On-Line Change Mechanisms the Software Architectural Level

(Experience Paper)

Sylvia Stuurman, Jan van Katwijk
Department of Mathematisc and Informatics,
Delft University of Technology, P.O.Box 356, 2600 AJ Delft,
The Netherlands.
E-mail: S.Stuurman@twi.tudelft.nl, J.vanKatwijk@twi.tudelft.nl

Abstract

Our interest in the field of software architecture is focused on the application in technical systems, such as control systems. Our current research in this field is centered around a real-life case study, a control system for unmanned vehicles transporting containers on the "Maasvlakte", an area in the ports of Rotterdam.

Important issues in this control system are scalability, evolvability, and on-line change capacities.

In this paper, we discuss two mechanisms for on-line change in the distributed control system for the Maasvlakte system, which we have implemented in Java. The software architecture we use is a configuration of distributed processes, communicating according to the subscription model. We will focus on the software architectural aspects of the mechanisms for on-line change. One of these mechanisms is associated with the decoupling of processes as a result of the subscription-based communication model. The other mechanism is based on the late-binding properties of Java.

1 Introduction

1.1 On-Line Change

A structural property of software seems to be that changes are needed from time to time. Even in the situation that we would be able to deliver systems without bugs, meeting exactly the specified requirements, the dynamic environment in which the piece of software struggles for life would eventually dictate new or different requirements, which can only be met by changing the software system.

In fact, a requirement of probably all software systems should be a certain degree of flexibility with respect to the remainder of the requirements. One should design for change.

In several systems, shutting the system down to apply changes is unacceptable. According to Stankovic in [13], on-line change capabilities will especially be needed in the field of real-time and embedded systems.

SIGSOFT '98 11/98 Florida, USA

1.2 Software Architecture

The recently emerged field of software architecture addresses the design of overall system structure. Design for change should start at this level.

Software architectures are typically described as a composition of high-level connected components ([5]). The expression has often been used to indicate structures representing the development view of a system, i.e. the high-level structure of the code (in [15] for instance, "software architecture" is always used in this way).

In recent years, software architectures more and more describe the high-level design of the software system as it is seen during execution, with connections representing "interact" relationships as opposed to "implements" relationships ([1]). Figure 1 shows these two usages of software architecture, with the "run-time view" indicating the latter concept.

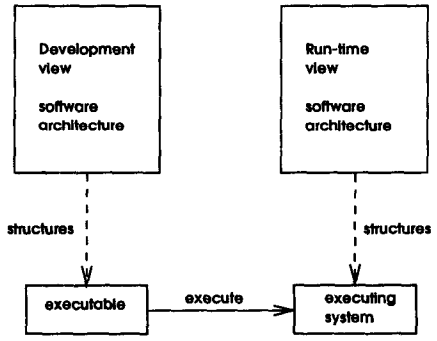

Figure 1: Development and Run-time view Software Architectures

Kruchten extends this concept of software architecture in [9] into four views: the logical view, supporting the functional requirements; the process view, focusing on concurrency and synchronization aspects; the physical view, mapping software on hardware; and the development view, describing the "implements" relationships.

In this paper, we will adhere to the notion of software architecture as components connected by "interact" relationships, which generally means a combination of the logical and process view as seen by Kruchten.

We will tackle the problem of on-line change at the software architecture level, using a real-life example, and discuss

the problem more generally.

1.3 A Case Study

The case study we use to experiment with our ideas is the control of unmanned vehicles on the "Maasvlakte" in the ports of Rotterdam. Rotterdam is one of Europe's mainports, were huge numbers of containers are handled.

In the harbour area, containers are moved to and from ships, onto trains and lorries. Cranes are used to carry the containers from a ship onto a lorry, from a lorry to a stack, from a stack onto a train, or the other way around.

Because of the huge safety risks involved, the harbour currently employs a system of automated, unmanned vehicles for the transport of the containers. In the near future, extensions of this system are foreseen. More terminals will be involved, the number of vehicles will increase (rather 1000 than the current number of 50), and vehicles with different characteristics will be employed.

The current system, controlled from a central site, is not able to handle these changes. The system we are designing should not only be usable for the current situation, but also for future changes, and should allow large scale simulations as well. Scalability and evolvability thus are important non-functional requirements of the system to be designed.

1.4 Outline of the Paper

The paper is organized as follows:
In section 2 we give an outline of the system we designed. We took the decision to distribute the control, and to use communication between the processes according to the so-called "subscription model".

The two different mechanisms for on-line change that are available in our solution for the control system are discussed in section 3. One of the mechanisms is associated with the decoupling of processes as a result of the subscription based communication. The other mechanism is based on the late-binding properties of Java. We show the implications of both mechanisms on the software architecture.

Related work is discussed in section 4. In section 5 we discuss the implications of what we have found, come up with advantages and disadvantages of both mechanisms for on-line change, and show some loose ends. With section 6, we end with some concluding remarks, and express wishes for future research.

2 A Control System for the Maasvlakte

The problem of the case study comes down to control the movements of unmanned vehicles from one given place to another: the problem of assigning tasks to vehicles is another subproblem.
In an early stage, we took two decisions:

- Control is distributed. Each vehicle has its own controller dictating its behaviour. The main reason for this decision is that a distributed 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.
- Communication between the vehicle controllers and other processes is modeled according to the "subscription model", which is explained below.

2.1 Subscription Model Communication

Subscription based communication between processes ([2]) is a way of loosening the coupling between processes. A conceptual model for subscription based communication is the use of radiographic frequencies, called "channels":

- The availability of an unbounded number of channels is assumed.
- For each channel, the type of data which are sent is defined.
- Each process may send through whichever channels it wants, as long as it adheres to the kind of data that are expected.
- A process may subscribe to channels. Processes are able to "listen" to the data sent along the channels to which they are subscribed.
- Subscriptions may be done (and undone) on-line.

An analogy to this form of communication is the way usenet users talk to each other through newsgroups. Someone may send messages to newsgroups. One subscribes to the newsgroups one is interested in, and reads messages appearing in those groups. Users have no need for direct connections, and may in principle stay anonymous.

In a system of distributed processes communicating according to the subscription model, direct connections between processes are avoided.

However, one must keep in mind the restrictions of this form of communication:

- The order in which the data are sent is not necessarily the same order at which they are received.
- There is no way to encertain that data are not lost, other then sending some precious data twice or more times. The typical use of subscription-based communication is where the same kind of data is sent over and over again, every time slightly different. In such a case, the loss of one message is no disaster: the receiving processes temporarily use information slightly older than it should be.

2.2 Outline of the Control System

The control system we designed for the Maasvlakte case study is discussed in detail in [12]. In this paper, we will briefly outline the ideas.

The whole area where the vehicles may drive is characterized by x- and y-coordinates. Each unmanned vehicle is controlled by its own vehicle process.

We assume that:

- A seperate planner system provides each vehicle with a plan: the place where it should receive a container from a crane, and the place where it should deliver the container. Places are represented by their coordinates.
- A vehicle process is able to deduce a detailed plan (a sequence of places) from the given plan.
- Each vehicle process knows the position (in terms of coordinates), the velocity and the characteristics of the vehicle it controls.

Vehicles send their short-term plans (the part of the detail plan that should be followed in the immediate future) regularly through a channel. They listen to the short-term plans of other vehicles and evaluate possible collisions. In the case of a possible collision, traffic rules determine which vehicle should wait for the other.

To avoid scalability problems, we divided the whole area in so-called regions: a grid of for instance 10 by 10 coordinated points. Each region is associated with one channel. Each 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.

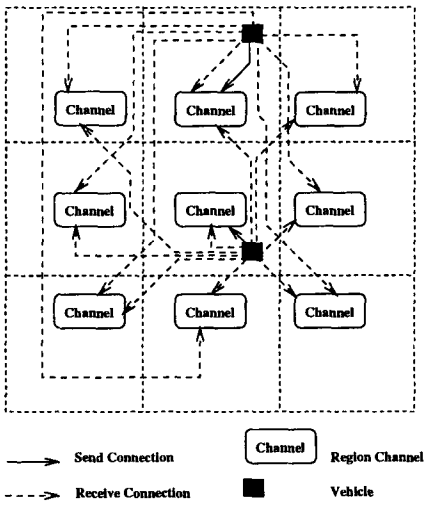

Figure 2: Vehciles in Regions

Figure 2 shows two vehicles in different regions. Each sends (solid line) to the channel associated to the region it is positioned in. Each vehcile listens to the same channel, and to the channels belonging to the eight regions around it. Note that it is possible that more than one vehicle drives around in one region.

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

2.3 Software Architecture Large-Scale

Figure 3 shows the software architecture of the proposed solution for the Maasvlakte case. A vehicle process (processes are represented by a circle) sends (denoted by a "Send"

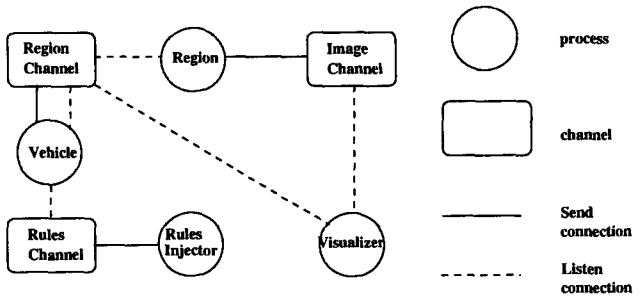

Figure 3: Configuration of Processes and Connections

connection) through a region channel (channels are represented by rounded boxes), and listens (denoted by a "Listen" connection) to nine region channels.

A region process summarizes the data going through a region channel, and sends the summary through the image channel. The visualizer process may either listen to the image channel, and receive the summaries of all regions, or listen to one region, to receive detailed information. Every vehicle process listens to the rules channel, which is, from time to time, provided with new rules, by the rules injector.

2.4 Implementation

The system is implemented in Java. At this moment, we have a simulation of the system running on the computer network (Unix, Linux and Windows95) of our research group. We are currently evaluating the possibilities of a "transport lab" with real miniature vehicles communicating through radio transmission.

Communication according to the subscription model is implemented in the "channel library", by de Rooij ([3]). This library offers a set of classes with facilities to send and receive objects, in the form of (unthreaded) Transmitter classes and (threaded) Receiver classes. One subclass of the Receiver class is meant for class definitions. It transforms a received class definition into a Java class, and instantiates the class into an object. This class makes use of the late-binding properties of Java. However, normally in Java, new class definitions are "pulled" by the class needing it; in this case, we had to "push" new class definitions into the system and force processes to instantiate them.

Figure 4 shows the internal view of a vehicle process. In this figure, active (threaded) objects are represented by a rounded box, while sharp-edged boxes denote unthreaded objects.

The vehicle object performs a loop:

- The VehicleState object is asked to construct the current VehicleInfo (consisting of the version of the traffic rules, the identity, speed and position of the vehicle, and the short-term plan).
- The Transmitter is asked to transmit the VehicleInfo object.
- In the meantime, the Receiver objects 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.

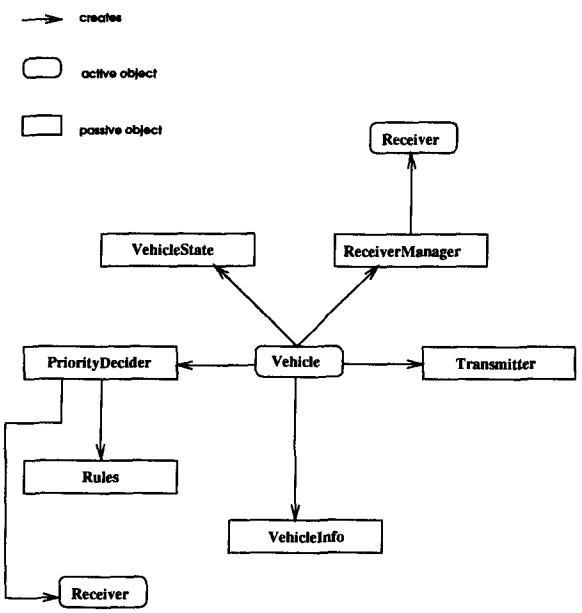

Figure 4: Objects in the Vehicle Process

- Once in a while, the PriorityDecider is asked to compute the next position to drive to, based on the information gathered, and the traffic rules. The VehicleState object is updated.
- The traffic rules themselves may be updated when a new classdefinition is sent through the Rules channel.

3 On-Line Change at the Software Architecture Level

In the solution for the Maasvlakte case, the initiative for changes is always taken from outside the system. Two mechanisms for applying these changes are available:

- Create or remove processes.
- Send a new subclass.

3.1 On-Line Change on the Process Level

The reason that on-line changes in the form of the creation or the removal of processes (or both), can be applied very easily in the system for the Maasvlakte, is the fact that we made use of the subscription model for communication.

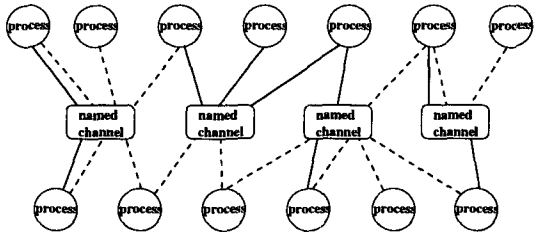

Figure 5: Subscription-Based Communication

Figure 5 shows a possible configuration of processes, channels, send- and listen connections. A solid-line connection stands for a "send" connection; a dotted line for a "listen" connection. Processes never have a direct connection to other processes, and are anonymous. Channels are named, so processes need to know the name (and the type of data connected with the channel) of the channels to which they send or receive data.

When a process is removed, no other process has to be informed. The same applies for the creation of a new process. Channels can be added too, without affecting other channels, or any of the existing processes.

What is needed however, to change a software system communicating according to the subscription model, is an "all-knowing" entity outside of the system. This "all-knowing" entity has a direct connection to any of the processes, to be able to delete them. Moreover, it knows the names and the datatypes of every channel.

Making use of process creation and removal to induce on-line change in a subscription-based system, means that one deviates from the model, and allows one process to have direct connections to every process and every connection within the system.

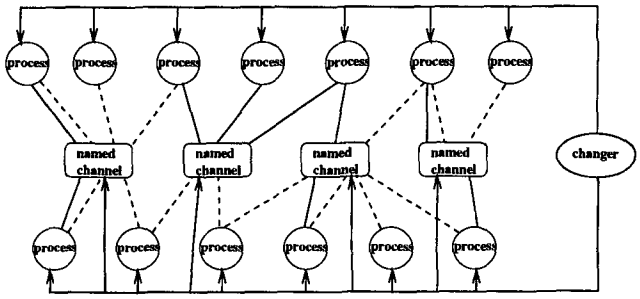

Figure 6: Creation and Removal of Processes

Figure 6 shows this need for direct connections. The "changer" in this figure may stand for a human, for a seperate process, or for one or more of the processes within the actual system. The type of connections needed for such a system are:

- "Send" connection. From process to channel. Belongs to the subscription model.
- "Listen" connection. From process to channel. Belongs to the subscription model.
- "Create or Remove" connection. From process to process. Deviates from the subscription model.
- "Create" connection. From process to channel. Deviates from the subscription model.

3.2 On-Line Change on the Class Level

Enforcing on-line change by the mechanism of sending new classes is possible when every process that has to be changed listens to the channel, reserved for the communication of class definitions.

Figure 7 shows that in this case, the connections available in the subscription model are sufficient.

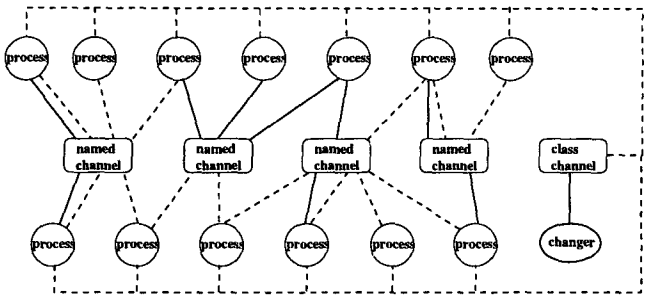

Figure 7: Sending New Subclasses

Processes should all listen to a channel, reserved for class definitions. Upon arrival, the class definition is instantiated into an object, which should connect itself to objects in the process. In the case study, this change mechanism is used for the traffic rules, but in principle, every class used in the processes of a system may be changed in this way.

3.3 Initiating an On-line Change

As we have seen, an advantage of a subscription based architecture is the ease of on-line changes. Both mechanisms are based on the fact that processes in this architecture, are anonymous.

Changing the system at the process level means that one introduces a meta-level: one (or more) processes need direct connections to processes that are to be removed. In general, this "meta-level system", will consist of a change management system and the control system itself. The control system is wired according to the subscription model, while the change management system maintains direct connections to the processes of the control system.

When changing the system by sending a new classdefinition, the processes that are affected are anonymous too: processes that listen to the Class channel, and that use the communicated class, will change their behaviour. The sender of the new class doesn't need to know which processes.

This mechanism for on-line change is more in line with the nature of distributed systems: there is no central control, even not for implementing changes.

This opens the possibility of implementing systems that are able to change themselves, from within: one of the processes might notice the need for change of a certain class. This process, within the system, may send a class definition for such a new class, and thus induce a change in every process using that same class. Such a scenario is not possible when using on-line change on the process-level, because the initiator of a change should know which processes are involved, and should have direct connections to them.

On-line change at the process level thus is most suitable for situations where the initiative for a change lies outside the actual system. A change management system is added to the control system, which can be used to check the validity of proposed changes, maintain consistency, etcetera. This change management system should maintain direct connections with the processes of the actual control system.

Changes at the process level may of course be used within the control system itself as well, but in this case, direct connections are needed within the control system, which is not obvious for a system based on anonymous communication.

Changes on the class level on the other hand, may be induced by every process within the control system, without any need for a change of the used subscription based architecture. Of course, changes on the class level may be initiated from the outside as well.

4 Related Work

A software architectural style which is closely related to the subscription based model, is the implicit envocation style ([6]). In this style, components communicate anonymously as well. The difference is, that implicit invocation is a mechanism to call a method of another component anonymously (by raising an event, upon which all components that have subscribed to the event are called by their associated method). This means that on-line removal of components requires more attention than in the case of subscription-based systems: when a component is removed, all components that are calling a method, anonymously or not, are affected.

The Distributed Software Engineering Group of Kramer and Magee has done a lot of work on on-line change in distributed systems. In [8] they introduced a model for dynamic change management in distributed system. They specify changes at the architectural level (without mentioning it explicitly: the paper was written before "software architecture" became an issue).

Changes have the form of creation or removal of processes. The software architecture they implicitly assume is one with direct connections between the processes. A change management system is used to make certain that changes that have an effect on certain processes are only applied at moments when these processes are not involved in a communication transaction. This notion of a "safe state" is elaborated upon in [7].

By using the subscription model, we avoid the problem of having to check for a safe state: because communication takes place through channels, no harm is done when a process is killed while sending or receiving data.

In [10] their specification language for software architectures, "Darwin", is described. Using Darwin, it is possible to specify the creation or removal of processes, and the associated connections.

Darwin might be a good candidate as an ADL for subscription based software architectures.

Oreizy, in [11], defines runtime architectural change in terms of components: he discerns addition, removal and replacement of components, and reconfiguration. He focuses on one architectural style, C2 ([14]). In the paper, tool-support for on-line change in C2-based systems is proposed. On-line change on the component level is again associated with direct connections between components.

Frieder and Segal described a scheme for procedure replacement in [4]. Our class-level mechanism is in fact, like this scheme, a change at source-code level. We use the decoupling aspects of the software architecture, together with the late-binding properties of Java, to enable the use of source-code changes at the architectural level.

5 Discussion

On-line change of systems, at the architectural level, is a rather new research area. The approaches that we have seen

until now aim at the creation and removal of components in an architecture with direct connections.

By making use of a subscription-based model, we get rid of the problem of applying changes at the right moment. Another reason why we don't have to worry too much about lost messages is the nature of subscription-based communication. It should be used for information that is sent over and over again, every time slightly different. In such a case, it is no problem when some of the data are lost during the removal or the creation of a process.

On the other hand, we make use of the same connections for the distribution of class definitions. As has been said, the only way to handle precious information, is to send it more then once. There is no way to make certain that eventually every process will receive a new class definition, and there is certainly no way to know whether or not every process has received a new class.

This fact has implications on the internals of the processes involved: one must always take into account that it might be possible that two processes have different versions of the same class. In our Maasvlakte case, where we send new traffic rules in the form of class definitions, we handle this inconsistency problem by having the vehicle processes send the version of the traffic rules they use, together with their short-term plan and other information. In the case of a version conflict, every process involved uses one general default rule.

On-line change by creation and removal of processes is not new. New processes may get connections to existing channels, or new channels may be created. When using this kind of on-line change in a subscription-based software architecture, one avoids connection-related problems on the one hand, but on the other hand, one needs to violate the software architecture to enable the removal of processes.

On-line change by the distribution of new class definitions is new. This mechanism is facilitated by the late-binding properties of Java, but on the other hand, we had to find a way to enforce receiving processes to use a received class: normally, new versions of Java-classes are communicated on a pull-base, while we needed communication of class definition on a push-base.

In fact, "precious" information should be sent through direct connections. This enables one to use protocols to check whether the information is really received, or to handle in the case of failing processes or failing connections.

A wish for future research certainly is to explore combinations of direct connections and anonymous communication.

Changes at the class level can be implemented by removal and creation of a process, provided that one has the means to save the state of a process. The same applies for the other way around. So, the choice for process level or class lever changes should be based on other considerations. One such a consideration is the question wether the initiative for changes come from within or from outside. Another consideration could be the fact that direct connections from "outside" are necessary anyway because the user should be able to have direct control.

6 Conclusion

We have presented two mechanisms for on-line change in a control system consisting of distributed processes, communicating according to the subscription model.

One mechanism is to remove and create processes. As a result of the used communication model, problems of finding a safe state almost don't exist. On the other hand, one needs direct connections to the processes involved.

The other mechanism makes optimal use of the anonymous connections, sending new class definitions through channels. The problem in this case is that the application should be able to handle version conflicts.

We already mentioned that a wish for future research is to explore the advantages and disadvantages of combining direct and anonymous connections.

For the future, we would like to explore both on-line change mechanisms. We would like to build a change management system, and elaborate on the different kind of consistency checks that might be done by such a system.

With respect to on-line change by distributing new class definitions, we would like to build ready-to-use components for control systems in the domain of logistic problems. With these components, one should be able to build distributed control systems, with the possibility to update the used classes while the system is running.

In order to build a system which enables users to construct a system by wiring together components, we need a formal description of the software architecture that we use. Darwin looks like a candidate specification language to do so.

References

[1] R. Allen and D. Garlan. Beyond definition and use: Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, January 1994.

[2] M. Boasson. Subscription as a model for the architecture of embedded systems. In *Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*, Montreal, Canada, 1996.

[3] R.C.M. de Rooij. Subscription-based communication for distributed embedded java. In *ASCI Conference*, accepted for publication 1998.

[4] O. Frieder and M.E. Segal. Dynamic program updating in a distributed computer system. In *Proceedings of the IEEE Conference on Software Maintenance*, Phoenix, Arizona, October 1988.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994.

[6] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe*, volume 551 of *Lecture Notes in Computer Science*, pages 31–44, Noorwijkerhout, the Netherlands, 1991.

[7] K.M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, 1996.

[8] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[9] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5):42–50, November 1995.

[10] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Fourth SIGSOFT Symposium on the Foundation of Software Engineering*, 1996.

[11] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering*, 1998.

[12] S.Stuurman. An implementation of a controller for unmanned lorries and its performance. Technical Report to appear, Faculty of Techincal Mathematics and Informatics, Delft University of Technology, 1998.

[13] J.A. Stankovic. Real-time and embedded systems. Group Report of the Real-Time Working Group of the IEEE Technical Committee on Real-Time Systems, at http://www-ccs.cs.umass.edu/sdcr/rt.ps, 1996.

[14] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, pages 390–406, June 1996.

[15] B. Witt, F.T. Baker, and E.W. Merritt. *Software Architecture and Design: Principles, Models and Methods.* Van Nostrand Reinhold, New York, 1994.