between different Java virtual machines is the use of TCP/IP-based socket communication. Other high-level protocols, such as Java's Remote Method Invocation are built on top of this. Our implementation also uses TCP/IP sockets for communication between *Transmitter*s and *Receiver*s located in different Java virtual machines.

The implementation uses one "server" process for each machine in the network on which processes that use the subscription communications are run. Even on a single non-networked host, this server process is still necessary to provide the communications between several processes running concurrently on this machine. Figure 2.3 illustrates the possible communication paths; the "clients" in this figure are Java virtual machines which execute programs that use the subscription facilities.

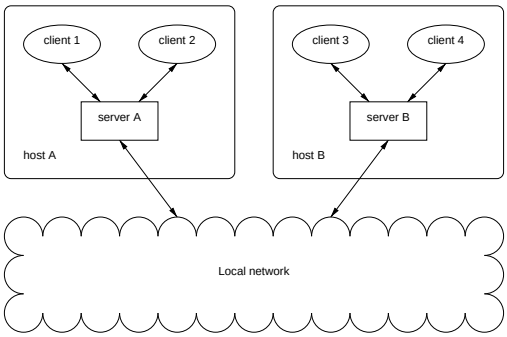

FIG. 2.3. *Communication between clients and servers*

Servers periodically send broadcast messages on the network, allowing other servers to locate each other without using (static) configuration files. It furthermore allows servers to detect when other servers have crashed (i.e., the absence of messages for a longer period of time).

For exchanging actual data over channels, server-to-server UDP messages are sent. The use of UDP, a datagram-oriented protocol, rather than the reliable TCP protocol is based on the assumption [3] that most embedded systems deal with a continuous stream of data from the environment, and that the environment serves as a back-up for occasionally missing a data item. Since TCP is a reliable stream-oriented protocol, the operating system will guarantee that all messages are delivered in the right order. If TCP was used, a packet dropped by the network would cause the data stream to stall until the dropped packet has been re-sent.

Clients communicate with servers using TCP rather than UDP, since using UDP, would have no advantage here. Communication between processes on a single host is always reliable, so the previous scenario in which a missed message causes a stall will not occur.

Periodically, the servers will communicate among each other, and to the clients attached to them, the set of channel names for which their clients have defined *Receiver* objects. Availability of this information minimizes the amount of data exchanged between servers and clients, and among servers.

A class diagram of the client-side library of the Radio Broadcast Paradigm implementation is shown in Figure 2.4. In this figure we show the *Transmitter*, *Receiver* and *Listener* classes. The *Transmitter* and *Receiver* objects contain a reference to the *Channel* object corresponding to the channel they are attached to.

For each channel name, a *channel* object exists. The mapping between channel names and *Channel* objects is maintained by a (single) *Registry* object. Finally, the *Forwarder* class, which is also a singleton, is responsible for communication with the server. It multiplexes and de-multiplexes incoming and outgoing data and control messages to a single TCP/IP connection.

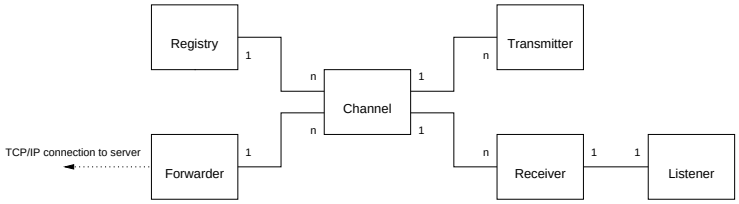

FIG. 2.4. *Class diagram of client-side library*

The server program contains a single "manager" object, and two objects for each connection to the server. These objects handle the incoming and outgoing data stream, respectively. The manager object keeps a table which contains the information on which channels are to be forwarded to which connections. In addition, there is a separate thread which periodically broadcasts to the network to discover the presence of other servers on the network. If a new server is found, a new connection is set up.

The use of Java provides a high degree of portability, the same Java code is used on MS Windows 95, MS Windows 98, MS Windows NT, several Linux versions, and Sun Solaris.

3. Controlling through Subscription: A Case Study.

3.1. Problem Description. Rotterdam is the world largest harbor. One of the largest container terminals in Rotterdam's harbor is fully automated. Cranes are used to carry the containers from a ship onto a lorry or from a lorry to a ship, from lorry onto stacks or onto a train or from trains or stacks onto a lorry.

The terminal employs a system of unmanned, automated vehicles transporting containers. The current system is running for several years now, and it is near its limitations. In a next-generation system, the number of lorries that can be handled has to increase dramatically, while the system should furthermore be able to handle lorries with different characteristics. Finally, traffic will be extended to include drives between terminals. There is no doubt that the complexity of the system will increase dramatically.

In order to get a good understanding of the problems involved in controlling the handling of these large amounts of containers, a prototype simulator/controller is being developed that is being used as a test-bed for a variety of implementation issues. The test-bed is based on control being distributed where each vehicle has its own controller dictating its behavior. The main reason for this design choice is that a distribution model conceptually fits perfectly to the situation of unmanned vehicles moving around. In practice, we will have to evaluate performance before deciding whether central or distributed control is more effective. Our implementation of the test-bed is using the earlier mentioned implementation of the RBP. The approach taken in our study was motivated by our cooperation with another group that is building a similar prototype using a centralized control approach, using called hierarchical semaphores [10] as controlling devices.

3.2. Requirements and Design Decisions. The current lorry handling system is able to handle a few dozen cranes and lorries. The system modeled through our test-bed should be usable

for larger systems, systems with e.g. 1000 lorries rather than the 50, being served currently.

It is obvious that the new system will have to be able to handle changes in the number of lorries that are being dealt with during operation, however, the system also has to take into account the addition of lorries with different kinds of behavior. Main requirements for a solution are therefore

scalability with respect to the number of lorries and the number of movements,

flexibility with respect to changes in the number of lorries and the behavioral patterns of the various vehicles operating on the terrain.

For this paper, it is assumed that a separate planner system exists that provides each vehicle with a plan: a function mapping the time to the position the vehicle should be (or should have been) at that moment. A complete plan provides information on where to collect the container, how to drive over the area and where to deliver the container.

In the design of a system based on an RBP, one must keep in mind that there is no guarantee that data will arrive, other then sending some precious data twice or more times. However, in a case like this, the loss of a single message is no disaster: the receiving processes temporarily use information slightly older than it should be.

3.3. Outline of the Control System. The control system we designed for our case study is discussed in detail in [14]. In this paper we briefly outline the ideas and the structure of the implementation.

The overall guiding principle in addressing the issue is separation of concerns. Separation of concerns on control is obtained by modeling a controlling process for each vehicle in the area, and by letting each controlling process be responsible for the control decisions for the movements of the associated vehicle. In order to do so, each vehicle controlling process needs a consistent view on the area it is in.

The system is built such that a vehicle control process can deduce a sequence of places that it will pass from its given plan. Essential in the model is that the process knows the planned position for the vehicle at any time. Furthermore, it knows the execution attributes of the vehicle it controls, such as the actual position, the velocity and the driving characteristics .

Vehicle control processes send their short-term plans (the part of the detail plan that should be followed in the immediate future) periodically through a channel. Such processes furthermore listen to the short-term plans of other vehicles and evaluate possible collisions. In order to obtain a reasonable behavior of the vehicles, each vehicle obeys a set of traffic rules, and knows the traffic rules of all other vehicles. Traffic rules can be as easy as

- *the lorry with the lowest order number has precedence* or
- *the lorry that is in the greatest hurry has precedence*

or as complex as *a vehicle coming from right has precedence, unless the vehicle from left is delayed more than the vehicle from coming right*

Under the assumption that any vehicle has a consistent view on the area, application of valid traffic rules allows a control process to take a decision on whether to continue, to stop or to alter the velocity in the presence of other vehicles.

We realize that in order for a lorry to take a decision, having a consistent view on the *whole* area is an overshoot. What is required is a consistent view of the direct environment of the lorry. Therefore, the area in which the vehicles move is partitioned in regions: a grid of for instance 10 by 10 coordinated points. To each region one particular channel is associated. A vehicle sends its short-term plan through the channel, associated with the region it is positioned in; it listens to the same region channel, and to the channels of the eight regions surrounding this region as well. Scalability is handled through this approach.

In Figure 3.1 we show two vehicles in different regions. Each sends (solid line) to the channel associated to the region it is positioned in. All vehicles in a given region listen to the same channel, and to the channels associated to the eight regions around it. Note that it is possible that more than just one vehicle drives around in one region.

Furthermore, to each region a region process is associated, that collects data of the vehicles driving in the region. It creates a summary which is sent through the "image channel", available for our visualization system. The visualizer can also be used to zoom in to one region, by telling

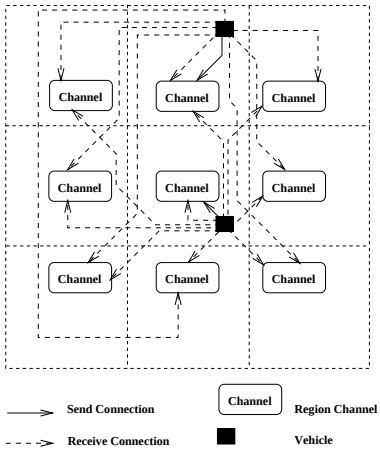

FIG. 3.1. *Vehicles in regions*

it to listen to the associated channel. The short-term plans of the vehicles in that region may be inspected.

The traffic rules are a part of each vehicle process. Every vehicle process listens to a "rules" channel, which may be used to send a new version of these traffic rules. To avoid version conflicts, vehicles send the version number of the rules they use, together with their short-term plan. In case of a version conflict, a default rule is used by both parties.

3.4. Implementation. The controller is built using the RBP library, that was discussed in Section 2.3. In Figure 3.2 we show the internal view of a vehicle process. In this figure, active (threaded) objects are represented by a rounded box, while sharp-edged boxes denote threadless objects.

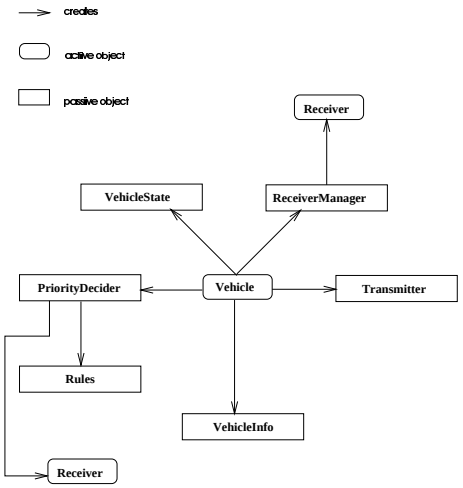

FIG. 3.2. *Objects in the Vehicle Process*

The vehicle process performs a loop:

- The $VehicleState$ object is asked to construct an updated $VehicleInfo$ object. This object contains relevant position information such as the vehicle identity, the speed and the current position of the vehicle, and the short-term plan of the vehicle. Furthermore, it contains the current version of the traffic rules,
- The $Transmitter$ object is asked to transmit the newly created $VehicleInfo$ object.
- In the meantime, the *Receiver* objects, that are connected to the $ReceiverManager$ object listen for information from other vehicles. The $ReceiverManager$ object is responsible for

the choice of the channels to listen to. It selects these channels based on the position of the vehicle.
- Once every few time units, the *PriorityDecider* is asked to compute the next position the vehicle will drive to. This is based on the information gathered from the local environment and - obviously - on the traffic rules. The *VehicleState* object is updated.
- The traffic rules themselves may be updated when a new class definition is sent through the *Rules* channel.

4. Analysis of the Case Study.

4.1. Our Approach to Abstraction and Verification. As stated in the introduction of the paper, analysis and verification should provide feedback on the design and implementation of real-time distributed control systems. Our focus is on analyzing applications and deriving bounds for the execution parameters of the underlying system, i.e. the RBP implementation in our case. The technique we use for this kind of analysis is model checking.

Model checking is becoming more and more part of the development process for hardware and software products. Companies like Intel and Lucent are using model-checking in their development processes and are even developing their own model-checkers. Model-checkers customized for hardware design are becoming commercially available [11]. Integration of model-checking in software development processes is not yet that far developed. There is, however, an increasing number of successful applications within the software industry, but it will take some time before it matches with the applications in the hardware sector.

The computational model that is generally used in the software case is (a variant of) the finite state automaton. The basic model of a finite state automaton provides only a basis for control abstraction. As the model was more widely used, constructions were added that enable the specification of data values and data transformation. To be of any use in the field of real-time systems, time and synchronization were added.

We designed a notation, eXtended Timed Graphs[2], an automaton-based formalism [1] with extensions in the area of data specification and a general form of control semantics. The primary use of the notation is as input notation for our model-checker[16].

The XTG notation is *not* meant to be a modeling language in itself. Other, better suited, modeling languages exist that focus more strongly on the various aspects of system and software design such as composition and decomposition, reuse, object-orientation, etc.

XTG is designed particularly for use as a verification language, it is supported by a more human-oriented specification language ATL, the Abstract Thread Language[17]. An ATL specification can be translated into XTG's. Such XTG's can be fed into the model-checker. In our approach, designs and implementations are are abstracted into ATL specifications, that are fed into the model checker.

Syntax and semantics of the behavioral constructions in ATL are based on C and Java. Furthermore, ATL supports a simple communications model which is loosely based on the subscription paradigm [3, 6]. Communication takes place along "channels". To send information on a channel, the send-operation is used, with the syntax "channelname.send(value)". To receive information from a channel, one defines a receiver thread. A receiver thread declaration consists of the receiver name, the variable which is used to store the value received from the channel, and a statement sequence to be executed every time the send-operation is invoked by some thread. A send-operation will not block until the receiver thread is ready to accept the communicated value. If a receiver thread is not idling, the value to be sent by the send-statement will be lost.

XTG and ATL share a single property specification language, a temporal logic based on CTL (Computation Tree Logic)[4] and TCTL (Timed CTL) [9]. The usage of these logics is strongly connected with the application of model checking verification. TCTL variants are real-time extensions of CTL. These extensions either augment temporal operators with time bounds, or use reset quantifiers.

The core syntax of CTL defines two temporal operators, AU and EU. The formula $(\phi_1 AU \phi_2)$ is satisfied in a state if for all computation paths starting from that state, a state which satisfies ϕ_2 is encountered, and until that time ϕ_1 is satisfied. $(\phi_1 EU \phi_2)$ is satisfied if there is at least one such computation path. Derived operators are: $EF\phi$ (There is path on which there is state satisfying

ϕ), $EG\phi$ (There is a path of which every state satisfies ϕ), $AF\phi$ (On all paths there is some state satisfying ϕ), and $AG\phi$ (On all paths every state satisfies ϕ).

Our CTL variant supports two types atomic properties: boolean expressions over values of variables and clocks of both the system and the property, and location expressions. The latter take the form $g@l$, which expresses the fact that the graph g of the system is currently at location l.

To address resetting quantifiers, we use a technique found in other TCLT's, which consists of modeling the reset quantifier by assignments to specification clocks and variables. As an example, in our CTL variant (referred to as CTL_{XTG} or simply CTL),

$$z := 0.AF(p \wedge z \leq 10)$$

expresses that p will always become true sometime within ten time-units. A secondary benefit from this approach is that we can use symbolic constants in our property formulae (as suggested in [8]). Consider for example, the CTL formula

$$AG(t := count.AF(count = t + 1))$$

in which *count* is a system variable and t is property specification variable.

4.2. A Model of the Controller. The correctness of operation of the controller we discussed in Section 3.3 does also depend on the behavior of the underlying RBP implementation. As discussed in Section 1, we therefore should model the behavior of the controller implementation, identify the dependencies on the underlying implementation and derive bounds for the various parameter values. In this section, we demonstrate the usefulness of such analysis using a highly simplified version of the autonomous vehicle system as described in Section 3. In the simplified scenario discussed here, we take only two vehicles into account. These vehicles drive in a straight line towards each other as shown schematically in Figure 4.1.

Each vehicle maintains two positions, its own and the other one's, both represented by a simple integer value. It is assumed that the vehicle's own position is accurate, the position it believes the other vehicle has, is based on the information it received.

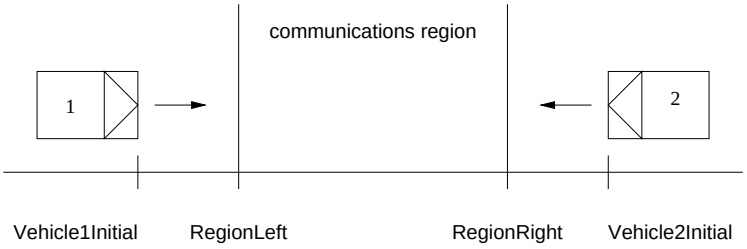

FIG. 4.1. *Scenario of the simplified model*

Even though this model is an oversimplification of the original system, it contains the essential elements: vehicles publishing their own position, and making decisions based on received information. The communications model is unreliable, in the sense that a message will be lost when a receiver is not listening, e.g. since it is busy processing a previous message.

In the ATL specification each of the two vehicles is modeled by of a controlling process and a receiver process. The controlling process periodically sends the vehicle's position to the receiver process of the other vehicle. The controlling process of a vehicle retrieves the (assumed) position of the other vehicle from a variable shared with the receiver process. Figure 4.2 shows the code of the controlling process and the receiver of one of the vehicles. The code for the other vehicle (defining the position variable $p2$) is similar and is omitted here.

4.3. Analysis of the Model. In our experiments we are interested in the the safety property "the two vehicles will never collide" and under which parameter values, i.e. indicating execution attributes of the underlying RBP implementation, the property holds. The non-collision property is expressed as `AG p1 != p2`. A vehicle might collide with the other one if the perceived position

```
// Vehicle1's position
int p1 = Vehicle1Initial;
// Vehicle1's idea of Vehicle2's position
int q1 = Vehicle2Initial;
// Controlling process for Vehicle1
process Vehicle1 {
   forever {
      if (p1 >= RegionLeft && p1 <= RegionRight)
         plan1.send(p1+1);
         sleep(VehicleDelay);
      if ((q1-p1) > Distance || (p1-q1) > Distance)
         p1++;
   }
}
// Receiver process in vehicle 1
// receives position from Vehicle2
receiver plan2(int buffer) {
   sleep(ReceiverDelay);
   q1 = buffer;
}
```

FIG. 4.2. *ATL code of the vehicle*

Vehicle1Initial	0
RegionLeft	3
RegionRight	12
Vehicle2Initial	15
Distance	2
VehicleDelay	4
ReceiverDelay	7

FIG. 4.3. *Constants used in experiments*

of the other vehicle differs sufficiently from the actual position of that vehicle. Such an inconsistent view might be obtained when processing of the received positions is too slow. Several verification experiments were performed based on the above model.

Verification of the non-collision property In a first experiment, we verified the safety property for a given set of constant values. The chosen values for the constants are shown in Table 4.3. As can be derived from the ATL description of the vehicle, $Vehicle1Initial$ and $Vehicle2Initial$ indicate the initial positions of the vehicles. $RegionLeft$ and $RegionRight$ indicate the region, $Distance$ indicates the required minimum distance between the two vehicles, $VehicleDelay$ indicates the time between two iterations of the $Vehicle$ processes and $ReceiverDelay$ indicates the Receiver delay. The safety property turns out to hold.

Deriving the communications region In a second experiment we derived the smallest communications region possible for which no collision will occur. When the region is too small, the vehicles will not communicate their position fast enough to each other to allow a reaction. If `RegionLeft` is made a parameter, the safety property is shown to be satisfied when `RegionLeft` ≤ 6, In the same way, the bound `RegionRight` ≥ 9 was derived.

Deriving upper-bounds for parameter values A third experiment was aimed to derive bounds on the various delays in the vehicle program. First, we determined an upper bound on the `ReceiverDelay`. Too high a value for the `ReceiverDelay` will cause the vehicles to collide, as they will not have a correct idea of the position of the other vehicle. Except for the value of `ReceiverDelay`, we used the same constant values as in Table 4.3. The safety property is satisfied for `ReceiverDelay` ≤ 7.

A similar (fourth) experiment was performed for the value of `VehicleDelay`. To ensure that the verification process ended, we needed to impose an upper-bound on the value of `VehicleDelay`, by replacing the statement

```
delay(VehicleDelay);
```

by

```
if (VehicleDelay < VehicleDelayMax)
    delay(VehicleDelay);
else
    delay(VehicleDelayMax);
```

where `VehicleDelayMax` is an appropriately chosen constant. After this modification, the property is satisfied for `VehicleDelay` ≥ 4.

Deriving dependencies between parameters In the fifth experiment, we took two parameters, `VehicleDelay` and `ReceiverDelay`. The values, derived our model checker PMC, for which the safety property is satisfied are shown as the shaded area in Figure 4.4. In this experiment,

we limited the values of `VehicleDelay` to the range from 0 to the value of `VehicleDelayMax` of 10. The relationship between `VehicleDelay` and `ReceiverDelay` approximates a linear relation described by: `ReceiverDelay` $< 2 \times$ `VehicleDelay`. As the delay value in the vehicle controlling process gets longer, the communications layer is allowed to have a longer latency. The coefficient of 2 can be explained by the value of `Distance` (the minimal distance vehicles will try to keep between them). Changing the value of `Distance`, changes the coefficient as well.

Finally, (sixth experiment) we considered independent delays of both receivers In Figure 4.5. We show the values of `ReceiverDelay1` and `ReceiverDelay2` for which the vehicles will not collide. From this figure, it can be seen that a longer communications latency in one direction can be compensated by a shorter delay in the other direction. This result shows a relationship which can not be described by an approximation of a linear relation. This can be explained by realizing that the vehicle controller process "samples" the position of the other vehicle at regular intervals of `VehicleDelay`. Changing the value of `VehicleDelay` will change the size of the "blocks" in the figure, although the shape will be similar.

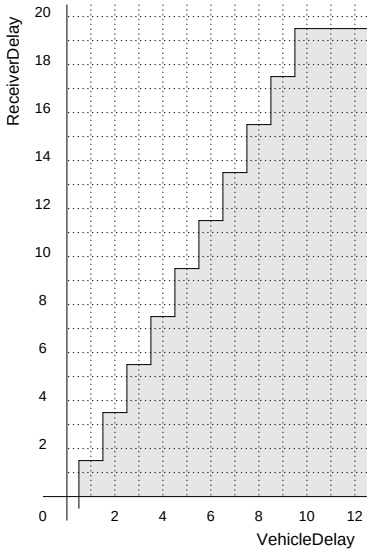

FIG. 4.4. *Verification result using* `VehicleDelay` *and* `ReceiverDelay` *as parameter; shaded area shows were property is satisfied*

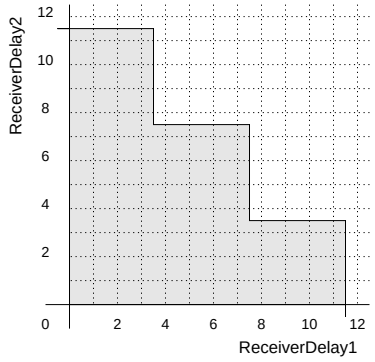

FIG. 4.5. *Verification result using* `ReceiverDelay1` *and* `ReceiverDelay2` *as parameter; shaded area shows were property is satisfied.*

Results. In Figure 4.6 the results of the verification experiments described above are described. The objective of these experiments is to demonstrate the use of our tool-set for the (formal) derivation of values for which the required property holds. The parameter values themselves denote execution attributes of the underlying system. The result indicates the ranges for the values of the various attributes such that the property is guaranteed to hold.

#	Property	Parameters	Result
1	AG p1 != p2	—	positive
2	AG p1 != p2	RegionLeft	RegionLeft ≤ 6
	AG p1 != p2	RegionRight	RegionRight ≥ 9
3	AG p1 != p2	ReceiverDelay	ReceiverDelay ≤ 7
4	AG p1 != p2	VehicleDelay	VehicleDelay ≥ 4
5	AG p1 != p2	ReceiverDelay, VehicleDelay	Figure 4.4
6	AG p1 != p2	ReceiverDelay1, ReceiverDelay 2	Figure 4.5

FIG. 4.6. *Verification results*

4.4. Running the Model. The next step, obviously is to investigate under which conditions the underlying system has as its execution attributes values within the ranges specified above. In order to get some feeling of the results obtained analytically, we performed a series of experiments with the same system that was used a basis for analysis. The system simulated (and controlled) the behavior of two vehicles.

One vehicle had as its mission to drive from a point (0, 0) to the point (30, 0), the other had to drive the opposite way, both with a step-size 1. The experiments were done three times:

- once with both vehicles running on a Linux system.
- once with both vehicles running on a Windows NT system,
- once with one vehicle running on the Linux system, and the other on the NT system.

In all cases the simulation was checked for collision between the vehicles.

Varied parameter The parameter that was varied in the experiment, was the cycle-time of the vehicles. This cycle-time determines the *vehicledelay*, which amounts to the cycle-time plus 5 milliseconds.

Measured parameter The parameter that was measured was *ReceiverDelay* (the time between the moment a message is sent, and the moment it is received), in relation to the occurrence of collisions.

Default, the cycle-time of the vehicles was set to one second, meaning that each second, a vehicle checks the data it did receive giving information on the position of the other vehicle, and it computes its next step based on those data. Measurement shows that the time needed for the computation is app. 5 milliseconds (on both systems). The actual cycle-time is therefore 1005 milliseconds, the measured *receiverdelay* showed to vary between 0 and 12 milliseconds, with consistent results in all three experiments. With these values for *cycletime* and *receiverdelay*, simulation showed that the vehicles always detect each other in time, and no collision occurs.

Varying the cycle time gives consistent results. The measured *ReceiverDelay* varies between the same values, whether both vehicles are run on the Linux system, on the NT system, or both on a different machine. The results are presented in Figure 4.7.

Cycle-Time	ReceiverDelay	Detect on Time
1000	0-12	yes
5	5400-6600	no
100	2900-3800	no
300	500-600	sometimes
350	250-360	yes
500	170-370	yes

FIG. 4.7. *Results of the experiment*

It shows that a shorter *cycletime* results in a longer *receiverdelay*. For our network (where the *ReceiverDelay* does not depend on whether the vehicles run on different systems or on the same system), the critical value of the *cycletime* appears to be about 300 milliseconds: as long as the value of *cycletime* is larger, collisions are avoided. When shorter *cycletime*'s are chosen, collisions almost certainly will take place.

These results are consistent with what was derived from the results of the verification experiment, summarized in Figure 4.4. When *ReceiverDelay* becomes larger than 8 (twice 4), collisions cannot be avoided.

5. Conclusions and Further Work. In this study we introduced our approach to software development of dynamic real-time distributed systems based on complementary use of experimentation, abstraction and verification. The focus of our study was on a particular approach to model the middle-ware between distributed control applications and the underlying network structure, the Radio Broadcast Paradigm (RBP).

We created a test-bed for analyzing applications using our RBP through a Java implementation and experimented the RBP on an industrial case study. Abstraction of the Java implementation

into a formal abstract language and the translation of the resulting representation into a modeling language for parametric verification enabled the derivation of constraints on the underlying execution environment. Our combined approach thus provides a sophisticated framework for the development of evolving distributed real-time systems.

Our experiments show that the approach is viable, the results of our experiments were very encouraging, although scaling is an issue. When more vehicles are involved, experiments based on model-checker predictions are more complex and the model-checking itself requires far more resources. One practical issue to be addressed is visualization of results, as more variables depend on each other, visual representations of the dependencies become complex.

Current validation runs were done on a large Linux machine, but could have been done on any PC. Our current experiments involve larger control configurations (over a dozen vehicles involved) than given here, and verification runs require somewhat more memory than available on an ordinary PC. However, due to the symmetry in the ATL specification, the amount of memory for model-checking in the complete verification remains within limits. The model(s) provide useful insight in further optimizations in the model checker.

Our current work consists of optimizing the model checker, and developing robust translators for design notations like UML, Java and C languages into ATL specifications.

REFERENCES

[1] R. Alur and D. Dill. The theory of timed automata. LNCS, pages 45–73. Springer-Verlag, 1991.

[2] M. Ammerlaan, R. Lutje-Spelberg, and W. Toetenel. XTG – an engineering approach to modelling and analysis of real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 88–97. IEEE press, 1998.

[3] M. Boasson. Subscription as a model for the architecture of embedded systems. In *Proceedings of the 2nd International Conference on Engineering of Complex Computer Systems*, pages 130–133. IEEE Computer Society Press, 1996.

[4] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.

[5] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, second edition, 1994.

[6] R. C. M. de Rooij. Subscription-based communication for distributed embedded Java applications. In *Proceedings of the fourth annual conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1998.

[7] C. Heitmeyer. On the Need for Practical Formal Methods. In *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 18–26. Springer Verlag, 1998.

[8] T. Henzinger. The theory of hybrid automata. In *LICS'96*, pages 278–292. IEEE Computer Society Press, 1996.

[9] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[10] J.J.M.Evers. Ditrans, toekomst gericht gautomatiseerd containertransport. Technical Report ISBN 90-5584-004-1, TRAIL Studies nr 94/4, 1999.

[11] R. Kurshan. Formal verification in a commercial setting. In *Proceedings Design Automation Conference*, 1997.

[12] S. Microsystems. The javaspaces specification. http://www.javasoft.com/products/javaspaces/specs/js.pdf, 1998.

[13] R. Rajkumar and M. Gagliardi. High availability in the real-time publisher/subscriber inter-process communication model. In *Proceedings 17th IEEE Real-Time Systems Symposium*, pages 136–141, Washington, DC, Dec. 1996. IEEE Computer Society Technical Committee on Real-Time Systems, IEEE Computer Society Press.

[14] S. Stuurman and J. van Katwijk. On-line change mechanisms, the software architectural level. In *Sixth International Symposium on the Foundations of Software Engineering*, Orlando, Florida, 1998.

[15] TIBCO. TIB/Rendezvous – white paper. World Wide Web Document, `http://www.rv.tibco.com/rvwhitepaper.html`, 1997.

[16] http://tvs.twi.tudelft.nl.

[17] J. van Katwijk, R. de Rooij, and W. Toetenel. ATL language definition. Technical Report To appear, Delft University of Technology, Faculty of Information Technology and Systems, 1999.