A comparison of Ward & Mellor's transformation schema with state & activitycharts

Eindhoven University of Technology

Department of Mathematics and Computing Science

A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts

by

J. Peleska, C. Huizing and C. Petersohn

94/11

Computing Science Note 94/11
Eindhoven, March 1994

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. M. Philips
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

A Comparison of Ward & Mellor's TRANSFORMATION SCHEMA with STATE- & ACTIVITYCHARTS

Jan Peleska[1], Cornelis Huizing[2], Carsta Petersohn[3]

September 27, 1993

Abstract

A comparison between Structured Methods, as represented by the Essential Model of Ward&Mellor's Transformation Schemas, and the Statemate specification language consisting of State- and Activitycharts, is presented. The comparison is based on the languages' semantic properties. An example from the field of fault-tolerant systems serves as a "benchmark problem" to investigate the practical applicability of both Transformation Schemas and Statemate in a context of meaningful "real-world" systems. While the article's contents is founded on formal mathematical concepts, its objective is also to reach the software engineers and CASE tool builders who not neccessarily are experts in the field of Formal Methods. Therefore all our results are presented in an informal natural-language style of reasoning.

Keywords: Fault-Tolerant Systems – State- & Activitycharts – Structured Analysis and Design Methods – Transformation Schema – Transition Systems

1 Introduction

In this paper we present a comparison between *Structured Methods (SM)* and the *State- & Activitycharts* specification language, as implemented in the *Statemate* tool (cf. [Ha90], [HPPSS87], [HGdR88]). Out of today's existing SM dialects we focus on Ward&Mellor's *Transformation Schema (TS)*, as introduced in [WM85], [Wa86].

Commercially motivated comparisons of CASE tools often concentrate on the tool interface's ease of use, on the capabilities to generate code from specifications, on configuration management and other features likely to facilitate the industrial software production process. Such comparisons can be rather misleading, if they fail to analyze the differences of the underlying specification methods. They suggest – at least to the inexperienced user –, that a specific specification problem can be properly solved with any method, as long as the CASE tool looks

[1]DST Deutsche System-Technik GmbH, Edisonstraße 3, 24145 Kiel,
e-mail: jap@informatik.uni-kiel.d400.de
[2]Eindhoven University of Technology,
e-mail: keesh@info.win.tue.nl
[3]Computer Science Dept. of the Christian-Albrechts-University at Kiel,
Preußerstr. 1-9, 24105 Kiel, Germany,
e-mail: cp@informatik.uni-kiel.d400.de

good. The experienced developer knows, that the selection of a suitable specification method – i. e. a formal language and an associated semantics – is crucial for every non-trivial software project. Especially in the field of safety-critical systems it is mandatory for the specification language to be sufficiently expressive to cover all the subtleties of the system to be developed and to enable the developer to produce correct abstractions of complex requirements without grossly simplifying the problem. As a consequence, the comparison presented in this article is based on TS and State- & Activitycharts semantics, with an emphasis on the language features neccessary to model parallel systems.

The article is structured as follows: In section 2 we present an informal requirements specification from the field of fault-tolerance. The dual computer system described will serve as a "benchmark" problem for the comparison of Transformation Schema and State- & Activitycharts. The example's significance as a basis for comparison will be motivated.

Section 3 describes the TS solution of the benchmark problem. To this end, subsection 3.1 introduces the graphical TS language and informally sketches the semantics to be used for TS interpretation. Though explicitely intended to support the development of complex command and control systems, Ward&Mellor did not provide a formal semantics for their specification language. However, their informal language descriptions give sufficient indications as to try to *reconstruct* its intended meaning. But on closer inspection, Ward's original ideas about SM semantics (see [Wa86]) are only applicable to describe systems that allow an extremely moderate degree of parallelism. As a consequence, this original semantics is inappropriate to model a solution for our fault-tolerant system. Our presentation of TS semantics is therefore based on research work described in [CAU93], [Pt92], where the mentioned flaws have been overcome by defining a family of operational semantics in the style of Plotkin [Pl83] suitable for a great variety of target systems. Subsection 3.2 motivates the selection of a suitable candidate from this collection to interpret the example's TS solution. Section 3 closes with the presentation of the TS solution.

Also for Statemate's state- and activitycharts, a variety of semantics have been suggested. Section 4 is therefore structured in the same way as section 3 and presents the corresponding State- & Activitycharts solution for our benchmark problem.

Section 5 presents the comparison part, using the previously introduced example specifications to point out the main language differences. Some people regard W&M's method as already outdated and succeeded by Harel' s State- & Activitycharts [Ha88]. This is only partially so. As we argue in Section 6, on the basis of comparing the two methods, W&M's method is probably still the best there is when a lot of data-processing is required, in combination with only mildly complicated control. In case of modelling really complex real-time embedded systems, which do not involve such an amount of data-processing, Statemate is regarded as superior [Wo89]. Moreover W&M's method is the most widely spred CASE method of the SM family in industry, so it makes sense to try and improve it. While their different abilities to express system control are a mere "technical" feature, we also try to point out the more sophisticated differences induced by the underlying semantic models. Here superficially similar State- & Activitycharts and TS Specifications have different meaning with respect to the possibilities of nondeterministic system behaviour, parallel execution of events and fairness properties. Our position is that the decision whether to chose a SM or Statecharts description technique should not be a global one but always be made by analyzing which semantics most closely fits for the problem to be solved.

The present paper has been stimulated by a project at DST to make structured specification

methods like [WM85] automatically analysable and suitable for application in the field of safety-critical systems.

Though the article's underlying concepts are of a formal mathematical nature, our intention is to make this paper readable for a greater audience of CASE experts and CASE tool users, not neccessarily working in the field of formal methods. We have therefore restricted ourselves to natural-language presentation style. For a further exploration of the article's theoretical foundations the reader is referred to [CAU93].

2 Fault tolerant dual computer system – informal requirements specification

In this section we informally describe a typical set of requirements from the field of fault-tolerance. We motivate, that this set can be used as a meaningful example for the comparison of Transformation Schema to State- & Activitycharts.

The objective is to specify a fault-tolerant solution for a computation service P that can be characterized as follows:

- P inputs data provided by a *producer* on channel `A`.
- For each input x on `A`, a computation $y = f(x)$ is performed by P and delivered via channel `B` to a *consumer*.
- We assume a *synchronous communication* between server and environment: The producer will only send a new job after having received a `NEXT`-message from the server computer to indicate that P has finished the last computation.

Next, we describe the boundary conditions for the desired type of a fault-tolerant server platform: The fault-tolerant system shall be designed as a dual computer system `DCP` according to the *master-slave principle*: `DCP` consists of two computers `CP1` and `CP2`. Each of these components may fail independently. As a *fault hypothesis*, we may assume that each computer acts as a *fail-stop component*, i. e. the failure event leads to the computer's total deactivation without any remaining sub-activities. In normal operation (both components available), `CP1` acts as the *master*: a copy `P1` of P runs on `CP1`, producing computations after which a protocol handler of `CP1` requests a new job by means of a message `NEXT1`. `CP2` operates in *standby* mode by only storing jobs in its local memory without activation of a P-copy. Each job is kept by `CP2` at least until the `NEXT1`-message indicates that it has been successfully delivered to the consumer.

If `CP1` fails, this will also be detected by `CP2` which then continues as the master component by activating a copy `P2` of P and producing messages `NEXT2` to request new jobs. Though `CP1`'s failure can occur while a job is still being processed, it is required that this job must not be lost: `CP2` shall use its still available copy of the input and calculate the corresponding result. It must be taken into account, that `CP1`'s failure can occur *after* having delivered a result on channel `B` and *before* having produced the `NEXT1`-message. In such a case it cannot be avoided that `CP2` also processes this job, and the result is sent to the consumer for a second time. To this end, each input is equipped by the producer with an alternating bit, that is also attached

to the result transferred to the consumer. We assume that the consumer has implemented an *alternating-bit protocol* to detect duplicated bits and discard the corresponding results.

Before presenting the Transformation Schema and State- & Activitycharts specifications, it is appropriate to justify why we think that the above example is adequate for the comparison of specification techniques:

- Both Transformation Schemas and State- & Activitycharts claim to be powerful means for the specification of problems in the field of *reactive systems*. DCP presents a typical example of such a system.
- The graphical presentation style of both Transformation Schema and State- & Activitycharts is intended to facilitate a quick understanding of complex specifications. We feel that the DCP specification is sufficiently complicated, so that any proposed solution should be supported by graphical visualization. Experiences with formal specifications have shown, that a pure textual presentation is rather hard to communicate to people unfamiliar with the problem.
- Both specification languages claim to be helpful in the description of problems of a complexity, that makes a natural-language specification unlikely to cover all aspects with sufficient precision. DCP presents a protocol specification problem, where it has to be ensured that certain safety- and liveness-conditions can be guaranteed *"in every possible situation"*. In general – because of its "fuzzy" semantics –, it is very hard to examine natural-language specifications for completeness of case analyses. Therefore it is interesting to see how much help our specification languages offer with respect to completeness checks.
- Problems from the field of fault-tolerance have been widely used to compare the power of formal specification languages and verification methods, since the use of fault-tolerant systems in safety critical applications also suggests formal development techniques to produce trustworthy specification, design and code. Our example allows the comparison of semi-formal specification methods to methods of greater formal rigour (see for example [Pel91], where a CSP solution has been specified, designed and verified for the above example). Moreover, the development of fault-tolerant systems requires that certain design decisions must be incorporated already on specification level. These decisions are related to the concept of hardware redundancy selected (like chosing a dual computer or triple-modular redundancy) and the design-dependent fault hypotheses (like "each computer behaves like a fail-stop component"). They must be shown in the specification, to allow a complete capture of safety requirements. As a consequence, specifications in the field of fault-tolerance do not allow a "pure" top-down approach from specification to design, and therefore represent a significant touchstone both to the specification language and to the developer's skill.

3 A Transformation Schema solution

In this section, a transformation schema specification for the fault-tolerant dual computer system DCP informally introduced above will be given (subsection 3.3). To make this article sufficiently self-contained, we also present an informal introduction of TS syntax and a class

of semantics for their interpretation (subsection 3.1). Ward's original semantics is a member of this class; however, it is inappropriate for the DCP specification. This and the selection of a class member with properties suitable for the DCP interpretation, will be discussed in subsection 3.2.

3.1 Informal description of TS syntax and semantics

Transformation Schema use graphical language elements to describe a system's flow of data and control. Most of these elements can be seen in the diagram in figure 5.

Data Transformations are depicted as solid circles (e. g. RC2, P2) and represent specification parts where the processing of data is described.

Data Flows The data exchanged between the environment and the system and between different data transformations inside the system travels on *data flows*, depicted as labelled solid-line arrows (like ACP2). Data flows only carry typed information. New types can be constructed from atomic types using constructors, such as cartesian products (denoted as A + B) and union (denoted as [A | B]). Consuming the data item travelling on a flow is a *destructive* operation. Therefore the value $\perp$ extends each flow's type to indicate that no data is presently available. Flows of product or union type can *diverge* to feed their components into different transformations. For example flow R2 in figure 5 splits into A2_X, A2_B and the control flow wrA2. If a value is placed ob R2, the corresponding components are placed on A2_X and A2_R at the same time. Moreover, flows emerging from different sources can *converge* to be fed into a single sink.E. g. BCP2 is composed of Y2 + A2_B. As soon as values have been placed on *both* Y2 and A2_B, the corresponding composed value is defined for BCP2. If two flows A, B converge into a union flow C = [A | B], a C-value is defined, as soon as a value is placed on *either* A or B.

Stores To introduce data containers for *non-destructive* read operations, the *store* symbol is used (solid-line labelled open boxes, e. g. S2).

Control Transformations are depicted as dashed-line circles and represent specification parts where the system's dynamic behaviour is controlled (e. g. CCP2).

Control Flows carry the control information exchanged between the environment and the system and between different transformations inside the system. They are represented by dashed-line arrows (e. g. NEXT1). In contrast to data flows, each control flow only has a finite range of allowed values (including $\perp$). Each control flow must start or end in a control transformation. If it is generated by data transformations, it is called a *data condition*. There are specific control flows having data transformations as sink. They are used to *trigger* (e. g. TRIGGER P2), to *enable* (e. g. ENABLE P1 in figure 3) or to *disable* (e. g. DISABLE P1 in figure 3) the data transformation. A trigger leads to a single execution of the associated data transformation; after having produced its output it will wait for the next trigger event. Triggered data transformations do not input from flows, they only use stores. Disabling a

data transformation will prevent it from producing any outputs until occurrence of the next enable event.

Process Specifications are textual descriptions associated with each data transformation to define the relation between input data and output data.

State-Transition Diagrams describe the transformation of control items arriving on input control flows into output control flows by means of Mealy-style automatons (e. g. figure 6). Incoming events passed via input control flows can be processed, if the automaton's actual state defines a corresponding transition. In this case the transition will be performed. At the same time, new actions are generated by placing values on output control flows. For example in figure 6, ocurrence of event `wrA2` in state `ACTIVE-NO_DATA` will lead to the automaton's transition into state `ACTIVE-DATA` and produce the action `TRIGGER P2`.

Scope Rules Each process specification or state-transition diagram can reference only flows and/or stores connected as input to the corresponding transformation and write only onto flows and/or stores connected as outputs.

Top-Down Presentation of TS To reduce the complexity of large specifications, a TS can be presented in *top-down* fashion. For the `DCP` specification, such a top-down presentation is given in figures 1 to 6. The top-level diagram (figure 1) shows the complete system as a black box (bubble `DCP`), together with its interface to the environment (flows between `DCP` and the *terminators* `PRODUCER`, `CONSUMER`, `FAILURES`). The top-level bubble is refined by a new TS, in our example given by figure 2. Each bubble on the new TS can either be associated with a process specification or again be decomposed into another diagram, until the associated transformations are sufficiently small and cohesive to be represented by a single process specification. However, the behaviour of a TS is completely defined by the *flattened* TS associated with each top-down presentation, where all the transformations, stores and flows are "glued" into a single diagram by connecting the corresponding flows.

TS Semantics of Dynamic Behaviour The behaviour of a TS is most appropriately interpreted in the framework of *transition systems* as defined in [MP92]. The *statespace* of a TS is defined by the cartesian product generated by the sub-statespaces of flows and stores, by the states of data transformations (state values `ENABLED/DISABLED`) and the states of the automatons associated with control transformations. The *initial condition* of a TS requires that all flows assume the "clear" state $\perp$ and all state-transition diagrams assume their initial state (specified by the transition arrow without source state). The *transitions* of a TS are defined by the data and control transformations. A *TS execution* is a sequence

$$s_0 \rightarrow \tau_0 \rightarrow s_1 \rightarrow \tau_1 \rightarrow s_2 \rightarrow \ldots$$

where each s_i is a state and each τ_i is a transition enabled in state s_i, such that (s_i, s_{i+1}) is a possible pair of pre-state and after-state of τ_i. s_0 must be consistent with the initial condition.

In the context of transformation schemas, each $s_i \rightarrow \tau_i \rightarrow s_{i+1}$ is called a *micro step*. Informally speaking, micro steps represent the system's internal processing steps, that are not observable by the environment. They are atomic and happen in "zero-time" (see [BB92]). *Macro steps* define sequences of micro steps that always terminate with a step $s_{n-1} \rightarrow \tau_{n-1} \rightarrow s_n$ allowing in its after-state s_n the placement of new input data on the system's interface. Macro steps are interpreted as the observable interactions between system and environment.

Given the TS syntax as introduced above, *micro step rules* define the set of possible transitions τ_i with their pre-state/after-state relations. *Macro step rules* restrict the space of possible TS executions by imposing additional "global" conditions for a micro step to be executed. These conditions depend on the complete sequence of micro steps already executed during the macro step. Each set of micro step rules plus macro step rules defines a new type of transition system, each operating on the TS statespace. Therefore these sets of micro step/macro step rules define a family of semantics for Transformation Schemas. This approach has been followed in [CAU93].

The family of semantics introduced in [CAU93] has the following members:

Causal-Chain Semantics This semantics most closely reflects Ward's original ideas described in [Wa86].

Each micro step is defined by execution of either a complete data transformation or a complete control transformation. A transition defined by a data transformation is enabled according to the micro step rules, if its input flow or trigger has a value $\neq \perp$, and if the output flow (if existing) has value $\perp$. The latter condition means that the output flow's consumer must have consumed any data item previously transmitted via this flow. As a result of the data transformation the input flow has value $\perp$ and the output flow and/or output stores have values defined by the corresponding process specification's input/output relation. If the data transformation has been previously disabled by a control flow, the transition results only in resetting the input flow to $\perp$. A transition defined by a control transformation is enabled, if a value $\neq \perp$ is placed on a control flow. Moreover, the control transformation must be either in a disabled state or the associated state-transition diagram must be in a state where this event stimulates a transition. As a result the state transition is performed, and the associated output events are placed on the output control flows. If the control transformation is disabled, the input is discarded.

A macro step is always initiated by an input on the system's interface. It will then be processed in *causal chains*: In such a sequence of micro steps, the output of one transformation causes the activation of the corresponding consumer transformation. If more than one input is placed at the macro step's beginning or a transition enables more than one successor, the next transition is chosen nondeterministically. If the causal chain cannot be prolonged, because the last transformation has produced an output to the environment or the consumer transformation needs additional data to operate, a new chain is activated starting with any of the transitions enabled according to the micro step rules. The macro step terminates, if no more enabled transitions are left. If any flow values have not been processed at the macro step's end, they are reset to $\perp$. New inputs at the system's interface may only be placed at a macro step's beginning.

Weakly-Fair Interleaving Semantics This semantics has the same micro step rules as the causal-chain semantics. The macro step rule has the same initiation and termination rules. but it drops the causal chain condition by allowing *any* enabled transition to be taken for each micro step. The semantics' name is motivated by the fact that non-diverging specifications automatically possess a weak fairness property. This will be discussed in section 5.

Full Interleaving Semantics This semantics has the same micro step rules as the two above, but it drops the input rule by allowing new inputs from the environment to be placed and processed at each micro step. Therefore macro steps are identical to micro steps and there is no situation, where data resp. control signals placed on flows is discarded.

CSP Semantics This semantics has been introduced in [Pel93], by giving translation rules of Transformation Schemas into CSP ([Ho85]). It introduces "shorter" micro steps by separating each transformation's input and output phase into two transitions, where other micro steps may interleave. The macro step rules are the same as in the full interleaving semantics.

3.2 Selection of an appropriate TS semantics

In this subsection we motivate, which of the TS semantics available is the most appropriate to interpret a TS specification of the DCP-problem. Informally speaking, we are looking for a semantics that allows to model all critical aspects of the system, but is not unneccessarily complex by allowing executions of the specification that are irrelevant or even impossible in the target system.

One of the crucial difficulties when developing fault-tolerant systems lies in the fact that it is not possible to influence the point in time when a failure happens. In terms of our example, this means that occurrence of CP1's failure event CRASH1 must also be considered for situations when a job is just being processed by P1.

We first motivate, why the causal-chain semantics introduced above is inadequate. Using transformation schemas, CRASH1 will be modelled as an external event (cf. figure 7), since it happens spontaneously and independently on DCP's internal state. Therefore investigating the consequences of a CRASH1 occurrence means applying the execution rules of TS semantics from a pre-state, where a token is placed on the control flow CRASH1 on DCP's interface. But even if an input token is placed at the same time on data flow A, the causal-chain semantics offers no possibility to investigate the impact of CP1's failure while a job is being processed: The semantics' execution rules specify, that in the situation described either the impact of the A-input or that of the CRASH1 event is completely processed in its corresponding causal chain before the other token is taken into account. This means, that only the situation "failure occurs *after* the last job has been completely carried out and *before* the next A-input arrives" can be analyzed in the causal-chain semantics.

The full-interleaving semantics is not selected, because it requires that the developer explicitely expresses all fairness properties needed in the specification. We will see in the TS solution presented below, that a certain fairness property is a "natural" requirement for the DCP specification. Therefore it would be an advantage to have this property automatically contained in the specification, without making it explicitely visible.

Though the CSP-based semantics introduced in [Pel93] would be an appropriate candidate to model the DCP system, it introduces an unneccessary degree of complexity for this application. First, fairness properties have to be explicitely expressed, just as in the full-interleaving semantics. Second, the higher degree of interleaving offered is not required for the DCP problem. For a data transformation P the CSP semantics allows to model situations like *"CRASH1 appears after P has input x and before it could produce output y."* It will become apparent that for our example this situation can be "approximated" by the event sequence *"CRASH1 occurs before P inputs x"*.

This leaves us with the weakly-fair interleaving semantics as a well suited candidate for a TS specification of the DCP. The solution for DCP and the weakly-fair interleaving semantics' appropriateness will be discussed in the next subsection.

3.3 Dual Computer System – TS solution

The TS solution for DCP is presented in figures 1 to 6. Note that our specification of CP1 and CP2 is asymmetric, as shown in figures 3 and 5, because we only wish to specify DCP's behaviour in the situation where CP1 fails, while it acts as master. A symmetric specification could also cover the cases where repaired components are re-integrated into the system. However, this would make our example unnecessarily complicated without adding new insight for the comparison between TS and Statecharts.

Description of interfaces The interface of DCP in the TS model is shown in figure 1. New jobs for the service P implemented in DCP are input on data flow A, the results are delivered on data flow B. Control flow NEXT1 is defined as `NEXT1 = ["OK"|"WATCHDOG ALARM"]`. The producer is notified via signal `NEXT1 = "OK"` (or NEXT2, if CP2 acts as master) that DCP is ready to accept a new job. (The meaning of `"WATCHDOG ALARM"` will be explained further below.) The occurrence of CP1's failure (we do not consider the crash of CP2) is modelled by means of the control flow CRASH1.

The diagram in figure 2 shows the interfaces between CP1 and CP2 and the outside world: Each input on A is broadcasted on ACP1 and ACP2 to CP1 and CP2, respectively. The new names ACP1, ACP2 have only been introduced to distinguish CP1's and CP2's input flows. Formally, A is defined as `A = ACP1 + ACP2`, and both ACP1 and ACP2 carry the same value.

Output B has structure `B = [BCP1 | BCP2]` and passes values from either BCP1 or BCP2 to the environment. `NEXT1 = "OK"` is both sent to the environment and also to CP2 to indicate that the last job has been successfully terminated. CRASH1 is consumed by CP1; after its occurrence CP1 stops processing A-inputs. As a reaction to CRASH1, the signal `NEXT1 = "WATCHDOG ALARM"` is transmitted. This can be interpreted as a watchdog mechanism that controls CP1, detects occurrence of the failure CRASH1 and passes the alarm message on flow NEXT1[4]. As a reaction to this signal, CP2 will take over as the master component. (The environment will simply discard this signal.) After occurrence of CRASH1 and `NEXT1 = "WATCHDOG ALARM"`, CP2

[4]To motivate that it is reasonable to specify an alarm signal that still can be generated after CP1 has crashed, think of an independent hardware device like a watchdog observing CP1's local bus. If CP1 fails, this will not affect the watchdog, so it can detect that no messages pass CP1's bus anymore and signal the alarm message to CP2. On our specification's level of abstraction, it makes sense not to show the watchdog explicitely, but simply demand a design that is able to signal CP1's failure to CP2.

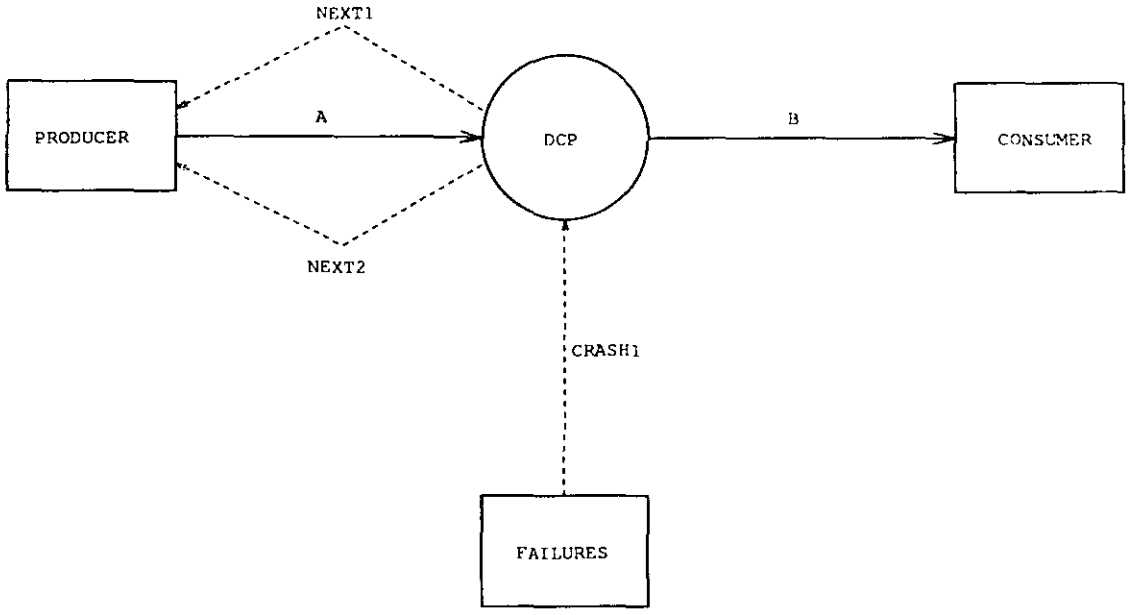

Figure 1: Dual Computer System DCP - TS specification of the interface.

will start to produce outputs on BCP2 and request new jobs via NEXT2. The alarm message "WATCHDOG ALARM" is transmitted for reasons of proper sequencing on the same flow as the request "OK" for new inputs. This will be further motivated in section 5.

Behaviour of CP1 Figure 3 shows the internal structure of CP1. Its behaviour is controlled by control transformation CCP1 ("Control CP1", figure 4). At system startup, CP1 assumes state UP, and data transformation P1 is enabled. Additionally a NEXT1 = "OK" message is sent to the environment to indicate that DCP is ready for data processing.

An input on ACP1 is split into the components A1_X carrying the data to be processed and A1_B carrying the bit that alternates with each job. Data transformation P1 consumes A1_X and computes the result Y1 := f(A1_X). This result is combined with signal wrB1 ("written B1") on output flow Y11.

The Y1-component is combined with the alternating bit A1_B, so that BCP1 = Y1 + A1_B carries both the result calculated by P1 and the unchanged bit received with the calculation's corresponding input. The signal wrB1 is passed to CCP1 to indicate that the job has been delivered. By definition of our semantics, the data transfer on BCP1 and the placing of the wrB1 signal happen in the same micro step that includes the output production of P1 on Y11. On reception of the wrB1 signal, CCP1 outputs the NEXT1 = "OK" message. The state UP is kept until occurrence of event CRASH1. Then CCP1 performs the transition into the final state DOWN, at the same time producing signal NEXT1 = "WATCHDOG ALARM" and disabling P1, so that CP1 is completely deactivated.

The weakly-fair interleaving semantics adequately models the situation when a new input A and the failure CRASH1 occur "at the same time", i. e. in the same macro step, as it is observable by the outside world. If at the beginning of a macro step tokens are placed both on A and CRASH1, the impact of the failure event can interleave in three places:

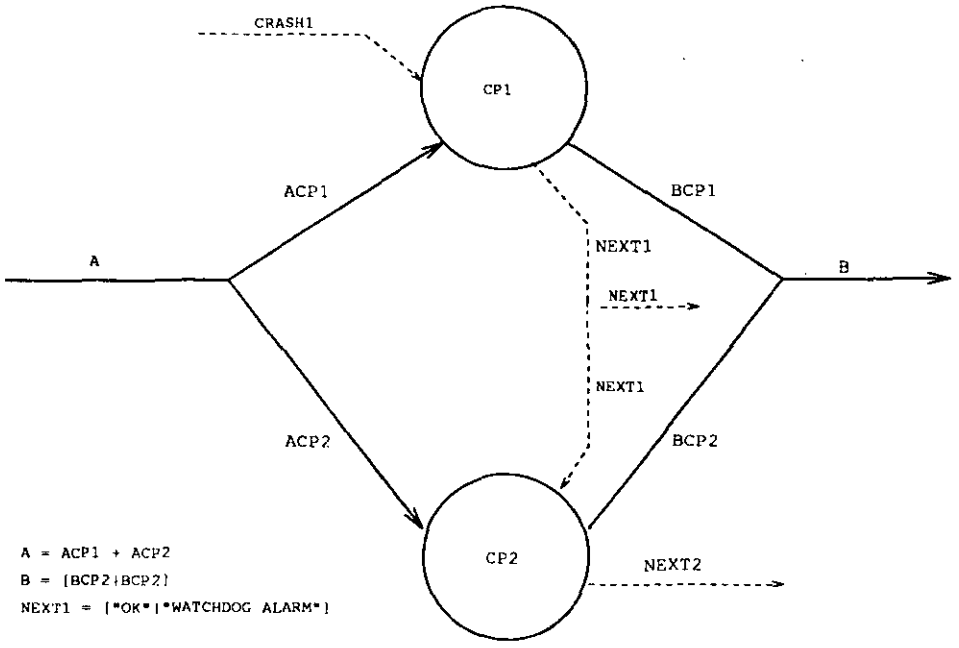

Figure 2: Dual Computer System DCP – TS specification of the interfaces of computers CP1 and CP2.

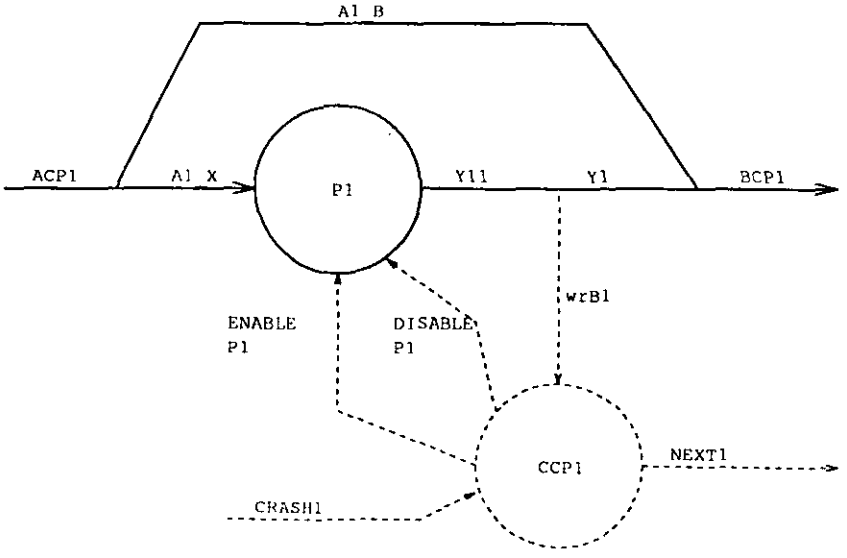

Figure 3: Dual Computer System DCP – TS specification of computer CP1.

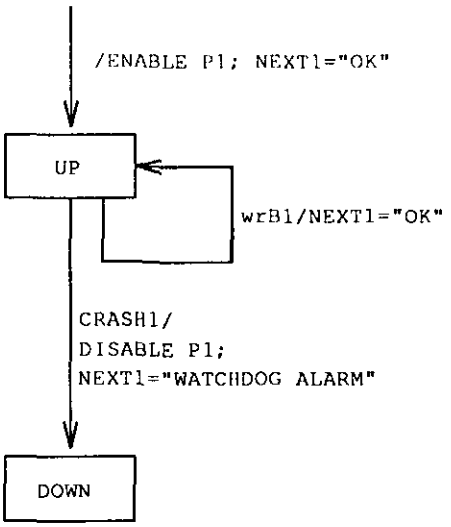

Figure 4: Dual Computer System DCP – Control transformation CCP1 of computer CP1.

1. before P1 consumes the A1_X-input
2. after P1 has placed the output, but before the NEXT1 is produced
3. after NEXT1 has been produced

The "most complicated" second case, making the introduction of an alternating bit protocol necessary, corresponds to a TS execution, where event CRASH1 is already placed on CCP1 but has not yet been consumed, the result has been delivered on BCP1, so that wrB1 is also placed on CCP1, and the next micro step starts by processing the CRASH1 token.

The fairness property that every token that *can* be permanently processed during one macro step *will* be processed during this step ensures that CP1 will always react to the placement of the CRASH1 token during the actual macro step. If this condition would be dropped, CP1 could "ignore" the occurrence of the failure and continue processing new jobs. This would certainly not be an appropriate model for a failure event, that typically has just the unpleasent property that its impact cannot be postponed.

Note that we also have to make use of the condition that unused tokens are discarded at the end of each macro step: In state DOWN, a new input on ACP1 is split into A1_X and A1_B. The A1_X-token is thrown away, because P1 is disabled, but the A1_B-token remains unconsumed until the macro step's end. Now our semantics for diverging flows (in our case ACP1, A1_X and A1_B) demands that all outgoing branches must be cleared, before a new token can be placed on the flow. Therefore ACP1 would be blocked and as a consequence also prevent new messages from being placed on the input flow A, if the unused A1_B-token would not be discarded at the macro step's end.

Behaviour of CP2 Figure 5 shows the internal structure of CP2. Its behaviour is controlled by control transformation CCP2, shown in figure 6. At system startup, CP2 assumes state PASSIVE-NO-DATA. A receive process RC2 consumes new jobs arriving on ACP2. Using output flow R2 = A2_X + A2_B, it then places the job's data component A2_X in store S2 and

the alternating bit on flow A2_B, at the same time signalling the job's arrival to CCP2 via wrA2. CCP2 then assumes state PASSIVE-DATA. Occurrence of the signal NEXT1="OK" indicates that the job has been successfully completed by CP1. Therefore CCP2 again assumes state PASSIVE-NO-DATA. The remaining token on A2_B is discarded at the end of the macro step.

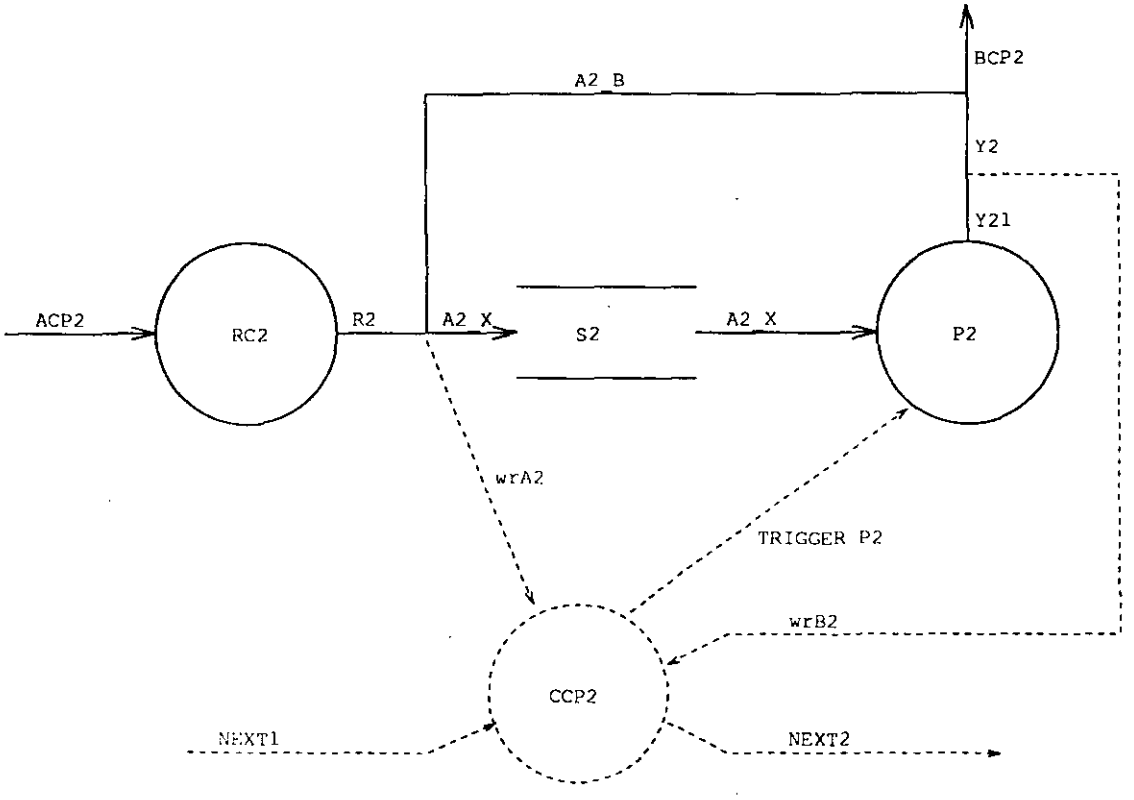

Figure 5: Dual Computer System DCP – TS specification of computer CP2.

If the failure of CP1 is indicated via NEXT1="WATCHDOG ALARM", two cases must be considered:

1. CP2 *is in state* PASSIVE-NO-DATA: In this state it is ensured that no job was being processed when CRASH1 occurred. Therefore CP2 will simply assume state ACTIVE-NO-DATA. At the next arrival of a job on ACP2, the corresponding signal wrA2 will cause CCP2 to trigger P2 and perform the transition into state ACTIVE-DATA. Placement of wrA2 is performed in the same micro step where the new input data is placed in store S2. Therefore P2 always finds the actual input data in S2, when the trigger leads to P2's activation in the subsequent micro step. P2 reads input data A2_X from the store and produces the output Y21 = Y2 + wrB2, where Y2 = f(A2_X) carries the calculated result that is combined with the bit A2_B on BCP2 = Y2 + A2_B, and wrB2 indicates that the output BCP2 has been delivered, so that CCP2 can produce the required NEXT2 signal and return into state ACTIVE-NO-DATA.

2. CP2 *is in state* PASSIVE-DATA: In this situation, CP2 has to reproduce the job stored in S2, because it is uncertain, if CP1 could deliver the result before the failure happened. CCP2 therefore directly triggers P2 and performs the transition into state ACTIVE-DATA. Afterwards it exactly operates as described in the first case.

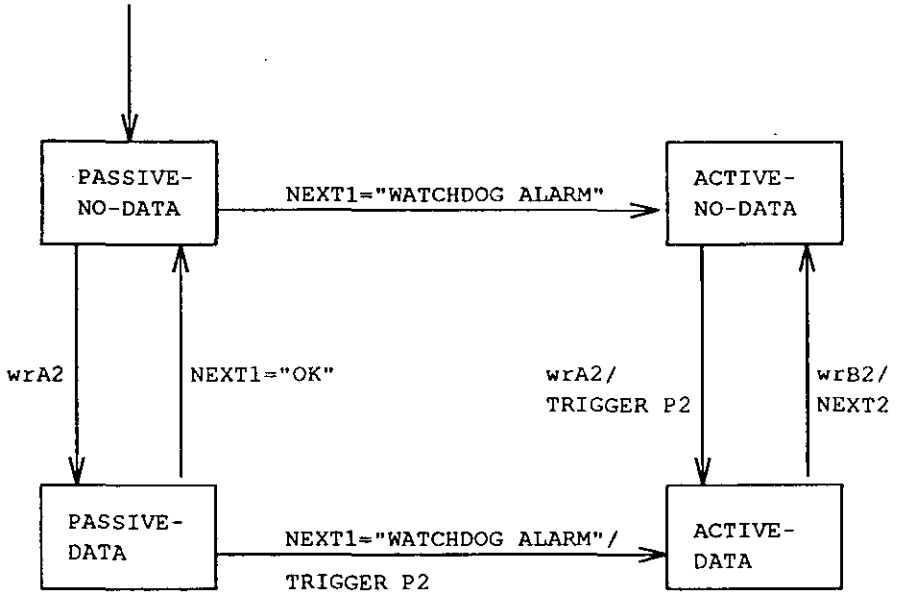

Figure 6: Dual Computer System DCP – Control transformation CCP2 of computer CP2.

4 A State- & Activitycharts solution

In this section, we present a State- & Activitycharts specification of DCP. As in the previous section, a short introduction into the State- & Activitycharts specifiction language is given in subsection 4.1. For a more detailed description, the reader is referred to [Ha90], [iL89].

4.1 Informal description of State- & Activitycharts syntax and semantics

In general, a State- & Activitycharts specification consists of two parts. The first part, called *activitychart*, describes the conceptual structure of the system, showing the flow of information and control between the components. The second part, consisting of one or more *statecharts*, specifies the actual behaviour of these components, i.e., how the information flow is processed.

Activitycharts In figure 7, we see the activitychart of the Dual Computer. It consists of one root activity DCP, which is connected by the environment activities PRODUCER, CONSUMER, and FAILURES. DCP consists of two subactivities, CP1 and CP2. In general, this nesting of activities can be applied to any depth. The behaviour of an activity can be specified with a statechart, as shown for CP1 and CP2 which are controlled by two statecharts CCP1 and CCP2 in figures 8 and 9. Activities that are not further refined by sub-activities and/or statecharts represent processes that perform transformation of data without controlling any other components. In our example, P1 and P2 are such transformation processes (we left the actual transformation unspecified, because its definition is not relevant in our context). All activities may be self-terminating or be terminated by a statechart (see below) via special *stop*-events. Furthermore, the execution of an activity may be temporarily suspended and later be resumed by sending it specific events.

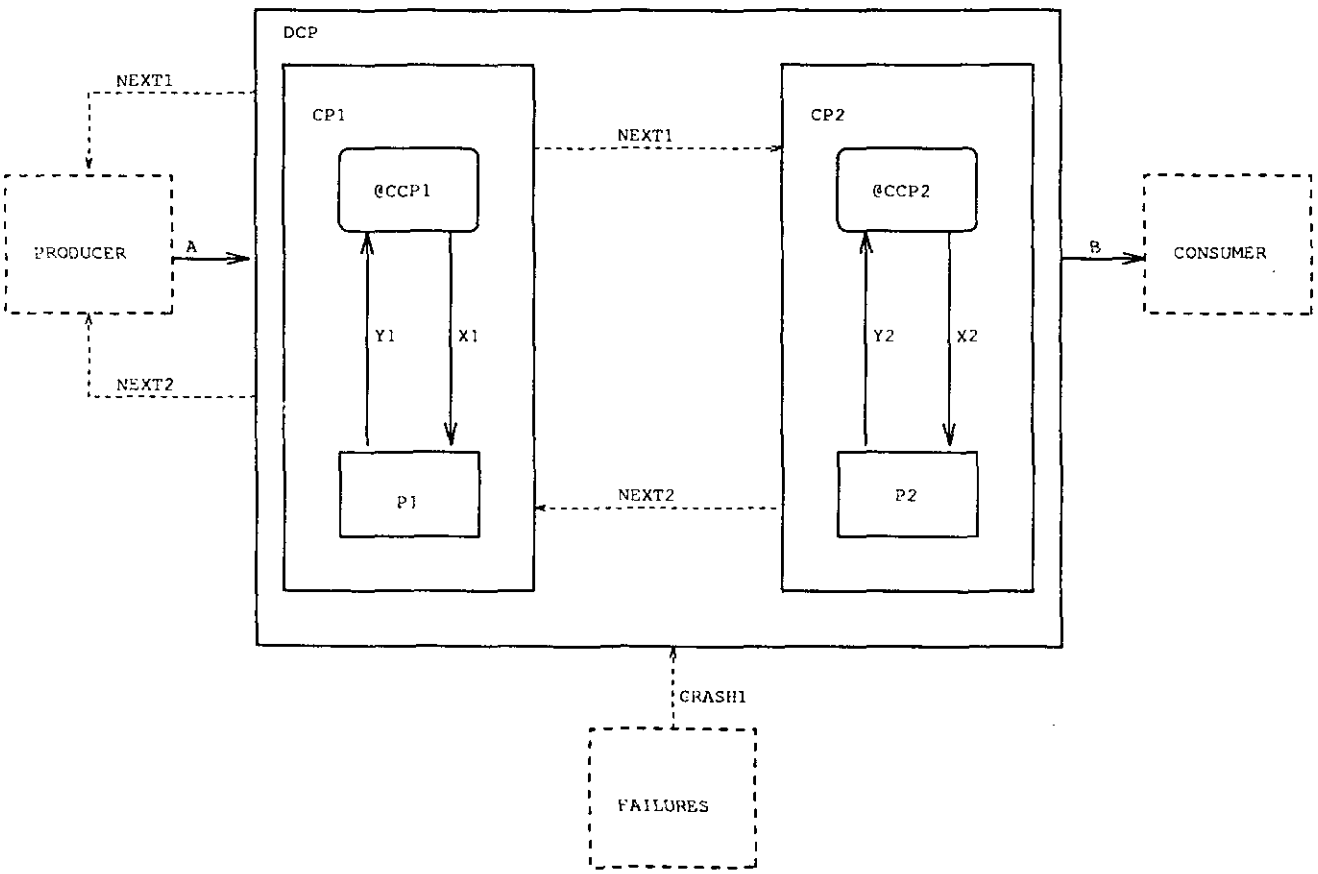

Figure 7: Dual Computer System DCP – Activitychart.

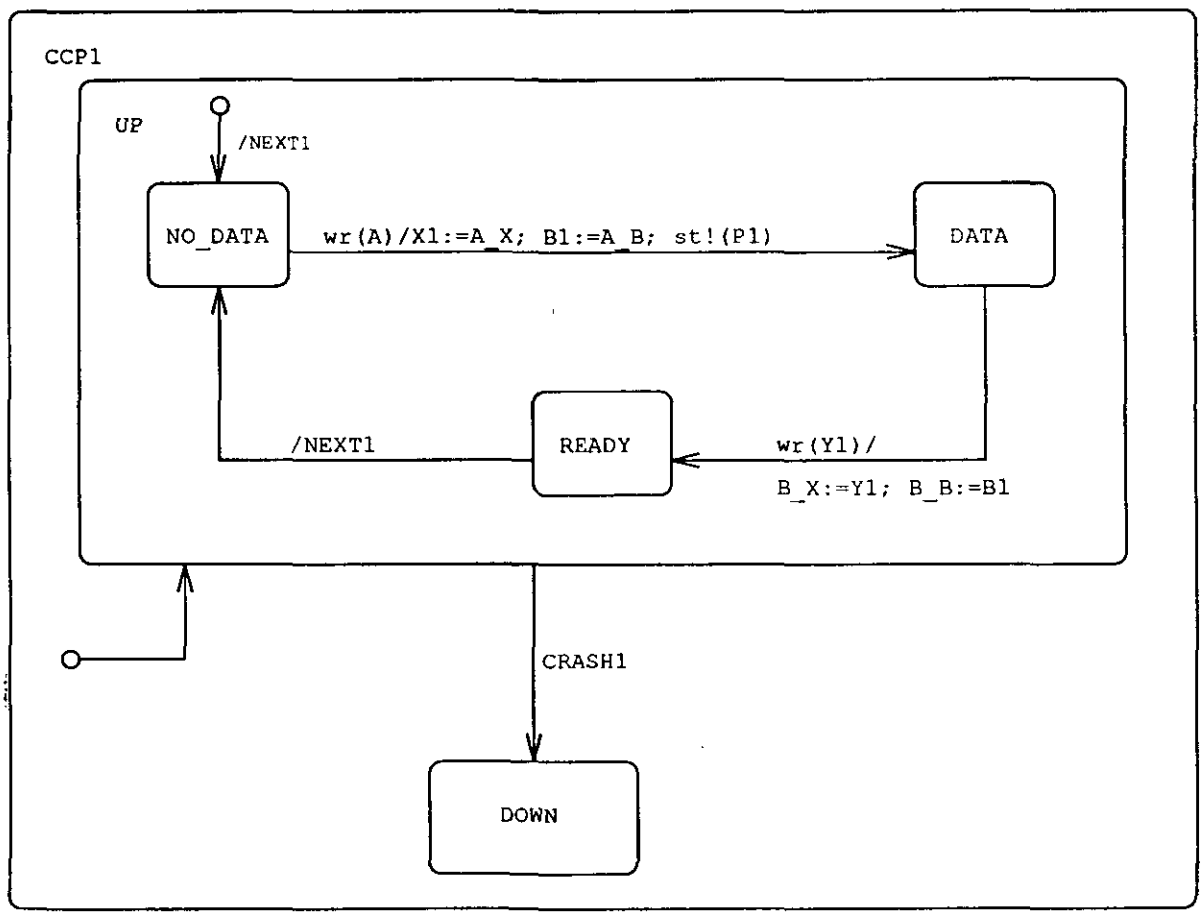

Figure 8: Dual Computer System DCP – Statechart CCP1 of computer CP1.

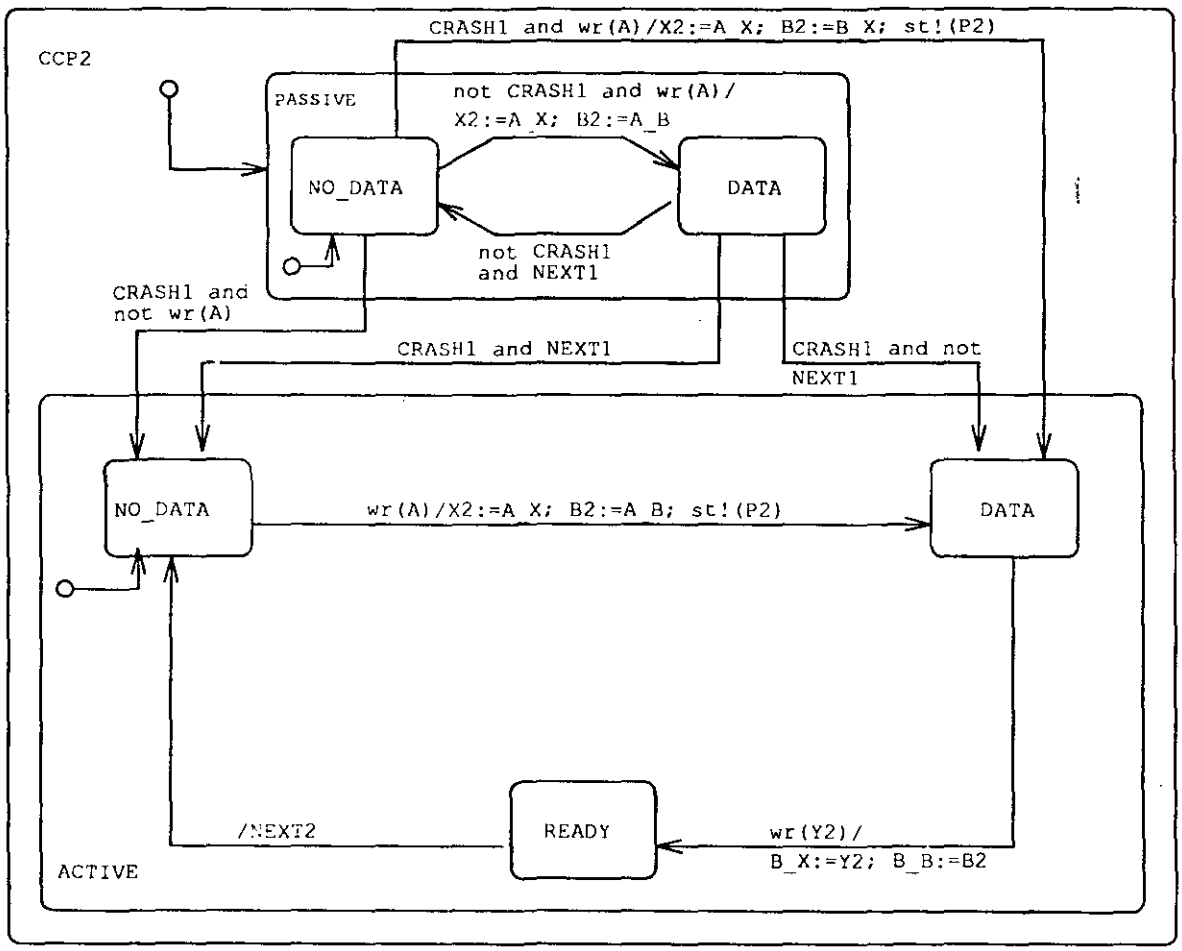

Figure 9: Dual Computer System DCP – Statechart CCP2 of computer CP2.

Statecharts The Statechart formalism is derived from the finite state diagram. *States* are denoted like rounded cornered boxes and connected by arrows called *transitions*.

The *label* of a transition has two parts, separated by the symbol /: an *enabling part*, or *trigger*, and an *action* part.

The trigger part specifies under which conditions the transition can be taken. To this end, the trigger is defined by means of an *event expression* plus a boolean expression ranging over variables and states. *Events* are the basic means to mediate control, they can be interpreted as atomic signals that are only visible "for one moment of time" (this will be explained in more detail below). In our DCP example, the events are are `CRASH1`, `NEXT1`, `NEXT2`, to be interpreted as signals exchanged between the environment and the system, and `wr(A)`, `wr(Y1)`,... (*"written A"*), to be interpreted as signals internally generated by DCP, as soon as new data is written on `A`, `Y1`,.... Event expressions are defined by boolean expressions ranging over events. In figure 9 for example, the event expression `CRASH1 and not wr(A)` is interpreted as *"the corresponding transition may be taken, if event* `CRASH1` *is signalled, and at the same time no data is written on* `A`*"*. If an additional expression over variables and states is associated with the trigger, the corresponding transition is enabled, as soon as both expressions evaluate to TRUE. For example, a transition with the trigger

```
a and not b[x=5 or not in(DATA)]
```

can be taken the moment that the event a occurs and b does not and at the same time the value of variable x equals 5 or the system is not in the state `DATA`. When a transition has an empty trigger, such as in the transition from `READY` to `NO_DATA`, it can be taken immediately and unconditionally.

When a transition is taken, its action part specifies a list of atomic actions generated by the transition. These actions may consist of a generation of new events that can be sensed by other tansitions' triggers and the environment. Furthermore, they can define assignments to varibles, and specific actions can be defined to control (i. e. start, stop, suspend etc.) activities. In figure 8 for example, the action `st!(P2)` is interpreted as an activation of activity `P1`, as soon as the corresponding transition is taken.

States Unlike ordinary Finite State Diagrams, states are boxes and may contain subcharts, i.e., specifications of state machines that are to be executed when the system is in the surrounding state. When the surrounding state is entered, the subchart is started in the initial state, designated by a bullet-tailed arrow and when it is exited, the execution inside is aborted. For `CCP1`, this means that `UP` is started at the beginning and consequently `NO_DATA` is entered. Depending on the occurrence of the events `wr(A)` and `wr(Y1)`, the system will cycle through the three substates of `UP`, until the event `CRASH1` occurs. This will move the system immediately to the state `DOWN`, irrespective of which state in the cycle it occupies. Notice, however, that transitions are considered atomic, so an action is always completed and the occupied state is always defined.

Variables Data that should persist over time can be stored into so-called *data-items*, which are basically program variables (e. g. `X1`, `B1` in figure 8). They can be integers, reals, booleans (called conditions), or strings, and compositions of these, such as records and lists. Data-items can be changed by the action parts of transitions using ordinary assignments. The

lifetime of a data-item is limited by the existence of the activity to which it belongs. If they should live independent of activities, they should be explicitly introduced in a so-called data store.

State- & Activitycharts Semantics of Dynamic Behaviour The dynamic behaviour of a State- & Activitycharts specification is completely controlled by its statecharts. A variety of semantics has been defined over the last five years (see [HG89], [HG92], [KP92], [iL89]). We give a short summary of these definitions.

As introduced for Transformation Schemas, we can also define *micro steps* and *macro steps* for the Statecharts semantics introduced in [iL89] and by Huizing and Gerth [HG89]. A micro step is executed in three phases:

1. All input events are evaluated in order to decide which components are enabled to perform a transition.
2. All enabled parallel components perform their transitions *in parallel*; there is no interleaving. If two transitions write in parallel on the same data item, this conflict is resolved nondeterministically ([iL89, p. 2-60]), i. e. it is unpredictable whose write operation will be effective at the micro step's end. However, the developer is encouraged to write only specifications where such *racing conditions* cannot occur.
3. The actions triggered by the transitions performed are collected and made available for the next micro step. This means, that a component cannot sense events generated by any other component during the same micro step.

[iL89] describes the semantics actually implemented in the Statemate tool's simulation component:

"Go Repeat" Simulation In this regime, events live only for the duration of the micro step directly following their generation. A macro step (called *"super step"* in [iL89]) consists of a maximal sequence of micro steps: The output of the first micro step is evaluated as the input set of the second micro step and so on, until the last micro step's output does not enable any additional transition. New inputs at the system's interface can only be given at the beginning of a macro step and they are only visible in the first micro step. Data inputs live for the duration of a complete macro step, and reading them is non-destructive.

"go step" Simulation The lifespan of events is defined as in the GO REPEAT semantics. But in this regime, a macro step coincides with a micro step. Therefore new data and events can always be input on the system's interface as soon as a micro step has terminated.

In [HG89] five Statecharts semantics (*A-semantics,...*, *E-semantics*) are introduced. The A-semantics corresponds to the GO STEP semantics. The other four semantics differ from the GO STEP/GO REPEAT models in the way they handle the observability of events during each macro step. Specifically, each of the [HG89]-semantics assigns a lifetime of a complete macro step to input events placed on the system's interface.

The semantics introduced by Kesten and Pnueli in [KP92] strongly differs from those sketched above: Here the notion of micro- and macro steps is dropped, instead the possible transitions are classified as *untimed* and *timed* transitions. Untimed transitions consume and manipulate data and events. In contrast to the semantics above, their execution does not affect the lifetime of events. Furthermore, parallel untimed transitions are executed in an interleaved mode. Timed transitions synchronously advance the clock and terminate the lifetime of existing events. Consecutive untimed transitions are interpreted to happen "in the same time interval" defined by the surrounding timed transitions.

4.2 Selection of an appropriate Statecharts semantics

In analogy to our observations in section refselectTS leading to a selection of the weakly-fair interleaving *semantics to interpret the TS specification*, we will *now chose* an appropriate Statecharts semantics for the DCP problem.

The GO REPEAT semantics and the [HG89]-semantics B to E are all inadequate due to the same reason: A failure event placed on the system's interface will always enforce the transition of CCP2 from state PASSIVE-DATA into state ACTIVE-DATA. The interleaving of the processing of an input A and the occurrence of the failure CRASH1 – i. e. a transition from PASSIVE-DATA into ACTIVE – could not be simulated.

In contrast to this, the GO STEP semantics allows placement of a failure event at each critical processing step. This is a consequence of the step definitions allowing new inputs after each micro step. Specifically, CRASH1 can be placed *after* CCP2 has reached PASSIVE-DATA and *before* it returns into initial state PASSIVE-NO_DATA.

The [KP92]-semantics would also be an interesting candidate for the DCP problem, because it allows to model and investigate more complicated situations involving explicit time intervals given for the duration of computations and for the lifespan of events. However, this semantics is at present unavailable in the Statemate tool. Furthermore, we explicitely chose an untimed approach to model the DCP problem. Therefore we will interpret our State- & Activitycharts solution by means of the GO STEP semantics.

4.3 Dual Computer System – State- & Activitycharts solution

Master CCP1 When the Dual Computer System is started, its root activity DCP is automatically started. Since it does not contain a control part, its subactivities CP1 and CP2 are started also. This means that the statecharts CCP1 and CCP2 are started in their initial states. The activities P1 and P2 remain inactive. CCP1 starts in the state UP and substate NO_DATA. When a value is written on the incoming variable (data-item) A, this generates an event wr(A) ("written A") and CCP1 goes to the state DATA, performing two assignments and sending a start signal (st!(P)) to P1. The variable A is in fact a record consisting of a data component A_X and an alternating bit A_B. These two components are stored in local variables X1, B1 as a result of the transition from NO_DATA to DATA.

As a third result of this transition, P1 is started. Its behaviour is not specified formally here, it may be implemented in some other programming language or perhaps in hardware. It will read the input value written on X1 and use this to compute a certain function f(X1), return the result in Y1 and terminate.

As soon as CCP1 sees that P1 writes its result, it will move from DATA to READY and write the result on the output B. From READY it will move to NO_DATA and generate a synchronisation event NEXT1 to the environment activity PRODUCER. This notices the producer process that it can provide a new value on A, which will start the cycle all over again. Notice that in contrast to the TS specification above the NEXT1 event is single-valued. This difference will be explained in section 5.

This will continue until the event CRASH1 occurs. This immediately stops the execution of CCP1 and CCP2 should take over as smoothly as possible.

Slave CCP2 As long as no CRASH1 occurs, CCP2 will stay in the state PASSIVE, following a simple version of the cycle of CCP1, in which the input value on A is stored but not processed. Notice that the transitions from NO_DATA to DATA in CCP1 and CCP2 are taken simultaneously.

When suddenly the event CRASH1 occurs, CCP2 goes from PASSIVE to ACTIVE, ideally to the same state as CCP1 occupied when it crashed. Note that the transitions from DATA to READY are driven by events internal to CCP1. Hence, when event wr(A) has occurred but NEXT1 has not, CCP2 has no means to know, and should not have indeed, whether CCP1 was in DATA or in READY at the moment of the crash. So the protocol decides to stay on the safe side and go to DATA, possibly duplicating the output on B. This is the reason that an alternating bit is added to the data. In this case, the CONSUMER activity will see two consecutive values with the same bit and discard the second.

There is a subtlety here in the case that the events CRASH1 and wr(A) occur at exactly the same moment. In order for CCP2 not to miss one of these events, the "cross" transition from NO_DATA in PASSIVE to DATA in ACTIVE is added.

In analogy the case where CRASH1 and NEXT1 become visible to CCP2 at the same time must be handled: If CCP1 performs the transition from READY to NO_DATA generating event NEXT1, this becomes visible to CCP2 in the next micro step. In the GO STEP semantics described above, also CRASH1 may be placed for this next step, so while CCP1 goes to state DOWN, CCP2 has to cope both with CRASH1 and NEXT1. If CCP2 ignores CRASH1, it will never reach state ACTIVE. If it ignores NEXT1, it will reproduce the last A-input (which does not do any harm), while the PRODUCER might send a new job as a reaction to NEXT1. As a consequence, a wr(A)-event could occur while CCP2 was in state ACTIVE with substate DATA or READY and therefore be lost (which definitely does a lot of harm). Therefore the transition from substate DATA in PASSIVE to substate NO_DATA in ACTIVE is introduced.

When CCP2 has entered ACTIVE, it will stay there, performing the same cycles as CCP1 did before. In the full protocol, CCP1 will have the possibility to be repaired and turning to UP again, now adopting the passive rôle and waiting for CCP2 to receive a CRASH2 event. This would make the two diagrams completely symmetric but also more complicated, so we left it out to concentrate on one representative half of the protocol.

5 Comparison between Transformation Schemas and State- & Activitycharts

In this section we compare Transformation Schemas and State- & Activitycharts and illustrate the major differences by means of the dual computer system example DCP introduced above.

As a basis for this comparison, we concentrate on the weakly-fair interleaving semantics for Transformation Schema and on the GO STEP/GO REPEAT semantics for Statecharts.

5.1 Presentation of specifications

Though the evaluation of State- & Activitycharts' and Transformation Schema's graphical presentation style is not the main objective of this article, it may be appropriate to point out, which differences are the most important ones from our point of view when using the specification methods in "real-world projects".

State- & Activitycharts offers two concepts to present the modularity of a specification: Activity charts can be modularized by drawing new boxes into the top-level activity (as shown in figure 7); in analogy the modularization of statecharts can be shown by drawing higher-level boxes around subordinate states on the same sheet. The second concept allows top-down presentation of activities and statecharts, where the structure of a subactivity or a sub statechart is shown on a separate diagram, while they appear as black boxes on higher-level sheets (as shown for the statecharts `CCP1` and `CCP2`).

For **Transformation Schemas** top-down presentation by means of separate diagrams associated with higher-level black boxes is the only way of showing the structure of a specification.

5.2 Scope and persistency of data objects

In **State- & Activitycharts**, the data item entering an activity on a flow is in the scope of every subordinate activity or statechart, and it can be read arbitrary many times without changing its contents. In principle, data items can be processed both by activities and statecharts. Additional data items can be defined inside an activity or a statechart (like `X1`, `B1`, `Y1` in statechart `CCP1`). These items also have the defining chart and all subordinate charts as scope; but in contrast to incoming data flows, they live as long as the defining activity. The third category of data items are stores which preserve their contents independently of any activity's lifespan.

A consequence of the non-destructive read concept defined for Statecharts data items is that activities processing data must always be controlled by statecharts, because the data items themselves do not provide a trigger that indicates to the activity when to start executing.

Using **Transformation Schemas**, only two types of data items are available: flows and stores. In contrast to State- & Activitycharts, every potential consumer of a data flow must be made explicit by feeding a branch of the corresponding input flow into the data transformation. Unused tokens on flows can be used until the end of the macro step; but the usage is destructive. Therefore the only type of persistent data container for TS is the store, and the data contents of stores remains well defined over the sequence of macro steps.

The destructive read operation on flows allows to define TS that are executed in the *data-triggered* mode: In absence of control transformations the dynamic behaviour of the TS is completely controlled by the data flow, as defined in the micro step rules for data transformations.

Moreover, TS strictly separate information travelling on data flows and information on control flows: It is impossible to process both data and control items inside the same transformation.

If control decisions depend on certain data values, these have to be evaluated by means of data transformations and fed into the corresponding control transformation as data conditions.

5.3 Scope and persistency of control objects

While the scope of control objects (i. e. events) in **State- & Activitycharts** is defined in the same way as for data objects, events in the GO STEP/GO REPEAT semantics only live for the duration of the micro step directly following the step where the events were created. As a consequence, processing of events cannot be "postponed" to later micro steps. This implies the necessity to evaluate event expressions for transition triggers instead of atomic events only.

For **Transformation Schemas**, scope and persistency of control flows is defined exactly as for data flows, so events live until they are consumed by the consumer control transformation or – if they stay unconsumed – until the end of the macro step.

The persistency rules for control objects are motivated by the differences of the underlying semantics: For State- & Activitycharts, consuming an event cannot be destructive, because just as in the case of data flows the event can be consumed by more than one transition. On the other hand, keeping events alive until the end of a macro step would lead to undesirable behaviour of very simple specifications, when interpreted in any semantics allowing several micro steps per macro step. For example, the trivial flip-flop switch shown in figure 10 would lead to a never-terminating macro step.

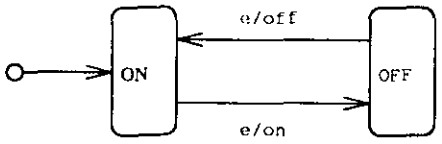

Figure 10: In any semantics allowing several micro steps per macro step this flip-flop switch operation only terminates, if events just live for the micro step directly following their generation.

For Transformation Schemas, events must not be discarded after one micro step, because the interleaving semantics only allows one transition per micro step. As a consequence, the read must be destructive, because otherwise the analogous situation as described in figure 10 would arise for the corresponding control transformation.

5.4 Parallelism

The GO STEP/GO REPEAT semantics as well as the [HG89] semantics of **State- & Activitycharts** are based on the concept of *simultaneous* processing of every input or event that is enabled in a micro step. Therefore in our DCP-example, feeding the failure event CRASH1 into DCP has the effect that both CP1 and CP2 perform their reaction in the same micro step, and this is exactly what we wish to express: Occurrence of CRASH1 has the effect of a high-priority interrupt that leads to immediate reactions on computer CP2.

On the other hand, simultaneity introduces additional complexity in handling the inputs of a statechart, as for example shown in CCP2: For the transition from state PASSIVE, NO_DATA

to state ACTIVE the input signals CRASH1 and wr(A) have to be simultaneously taken into account, because every event (like wr(A)) must be consumed in the micro step following their creation; afterwards they are lost.

In contrast to State- & Activitycharts' behaviour, the interleaving semantics of **Transformation Schemas** does not allow to abstract from the signal "WATCHDOG ALARM" and simply feed CRASH1 into CP2, too, as it is specified by the Statecharts solution: Input of CRASH1 into CP1 and CP2 might lead to a sequence of micro steps, where the failure event is processed by CP2 *before* it is processed by CP1. As a consequence, CP2 could produce the output of a job before the same output is re-produced by CP1. This would not do any harm to the consumer, because he will detect the duplicated result by means of the alternating bit. But this situation could also lead to a duplicated NEXT-message (first NEXT2, then NEXT1), and this has to be regarded as a specification flaw, because the producer might send a new job in reaction to the superfluous NEXT1 message. The "WATCHDOG ALARM" is produced by CCP1 *after* CRASH1 has been consumed. Therefore the reasonable causal relation *"first* CP1 *'s failure, then its detection by* CP2*"* is ensured by our TS.

There is a second case in the TS specification where specific measures had to be taken to exclude unwanted sequences of transitions: It is necessary to ensure that the NEXT-request for a new job sent to the PRODUCER and the "WATCHDOG ALARM" are transmitted on the same flow. Otherwise the weakly-fair interleaving semantics could allow the "WATCHDOG ALARM" to "overtake" the NEXT-request i. e. a NEXT token and a "WATCHDOG ALARM" token could both be placed on CCP2, and CCP2 could first chose the "WATCHDOG ALARM" token. As a consequence, CP2 would disregard the NEXT signal, reproduce the last job already delivered by CP1, and – just as in the case described above – produce a superfluous NEXT2 signal. Transmitting both the "OK" and the "WATCHDOG ALARM" signals on flow NEXT1 ensures that CP2 will receive them in the same order as it has been produced by CP1, i. e. *"first* "OK", *then* "WATCHDOG ALARM"*"*.

These two cases show that for certain cases the degree of interleaving offered by the weakly-fair interleaving semantics is too ample, so that specific specification constructs have to be inserted to reduce the set of possible execution sequences.

5.5 Nondeterminism

An important difference between the DCP-example's Statecharts and Transformation Schema solution is, that the former's execution under the GO STEP regime is completely deterministic, while the latter allows nondeterministic executions: In the Statecharts solution, both automatons CCP1 and CCP2 are deterministic. Since they do not write on the same data items (i. e. racing conditions do not occur), their parallel composition is deterministic, too. Nondeterminism with respect to the failure's occurrence has to be "simulated" by placing the CRASH1-event at random into the series of macro step inputs. In contrast to this, each macro step of the TS solution is nondeterministic, as soon as both the CRASH1 event plus an A-input are placed on the DCP-interface. This is not only caused by the fact that CP1's control transformation CCP1 is nondeterminstic in the TS solution, but mainly by the properties of the interleaving semantics: It cannot be predicted, at which micro step the failure event will lead to the corresponding transitions. At the system's interface this becomes visible by the fact that it cannot be predicted whether the NEXT signal will be delivered via NEXT1 (i. e. CP1 was still able to produce the NEXT-event before crashing) or via NEXT2.

This stronger degree of nondeterminism in Transformation Schemas is in fact not very astonishing. For interleaving semantics, it is well known that parallel specifications can be transformed into nondeterministic sequential specifications (see [AO91, pp. 334]). Therefore in TS, parallelism introduces nondeterminism.

Statecharts allows *event expressions* as triggers of transitions. As a consequence, nondeterministic statecharts can be made deterministic by assigning priorities to events. In figure 8 `CRASH1` is the high-priority event forcing a transition into state `DOWN`, regardless of any other events. This corresponds to event expressions `not CRASH1 and wr(A),...` as triggers for the internal transitions between the `UP` substates. In W&M's definition of Transformation Schemas, only single events are allowed to trigger transitions. Therefore nondeterministic control transformations in general cannot be replaced by deterministic ones. This problem has been addressed by some builders of CASE tools for structured methods by also allowing event expressions as triggers. From our point of view, this is not an appropriate solution, because interleaving semantics do not suggest simultaneous evaluation of events.

5.6 Lifeness properties – fairness

When analyzing the differences between specification languages intended for parallel systems, it is interesting to ask whether certain liveness properties are automatically guaranteed by specific types of specifications. State- & Activitycharts and Transformation Schema seem to be rather similar with respect to the *divergence* of specifications: They both allow specifications with macro step executions that diverge due to statecharts resp. control specifications that perform continuous internal communications without providing any output at the system's interface. In contrast to the divergence liveness property, we can observe important differences State- & Activitycharts and Transformation Schema when looking at *fairness properties*.

Recall that an execution of any transition system is *weakly fair* with respect to a specific transition τ, if it is not the case that τ is continually enabled beyond some position in the execution, but is taken only a finite number of times. The execution is *strongly fair* with respect to τ, if it is not the case that τ is enabled infinitely many times in the execution, but is taken only a finite number of times (see [MP92, pp. 128]).

Fairness Observation 1 *State- & Activitycharts allows specifications that do not have the weak fairness property (and as a consequence also not the strong fairness property).*

For example a system described only by the nondeterministic automaton shown in figure 11 can lead to an unfair execution, if for each macro step the environment provides both `a` and `b` as input.

With the same inputs the analogous Transformation Schema specification would even lead to a strongly fair execution with respect to both `a/c` and `b/d`: Suppose both `a` and `b` are placed on the interface. If the first micro step choses transition `a/c`, then for the second micro step `a/c` is no longer enabled, but `b/d` still is. Since `b/d` is now the only enabled transition, it must be taken according to our semantics.

Because both the causal-chain semantics and the weakly-fair interleaving semantics do not allow a macro step to end, as long as an enabled transition still exists, any permanently enabled transition will be taken before the macro step ends. Since `b/d` is now the only

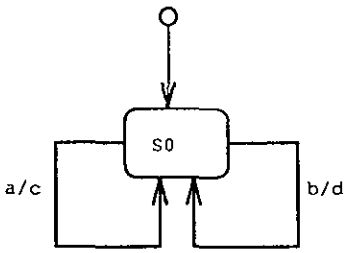

Figure 11: Statechart allowing unfair executions.

enabled transition, it must be taken according to our semantics. Only a non-terminating macro step, where other components communicate to produce "infinite internal chattering" can prevent a permanently enabled transition from being finally taken. This leads to

Fairness Observation 2 *When interpreted in the Causal-Chain or the Weakly-Fair Interleaving Semantics, every transformation schema only allows executions that are at least weakly fair with respect to all possible transitions or possess a non-terminating macro step.*

One of the major differences between the Statecharts semantics and the Transformation Schema semantics can be expressed by analyzing the reasons for absence of strong fairness: For Statecharts, an execution that is unfair with respect to a transition τ can only occur in a specification that contains τ in a *nondeterministic* automaton (like in the example of figure 11), where several transitions are enabled at the same time, but only one of them can be taken. The parallel composition of deterministic statecharts $S_1, \ldots, S_n$ will be strongly fair with respect to all of its transitions, because for each S_i, at most one transition τ_i can be enabled at the beginning of a micro step, so all enabled transitions are performed in parallel during this step. Note however, that the parallel composition of determinstic statecharts is not necessarily deterministic, because the racing conditions mentioned in section 4.1 are resolved nondeterministically.

Fairness Observation 3 *The parallel composition of deterministic statecharts only allows executions, where every transition is taken as often as it is enabled. As a consequence, the executions are strongly fair with respect to all their transitions.*

In contrast to this, when examining our TS semantics, unfair behaviour can be caused both by nondeterministic automatons and by parallel composition of (possibly deterministic) components. As mentioned in the previous subsection, for interleaving semantics, parallelism introduces nondeterminism and – just as in the Statecharts semantics – nondeterminism gives rise to unfair behaviour. This will be illustrated in the following example.

Consider the TS shown in figure 12 and assume a sequence of inputs from the environment that looks like

```
(a,b), a, (a,b), a, ...
```

Both control transformations `C1`, `C2` are deterministic. However, an execution to the inputs above could be as follows: In the first macro step, both transitions `a/disable C2` and `b/d` are enabled in the first micro step. Assume, transition `a/disable C2` is taken. In the second micro step, the `disable C2` event prevents `b/d` from being taken. The macro step ends without having engaged into transition `b/d`. In the second macro step, the input `a` leads to

enabling C2. The third macro step will be as the first and so on. As a consequence, this execution is not strongly fair with respect to transition b/d.

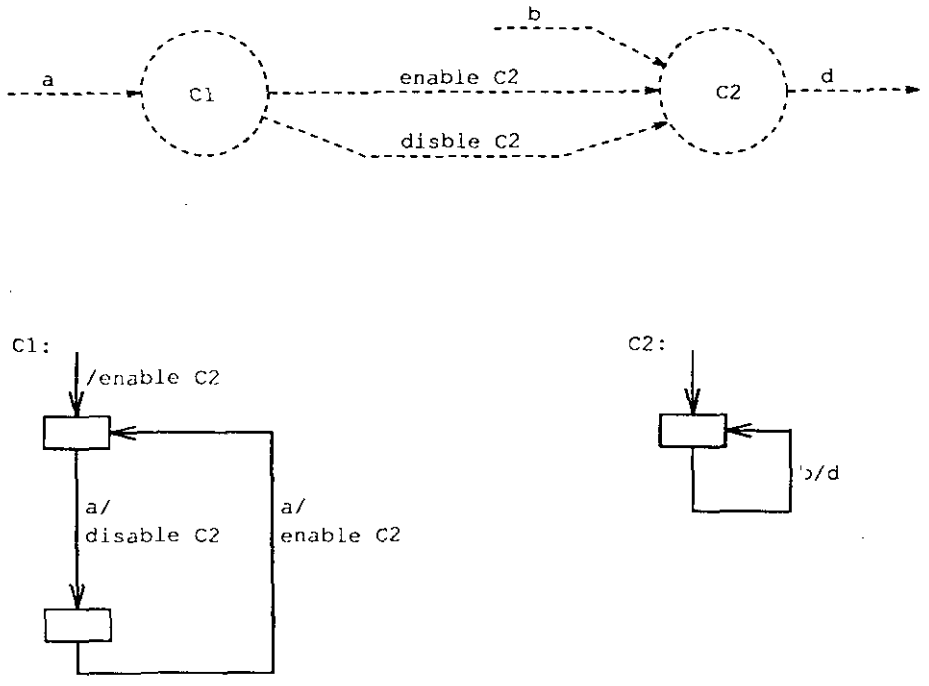

Figure 12: Parallel composition of determinstic control transformations allows executions that are not strongly fair with respect to certain transitions.

6 Conclusion

In this article a comparison between the CASE specification languages Transformation Schema and State- & Activitycharts has been presented, based on an example from the field of fault-tolerance. Having analysed the languages' semantics, we can evaluate this comparison as follows:

1. Both Transformation Schema and State- & Activitycharts do not possess a semantics that is "universally" applicable for most types of target systems. This has been motivated by the selection process neccessary to find a suitable interpretation for the Dual Computer System specification. Instead different semantics had to be defined for both languages, and it is an important task at the beginning of a project's specification phase to select the most appropriate interpretation model for the system to be developed.

2. Transformation Schema only allow one graphical presentation style for the top-down specification of systems. From practical experience we know, that rigorous top-down presentation - while being appropriate for the inspection of completed specifications - is not helpful when developing a new specification. Here a mixed approach using both top-down and bottom-up techniques is better. Therefore we prefer the State- & Activitycharts presentation style, where the degree of top-down structuring can be chosen by the developer himself. We think that the variety of State- & Activitycharts

presentation techniques could also be used for Tranformation Schemas in an analogous way without inducing another understanding of the TS meaning. It just depends on tool builders to sit down and implement it.

3. Transformation Schema and State- & Activitycharts use different communication paradigms. The scope and persistency rules of data items in Transformation Schema suggest communication concepts based on one-to-one or multi-cast channels for the target system. In contrast to this, State- & Activitycharts suggest mechanisms based on shared variables. Note that in most practical cases the underlying communication concepts cannot simply be selected according to the developer's personal taste. Instead they are often predefined by boundary conditions regarding the target environment and should be taken into account during the selection process for the appropriate CASE method.

4. While TS strictly separates the manipulation of data and control items, State- & Acivitycharts allow to specify operations on data directly with the state transition, without introducing a corresponding activity. From our experience, the strict separation of data processing and control enforced by TS rules leads to clearer specifications, therefore we do not regard this as a disadvantage. For Statecharts, the more flexible data manipulation concept requires quite an amount of discipline from the developer.

5. For specifications that do only require a moderate amount of control, TS specifications require less effort to write than the corresponding State- & Activitycharts, because TS allows to specify without control transformations. The data-driven dynamic behaviour will then be defined by the micro step rules for data transformations plus the macro step rule.

6. The TS semantics introduced and the semantics of State- & Activitycharts incorporate different notions of parallelism, nondeterminism and fairness, that are suitable for different types of target systems. As a consequence, syntactically similar transformation schemas and statecharts differ strongly with respect to their dynamic behaviour. Because of its restrictive use of micro step interleaving, the causal-chain semantics for TS maps well on multi-tasking/single CPU systems. The full-interleaving semantics is appropriate for distributed systems with a low degree of synchronisation, preferrably implemented by means of message passing mechanisms. The State- & Activitycharts semantics are especially well suited for multi processor systems with tight memory coupling and rather strict mechanisms for the synchronisation of the processors' input and output.

7. In the Dual Computer System example the Statecharts solution appears to be superior to the Transformation Schema solution, because the simultaneous processing of parallel components is just appropriate for the specification of reactions to a failure event. In this example, the TS solutions appears to possess a lower level of abstraction, because an additional event (`"WATCHDOG ALARM"`) had to be introduced to guarantee proper causal relationships. That is, the developer had to explicitely introduce "technical" synchronization and scheduling constructs, because otherwise the weakly-fair interleaving semantics would allow "unwanted" sequences of micro steps. However, in loosely coupled or even wide-area networks the concept of simultaneous parallel processing steps, as inherent to the State- & Activitycharts semantics presented, may suggest

misleading simplifications, so that such systems could be better represented by the interleaving semantics of Transformation Schema.

8. Though we have not studied real-time aspects in this article, it is interesting to note that TS concepts for incorporating real-time as sketched in [Wa86] are not suitable for complex applications. In contrast to this, the built-in real-time simulation features as implemented in Statemate are at least a step in the right direction, and more universal and theoretically sound techniques for real-time specifications with State- & Activity-charts have been worked out in the formal methods community ([KP92]). The development of TS extensions that incorporate real-time aspects will be a main objective of our future activities in this field.

There are a number of conceptual disadvantages or flaws, that are inherent both in Transformation Schema and State- & Activitycharts, but have not been discussed in this article, because we focused on problems related to parallel systems. Some of these draw backs can be solved in a similar way for TS and State- & Activitycharts; this is currently under investigation at DST in cooperation with Eindhoven University of Technology and Christian–Albrechts–Universität zu Kiel.

- Both languages offer insufficient means for the precise definition of complex data structures and functions operating on these structures. This could be easily improved by adopting the concepts of formal specification languages like Z ([SP92]) or VDM ([Jo86]) for the definition of data items and operations specified in data transformations resp. activities. This approach has been investigated at DST ([Pel92]).
- Both languages do not provide constructs for *data refinement*. If a "concrete" specification is intended to be a refinement of an "abstract" one, the relation between concrete and abstract data structures cannot be expressed. Again, this can be overcome by importing the Z or VDM concepts for data refinement.
- Both languages do only provide insufficient support for re-use of specification parts.
- Both languages do not support object-oriented specification styles.

It is often said, that a specification language merely serves as a vehicle for the developer to express her or his concepts for the system to be built. From our experience, the impact of using a specific language is much deeper, because both syntax and semantics not only influence the developer's specification style, but also his way of thinking about the system. As a consequence, the use of different languages will lead to different system solutions. Therefore the choice between CASE methods – as between Transformation Schema and State- & Activitycharts - should always be based upon a close analysis of the methods' underlying semantics and their appropriateness for the target system.

Acknowledgement We would like to thank Willem-Paul de Roever for stimulating our work on this article and for many helpful discussions.

References

[AO91] K.R.Apt, E.-R.Olderog. *Verifiction of Sequential and Concurrent Programs*. Springer, New York (1991).

[BB92] A. Benveniste and G. Berry. *The Synchronous Approach to Reactive and Real-Time Systems*, in IEEE-Proceedings *"Another Look at Real-Time Programming"*, 1992.

[CAU93] C. Petersohn, C. Huizing, J. Peleska, W-P. de Roever. *Formal Semantics for Ward & Mellor's TRANSFORMATION SCHEMA, and their Comparison with STATECHARTS*. Christian-Albrechts-University at Kiel, Technical Report (1993).

[Ha88] D. Harel. *On visual formalisms*. Communications of the ACM, 31:514–530, 1988.

[Ha90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. *Statemate: A working environment for the development of complex reactive systems*. IEEE Transactions on Software Engineering, 16(4):403–414, April 1990.

[HG89] C. Huizing and R. Gerth. *On the semantics of reactive systems*. Technical report. Eindhoven University of Technology, 1991.

[HGdR88] C. Huizing, R. Gerth, and W.-P. de Roever. *Modelling statecharts behaviour in a fully abstract way*. In *Proc. 13th CAAP*, LNCS 299, pages 271–294, 1988.

[Ho85] C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International, Englewood Cliffs (1985).

[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. *On the formal semantics of Statecharts*. In *Proceedings Symposium on Logic in Computer Science*, pages 54–64, 1987.

[HG92] C. Huizing and R.T. Gerth. *Semantics of Reactive Systems in Abstract Time*, in *"Real-Time: Theory in Practice"*, proceedings of a REX workshop, June 1991, Mook, edited by J.W. de Bakker, W.-P. de Roever, G. Rozenberg, LNCS 600, Springer Verlag, Berlin, Heidelberg, 1992.

[KP92] Y. Kesten, A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil (ed.) *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer (1992), pp. 591-619.

[iL89] i-Logix Inc. *Statemte Analyzer*. version 3.0, i-Logix Inc., Burlington (1990).

[Jo86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International series in computer science (1986).

[MP92] Z. Manna, A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer (1992).

[Pel91] J. Peleska. Design and Verification of Fault Tolerant Systems With CSP. *Distributed Computing* 5 (1991), 95-106.

[Pel92] J. Peleska. *Formal Software Engineering, Z and Structured Methods – provable correctness for safety-critical systems*. DST Deutsche System-Technik, Kiel, Technical Report (1992).

[Pel93] J. Peleska. CSP, Formal Software Engineering and the Development of Fault-Tolerant Systems. In Vytopil, J. (ed.): *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Kluwer Academic Publishers (1993).

[Pl83] G. Plotkin. *An operational semantics for CSP*. In *Proceedings of the IFIP Conference on the Formal Description of Programming Concepts II*, North Holland (1983) pp. 199-225.

[Pt92] C. Petersohn. *Modellierung reaktiver Systeme mit Transformationsschema und ein Vergleich mit Activity- und Statecharts*, Master's thesis, report, Christian-Albrechts-Universität zu Kiel, 1992.

[SP92] M.J. Spivey. *The Z Notation*. Prentice Hall (1992).

[Wa86] P.T. Ward. *The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing*. IEEE TSE, Vol. SE-12, No. 2, pp. 198-210, Febr. 1986.

[WM85] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems*. (3 vols), Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs, 1985.

[Wo89] D.P. Wood and W.G. Wood. *Comparative Evaluations of Specification Methods for Real-Time Systems*, draft, September 1989.

Computing Science Notes

Department of Mathematics and Computing Science
Eindhoven University of Technology

In this series appeared:

91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.
91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.

91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.

91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.
92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.

92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.
92/26	T.H.W.Beelen W.J.J.Stut P.A.C.Verkoulen	A modelling method using MOVIE and SimCon/ExSpect, p. 15.
92/27	B. Watson G. Zwaan	A taxonomy of keyword pattern matching algorithms, p. 50.
93/01	R. van Geldrop	Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.
93/02	T. Verhoeff	A continuous version of the Prisoner's Dilemma, p. 17
93/03	T. Verhoeff	Quicksort for linked lists, p. 8.
93/04	E.H.L. Aarts J.H.M. Korst P.J. Zwietering	Deterministic and randomized local search, p. 78.
93/05	J.C.M. Baeten C. Verhoef	A congruence theorem for structured operational semantics with predicates, p. 18.
93/06	J.P. Veltkamp	On the unavoidability of metastable behaviour, p. 29
93/07	P.D. Moerland	Exercises in Multiprogramming, p. 97
93/08	J. Verhoosel	A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.
93/09	K.M. van Hee	Systems Engineering: a Formal Approach Part I: System Concepts, p. 72.
93/10	K.M. van Hee	Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.
93/11	K.M. van Hee	Systems Engineering: a Formal Approach Part III: Modeling Methods, p. 101.
93/12	K.M. van Hee	Systems Engineering: a Formal Approach Part IV: Analysis Methods, p. 63.
93/13	K.M. van Hee	Systems Engineering: a Formal Approach Part V: Specification Language, p. 89.

93/14	J.C.M. Baeten J.A. Bergstra	On Sequential Composition, Action Prefixes and Process Prefix, p. 21.
93/15	J.C.M. Baeten J.A. Bergstra R.N. Bol	A Real-Time Process Logic, p. 31.
93/16	H. Schepers J. Hooman	A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27
93/17	D. Alstein P. van der Stok	Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.
93/18	C. Verhoef	A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.
93/19	G-J. Houben	The Design of an Online Help Facility for ExSpect, p.21.
93/20	F.S. de Boer	A Process Algebra of Concurrent Constraint Programming, p. 15.
93/21	M. Codish D. Dams G. Filé M. Bruynooghe	Freeness Analysis for Logic Programs - And Correctness?, p. 24.
93/22	E. Poll	A Typechecker for Bijective Pure Type Systems, p. 28.
93/23	E. de Kogel	Relational Algebra and Equational Proofs, p. 23.
93/24	E. Poll and Paula Severi	Pure Type Systems with Definitions, p. 38.
93/25	H. Schepers and R. Gerth	A Compositional Proof Theory for Fault Tolerant Real-Time Distributed Systems, p. 31.
93/26	W.M.P. van der Aalst	Multi-dimensional Petri nets, p. 25.
93/27	T. Kloks and D. Kratsch	Finding all minimal separators of a graph, p. 11.
93/28	F. Kamareddine and R. Nederpelt	A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.
93/29	R. Post and P. De Bra	GOLD, a Graph Oriented Language for Databases, p. 42.
93/30	J. Deogun T. Kloks D. Kratsch H. Müller	On Vertex Ranking for Permutation and Other Graphs, p. 11.
93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.

93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Pathwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Kornatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.

94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.
94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.