Evaluation of Software Architectures for a Control System: A Case Study

Sylvia Stuurman and Jan van Katwijk

Delft University of Technology

Abstract. In this paper, we give our view on the software architecture phase in the development process. During this phase, we distinguish modeling and structuring activities. A system is modeled according to a certain approach, and this model is used to instantiate a certain architectural style. In general, the activities are intertwined.
The choice for a certain software architecture has implications on the non-functional properties of the system. We illustrate our view with a case study of a software controller for a (toy) railroad system which we have available in our software lab. Several models of this system, expressed in formal specification languages, were made in the past, so we are able to produce a software architecture for the system while carrying out both activities separately.
The resulting software architectures are evaluated with respect to timing aspects, scalability, fault-tolerance, and extendibility. Extendibility of a software system is especially important for domains were changes should be applicable on-line. Design for change should start at the software architectural level.

1 Introduction

In this paper, we illustrate our view of the software architecture phase in the development process and the implications of the choice for a certain architectural style within that phase, with a case study of a railroad system. The essence of our view is that we distinguish modeling and structuring activities. The choice for a certain software architecture primarily has an impact on the non-functional requirements of the system. Therefore, when evaluating different architectures for a certain system, one should take into account the non-functional properties that are relevant. Roughly said, one addresses the functional requirements during the modeling activities, and the non-functional requirements during the structuring activities.

A software architecture-driven development process consists of a Requirements Analysis phase, a Software Architecture phase, a Construction phase, and a Maintenance and Change phase. During the Software Architecture phase, one models the system, chooses a software architecture style, instantiates this style, and refines the instantiation either by adding detail or by decomposing components or connections (again going through modeling, choosing a style, instantiation and refinement). This process should result in an architecture which is

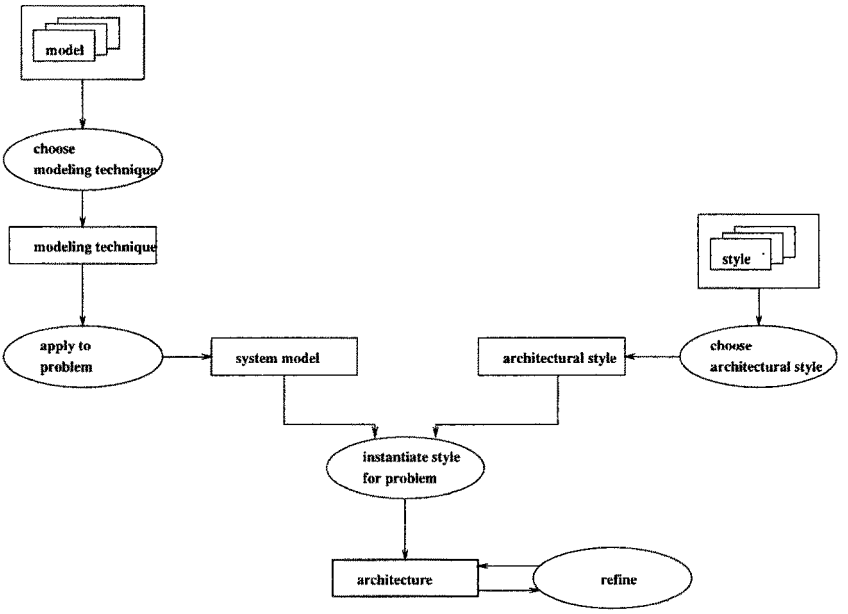

Fig. 1. software architecture in the development process

defined with so much detail that either reusable components and connections can be fitted, or components and connections can be designed and implemented. Note that this view does support a sequential as well as an iterative or an incremental development process, and that in general, the modeling and structuring activities are intertwined. The Software Architecture phase as we view it is depicted in Fig. 1. In this figure, ovals denote activities, while boxes represent products. Input for all activities are the requirements (not shown).

An architectural style is a pattern in the organization of software ([12]), or, somewhat more precisely, "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done" ([11]). Architectural styles are categorized in taxonomies in order to provide guidelines mapping classes of problems onto classes of solutions ([11]).

In [2], Boasson argues that at the highest level, two fundamentally distinct approaches towards software architectures can be discerned: the data-centered and the function-oriented approach. To relate this statement with our view of the Software Architecture phase in the development process, one could say that a data-centered or a function-oriented model each leads to different sets of software architectural styles.

We describe two software architectures on a high level of abstraction, based on two different approaches for system modeling for a (toy) railroad system which

we have available in our software lab. In [1], a data-centered model of the system is given, described in the formal specification language AE-VDM. A function-oriented model of the railroad system, described in PAISLey, is given in [15]. The railroad network described in the case study differs at some points from our toy railroad system, so we had to adapt the models.

For each architecture, we derive the implications they have on those quality properties that are important for a railroad controller:

Timing : The requirements of a real-time system usually contain temporal constraints. In the case of the railway network, there are strict temporal constraints because of safety reasons, and less strict temporal constraints with respect to the schedules. Performance with respect to these constraints can only be measured when all design decisions have been made. It is highly desirable to decrease this gap between temporal constraints and performance at the architectural level. We discuss time aspects of each proposed architecture.

Scalability : Both the toy railroad system and the railway network presented in this case study are scale models of real-life situations. Therefore, scalability is a requirement for a software architecture.

Fault-Tolerance : One of the problems of controlling a physical system is that such a system often does not behave exactly according to whichever model we use to represent it. Reliability addresses the behaviour of a system in an environment behaving according to the model; robustness addresses behaviour of the system in "abnormal" circumstances. Behaviour in abnormal circumstances is often indicated with the term "incident handling". In each architecture, we indicate which changes are needed to incorporate incident handling, in order to achieve a certain degree of robustness. Incidents are not only formed by unexpected events in the railway network, but also by failing communication and hardware.

Extendibility : Requirements are not as static and final as they are usually treated. They change, either as a result of an inaccurate modeling of the environment, or of a changing world. The answer to changing requirements is a changing system. In a system like the railway network presented here, it is necessary to apply changes on-line.

Two possible mechanisms for on-line changes of software found in literature are:

- A change at the architecture level, consisting of adding or destroying components and connections. A model for "dynamic change management" along these lines is presented by Kramer and Magee in [6].
- A change at source code level. Frieder and Segal described a scheme for procedure replacement in [3].

In our opinion, design for change should start at the architectural level. When evaluating an architecture, one should bear in mind that the first mechanism should be applicable in the changes one can think of.

The two architectures are proposed and discussed in Sects. 2 and 3. The last section contains conclusions and suggestions for future research.

2 Data-Centered Approach: A Global State Architecture

The first type of architecture that we analyze is based on a data-centered model of the railroad controller for our toy railroad system, described in detail in [1]. The solution given below is meant as an example of the global state architecture; we don't pretend to propose an optimal solution.

2.1 Event-Action Model

According to e.g. Parnas ([9]), the behaviour of reactive systems can successfully be modeled in terms of events and actions. Events can be defined in terms of changes in the global state of the system, including time. Actions consist of computations resulting in changes in the global state. Similarly, the functionality of the railway system can be described in rules, specifying an action for each discerned event.

In the first place, the speed behaviour of each individual train is modeled by a finite state machine, shown in Fig. 2. In state HALT, a train is stopped (temporarily). State ACC is the state of a (gently) accelerating train; state DEC for a gently decelerating one. A train in state CONST drives with a certain constant speed. A train in state EMERGENCY stops as soon as possible.

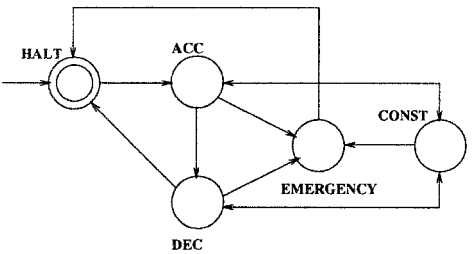

Fig. 2. state diagram for a train

The transitions in the finite state machine are described by action-event rules, stating the events that trigger a transition (Table1). These events involve information about the desired speed for each train, to be generated on-the-fly from the schedule of the train. The "before" column shows the state before the transition; the "after" column the state after the transition; the "event" column describes the event that triggers the transition.

Another finite state machine is used to model the overall behaviour of a train (Fig. 3).

Five states are discerned: in the STATION state, a train is situated at a station; in state START, the route to the next station is (being) determined; in

Table 1. speed of a train

	before	event	after
1	HALT, CONST, ACC or DEC	|desired speed| > | actual speed|	ACC
2	HALT or DEC	desired speed = actual speed = 0	HALT
3	ACC, CONST or DEC	|desired speed| = | actual speed|	CONST
4	ACC, CONST or DEC	|desired speed| < | actual speed|	DEC
5	ACC, CONST or DEC	state of train is Error	Emergency
6	EMERGENCY	actual speed = 0	HALT

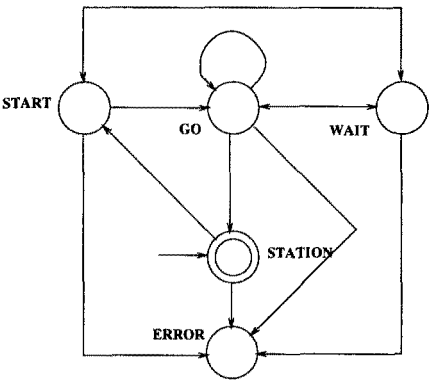

Fig. 3. state diagram for a train

state GO, the train is driving; in state WAIT, the train is stopped somewhere along the route; state ERROR is used for cases of failures. We have assumed that in the initial state, a train is always positioned at a station. The transitions are showed in Table 2.

Table 2. behaviour of a train

	before	event	after
1	STATION	departure time reached	START
2	START or GO	next part of route free	GO
3	START or GO	next part of route blocked	WAIT
4	GO	destination reached	STATION
5	WAIT	next part of route free	GO
6	WAIT	deadlock occurred	START
7	all states	error occurred	ERROR

Data such as the desired speed is set as a side effect of state transitions. Table

3 shows a simple way of setting the desired speed. Other side effects consist of determining the route to be taken, and the setting of switches in the railway network.

Table 3. setting the desired speed

	state transition	desired speed
1	START → GO	desired speed + maximum speed
2	GO → WAIT	desired speed = 0
3	GO → STATION	desired speed = 0

2.2 Software Architecture

Figure 4 shows an architecture, based on this model. A central data store component contains the relevant data and sends events, representing changes in the state or time, to components acting upon these events. These components are able to read and write the data. The proposed architecture can be seen as an instantiation of the blackboard style ([12]).

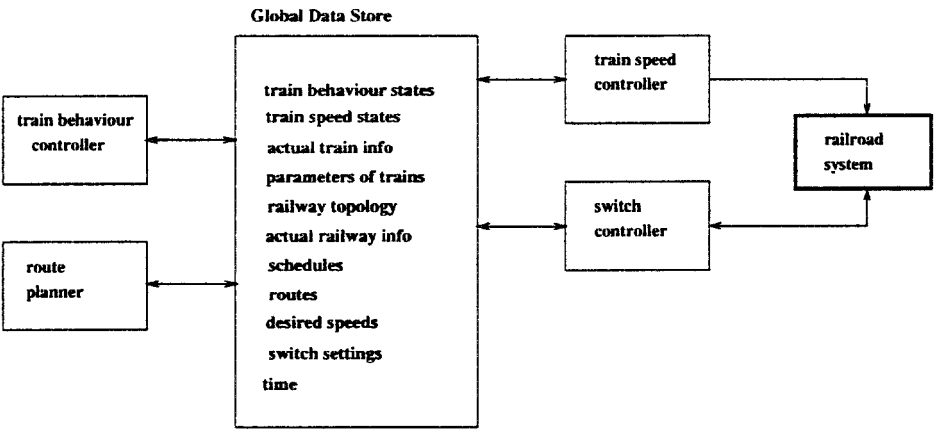

Fig. 4. global state architecture

The Global Data Store: is used to store the global state. Essential of this architecture is the fact that all data are stored globally. As a result, all data needed by components are found in the global data store.

Information kept in the global data store consists of the state of the behaviour and speed of each train, of the actual information (about speed and position) of

each train, the parameters of the trains, the schedules, the derived routes and desired speed, the topology of the railway, the switch settings, and the time.

Certain transitions in the global state represent events.

The Train Behaviour Controller: carries the responsibility of maintaining the finite state machine representation of the behaviour of the trains, according to the rules described in Table 2. The information it needs consists of the train behaviour states, the schedules, time, the positions, speed and directions of other trains, and the train parameters. The component modifies the train behaviour states, the desired speed, and the switch settings.

The Route Planner: is responsible for determining the route to be taken to the next station mentioned in the schedule of the train.

One may implement a deadlock avoiding route planner , or one that does less planning ahead. In the last case, a second task of the route planner is deadlock detection (and consequently determine new routes).

Information needed by the route planner consists of the schedules, the railway topology, and of the positions, speed and direction of the trains.

The Train Speed Controller: is responsible for maintaining the desired speed in a comfortable way, according to the rules in Table 1. Another task of this component is to update information about the actual position, speed and direction of the trains.

Changes in the desired speed for a train, and the transition to behaviour state ERROR, form events of interest for train speed controller.

The Switch Controller: has the task of updating information about the state of the switches and setting them.

Multiplicity of Components: The architecture as it is proposed here does not state anything about the multiplicity of the components. Obviously, there is only one data store. On the other hand, each train might have its own behaviour and speed controller, and route planner. Multiplicity of the switch controller is a possibility as well. Multiplicity of these components is an open design decision in this architecture.

2.3 Implications on Properties

Timing: To be able to analyze the timing behaviour of a system implemented along these lines, the components performing the functionality should be implemented as cyclic, asynchronously communicating processes. These processes poll the global data store to obtain information about the relevant data. Consequently, restrictions to the cycle time can be derived from the temporal constraints and the speed and duration of the connections and computations.

Scalability: In the case that routes for the trains are generated decentrally, on the fly, the possibility of deadlocks is present. With an increasing number of trains driving on a railway network, deadlocks will occur more frequently. The introduction of deadlock avoidance may become necessary, though this will have implications on the timing aspect. However, whether deadlock avoidance is chosen or not, the architecture as we have presented it here suits both solutions. Because the global state contains the data of all trains, a deadlock avoiding algorithm can be introduced very easily.

For scalability reasons, it should be possible to parallelize the computation. As we have seen, the train behaviour and train speed controller can be parallelized (one for each train). The train speed controller can be split into a component maintaining the speed, and a component polling the train for actual speed, position and direction information.

The route planner might be parallelized as well, but in that case, deadlock avoidance is better performed by a separate component. In both cases, computation time increases with an increase of the complexity of the railway network.

Another possible bottleneck is formed by the access on the global data store. The introduction of (parallel) agents detecting changes in parts of the state, and able to read and write data, might be needed with an increase of the railroad system.

Extendibility: Changes in the topology of the network are introduced as changes in the data of the global state. A component, responsible for deriving new schedules, might be introduced. In this case, the timing issue (components should not make use of the new data too soon or too late) is rather trivial: a physical change in the railway network topology will always take place with trains at a safe distance, so the new situation will be read by the controller components by the time that a train has reached a new situation.

Changes in the parameters of trains are to be introduced in the same way, by changing the data in the global data store. When the changes are applied when the train is in state STATION at a station, the new parameters will be used in time.

The same applies for changes in the schedules of trains: the schedule is read by the route planner component when the train is going to leave the station, so the new data will be used on time.

In general, the conclusion is that the proposed architecture is an easily extendible architecture. Data can be changed fairly easy, and an extension of the functionality can be done by adding or changing components, and adding or changing data in the global data store. No big changes in the architecture are needed, because components never communicate directly.

Prerequisites are that changes in the global data store can be applied from outside, and that components and new data and datatypes can be added at runtime.

Fault-Tolerance: A failing train should result in an emergency stop. In our model, this will be effectuated when the event "error" occurs in the global data state

(Table 2). Error-detection might be an extra task of the train speed controller (actual speed differs too much from the expected speed), or by a new component.

Incident handling requires an overall view of the system. In the global state architecture, each component conceptually has such an overview. As a result, one can add components with intelligent incident handling capacities fairly easy.

A failing communication network is another source of problems. We can discern different type of data in the data store: information that is updated frequently, such as the actual position, speed and direction of the trains, and information representing a state, where each change is a major difference with the old data.

The loss of messages containing the first type of data is not really a problem: as long as the time restrictions are not too tight, a decision based on information that is slightly older than it should be will do no harm.

Messages containing information about an event or a state transition may not get lost. A solution might be to handle this kind of information in the same way as continually updated information: instead of waiting for an event, components poll the data in the global data store. Each time when they poll, they update the state information (or data changed as a side effect) in the global data store.

To make the system fault-tolerant, the global data store should be duplicated and/or distributed on different hardware. In the case that all information is stored in the global data store, setting an extra processor with one or more components at work when another fails is easy, because local data don't exist.

3 Function-Centered Approach: A Data Flow Architecture

The second software architecture is loosely based on a PAISLey model for the railroad controller, described in [15]. This PAISLey model is based on two computation models: asynchronously interacting concurrent processes and functional programming. A specification written in PAISLey requires a process structure and a definition of the structure of interprocess communication, and therefore can be mapped almost directly onto a software architecture.

The PAISLey specification of the railroad controller consists of cyclic, asynchronously communicating processes.

3.1 Software Architecture

The dataflow architecture depicted in Fig. 5 is an instantiation of the control-loop architectural style ([10]). The position of the train is the process variable to control. The actual position is compared to the desired position, and differences between them trigger speed or direction commands. The desired position is computed by consulting the route (which contains time information) and the parameters for the train. The actual position is obtained by polling the trains.

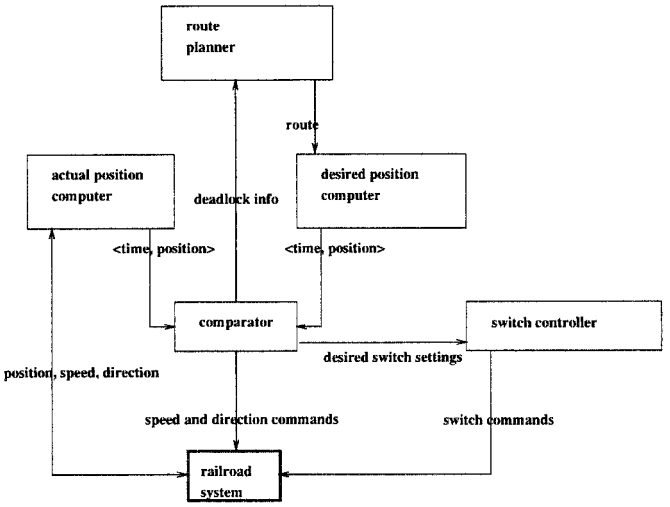

Fig. 5. a dataflow architecture

The Route Planner: computes the route for the trains. Data local to this component are the schedules. In the case of a deadlock, the route planner gets a message from the Comparator, whereupon it computes new routes.

The Comparator: compares the actual and desired position. To simplify comparison, positions are attributed with an indication of time. A difference between positions at a same time indicates the need for control, to be executed in the form of speed and direction commands for the train, and sometimes commands to set a switch.

Information needed by the Comparator consists of the position and speed of all the trains and the state of the switches in the railway network.

The Actual Position Computer: polls the trains and derives <time, position, speed> for the trains.

The Desired Position Computer: derives the desired position for each train from the route, sent by the route planner.

The Switch Controller: keeps track of the state of the switches in the railroad, and sets them according to the messages of the Comparator.

Multiplicity of Components: Both position computers and the comparator in this architecture can have multiple instances, i.e. one for each train. For the route planner , this is less obvious. Introducing deadlock avoidance in a system with one route planner for each train will be difficult, and will require severe

communication between the different components. In a system where the routes for all trains are computed by one component, the information needed to avoid deadlocks is available at the right place.

3.2 Implications on Properties

Timing: The comparator and the switch controller are triggered by both position controllers, which are the "drivers" of the system. From the cycle time of the processes, the speed of the connections and the time needed for different computations, one can analyze whether the system will respect the temporal constraints.

Again, there is the question of deadlock avoidance or detection. Deadlock avoidance should be performed by the route planner. In that case, there should be one route planner for the whole system.

The most logical place for deadlock detection is the comparator, because it receives information about the position of each train. However, the computation time needed for deadlock detection might conflict with the temporal constraints for the comparator. In that case an extra component should be introduced.

Scalability: As we have seen, parallelization can be introduced for both position computers and for the comparator. Inherent to this solution is that the computation time of the comparator increases with an increase of the number of trains: to determine whether a train is able to go on or should stop, the comparator needs information about the speed and position of all trains. An extra component, filtering the relevant information for the comparator, might be needed when upscaling the system.

The same applies for the route planner: its computation time increases with an increase of the complexity of the railway system.

Extendibility: A change in the railway topology network should be applied to data in all four components. Changes in the parameters of a train should be applied in the comparator for the train. New schedules are to be added in the route planner.

In general, because data and computation are intertwined in this architecture, as opposed to the previous one, one should, for each change in data, determine which of the components make use of the information. In the case of an extension of the functionality, one should determine which information is needed for a to-be-added component, and from which components this information can be derived.

Therefore, changes are inherently harder to apply than in the previous architecture.

Fault-Tolerance: A failing train can be detected by the comparator (because of an increasing difference between the actual and desired position). Because the system is based on the comparison between the desired and the actual situation, no extra measures are needed to take failing trains into account. The comparator is the component that has an overall view of the system, because it receives

speed and position information of all trains. However, this component is not the appropriate one to be charged with incident handling, because it should perform under strict temporal constraints. Adding incident handling is another case of adding functionality, and as has been said above, this is less straight-forward in the dataflow architecture as in the global state architecture, because in this case, components communicate directly, and data and functionality are not separated.

The connections between the components are of the dataflow type in some cases (continuously updated information): this is the case for the connection between the railroad system and the actual position computer, and for the connection between both position computers and the comparator. The other connections are used for commands, or for information that is delivered once (a new route, a deadlock situation). When these connections fail to deliver a message, the result may be a disaster. An obvious solution is to deliver these kind of messages multiple times.

Failing processors in this system are harder to replace than in the previous architecture. Every component has local data. The only way to be able to replace a processor is to keep track of these local data on the redundant hardware.

4 Conclusion and Future Work

4.1 Conclusion

Software Architecture in the Development Process: In this paper, we illustrated our view on the software architecture phase in the development process, sketched in Fig. 1, with a case study of a railroad controller in software. Concerning the development process, we can make the following remarks:

- In this case study, the software architecture phase was carried out sequentially: modeling the system took place first, and then a style was chosen and instantiated. The reason for this order was that several models of the system were already available.

  In general, system modeling and the choice and instantiation of an architectural style are carried out at the same time.
- The case study clearly shows the existence of relations between the activities within the software architecture phase. The choice of a model influences the choice of an architectural style, and vice-versa. The adequacy of different approaches toward system modeling for different architectural styles should be added to taxonomies of styles.
- Architectural styles differ more in the degree with which they satisfy the non-functional requirements than the functional requirements.

Implications of Architectural Styles on Non-Functional Requirements: Here, we summarize the effects of the proposed architectures on the quality properties that we found important.

Timing: Whether temporal constraints can be met or not can only be determined in a fully implemented system. In both architectures, we were able to reason under which conditions timing analysis would be possible, and we could reason about possible bottlenecks. At the level of abstraction of both proposed architectures, difference between the solutions with respect to timing issues cannot be found.

Scalability: A big difference between both proposed architectures is that in the global data store architecture, information is always available to every component. As a result, it is easy to divide the functionality of one component between several others. In the dataflow architecture, when breaking one component into several ones, according to functionality or to components of the controlled system, one should always bear in mind how the newly created components get their information.

Fault-Tolerance: Introducing incident handling requires on the one hand the possibility for a component to run in a separate thread, and on the other hand the possibility to gather information about the global state of the system. In the global state architecture, this requires a decision for multiple threads. In the dataflow architecture, multiple threads are part of the style. On the other hand, extending the functionality in this architecture is less easy, because components communicate directly (so one has to determine where the necessary information should be obtained).

Failing hardware is handled more easily in the global state architecture, because the state is always available. The global state itself however, should be duplicated.

Extendibility: Changes of data (topology of the railroad network, parameters of the trains, schedules) are very easy to apply in the global data store architecture. In the dataflow architecture, one should always determine which of the components store such information locally.

In general, changes to the functionality of the system are much easier to apply in the global data store architecture, because there is no need to analyze where the information, needed for each component, is to be obtained.

A requirements for the possibility of on-line changes is that it should be possible to add components, data and data-types on-line.

4.2 Future Directions

On-line system evolution in real-time systems is considered one of the future challenges in this area ([13]). A promising approach would be to explore the possibilities of the global state architecture with this respect. Changes in functionality within this architecture can be applied by adding or substituting components. In addition, facilities to change the global state, and the generation and distribution of events, should be developed.

Even without facilities to change the global state, it is comparatively easy to handle failing processors when using the global state architecture, assuming the

global state component is fail-proof: components performing computation may be substituted by other components without a loss of data.

The description of an architecture evokes a static view of components and connections. Architectures with possibilities for on-line system evolution are dynamic. Apparently, techniques to describe and analyze the dynamics of architectures are lacking. Representation of the dynamics of architectural styles and instantiations form an interesting subject for future research.

References

1. T. Biegstraaten, K. Brink, J. van Katwijk, and H. Toetenel. A simple railroad controller: A case study in real-time specification. Technical Report 94-86, Delft University of Technology, Department of Technical Mathematics and Informatics, 1994.
2. M. Boasson. The artistry of software architecture. *IEEE Software*, 12(6):13–17, November 1995.
3. 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.
4. B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*, 21(4):288–301, April 1995.
5. K. Jeffay. The real-time producer-consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 796–804, Indiana, February 1993. ACM Press.
6. J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.
7. P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(5):42–50, November 1995.
8. R.T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns and objects. *IEEE Software*, 14(1), January 1997.
9. D.L. Parnas, A.J. van Scouwen, and S.P. Kwan. Evaluation of safety-critical software. *Communications of the ACM*, 33(6):636–648, September 1990.
10. M. Shaw. Beyond objects: A software design paradigm based on process control. *ACM Software Engineering Notes*, 20(1), January 1995.
11. M. Shaw. A field guide to boxology: Preliminary classification of architectural styles for software systems. manuscript, http://www.cs.cmu.edu/afs/cs/project/compose/www/html/Publications/1.html, 1996.
12. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
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. A.S. Tanenbaum. *Structured Computer Organisation*. Prentice-Hall, 1976.

15. J. van Katwijk and H. Toetenel. Experience using paisley for real-time specification. Technical Report 95-29, Delft University of Technology, Department of Technical Mathematics and Informatics, 1995.