INSTITUT FÜR INFORMATIK
UND PRAKTISCHE MATHEMATIK

Formal Semantics for Ward & Mellor's TRANSFORMATION SCHEMA's and its Application to Fault-Tolerant Systems

Carsta Petersohn, Jan Peleska, Cornelis Huizing and Willem-Paul de Roever

Bericht Nr. 9420
Dezember 1994

CHRISTIAN-ALBRECHTS-UNIVERSITÄT
KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Formal Semantics for Ward & Mellor's TRANSFORMATION SCHEMA's and its Application to Fault-Tolerant Systems

Carsta Petersohn, Jan Peleska, Cornelis Huizing and
Willem-Paul de Roever

Bericht Nr. 9420
Dezember 1994

e-mail: cp-, jap-, wpr@informatik.uni-kiel.d400.de,
keesh@info.win.tue.nl

submitted to the CRL journal Computer Systems, Science & Engineering

Abstract

A family of formal semantics is given for the Essential Model of the Transformation Schema of Ward & Mellor [12] using recent techniques developed for defining the semantics of Statecharts [3] by Pnueli and Huizing. A number of ambiguities and inconsistencies in Ward & Mellor's original definition is resolved. The models developed closely resemble those used for synchronous languages [1]. Each model has its own application area, e.g., one fits best for fault-tolerant systems.

Keywords:
Real-time embedded systems and their design, Structured Analysis and Design Methods, Transformation Schema, formal semantics, synchronous languages, micro and macro steps, Berry's synchrony hypothesis, safety critical systems design, simulation of and semantics for fault-tolerant systems.

1 Introduction

1.1 Motivation and Goal

Structured Analysis and Design methods (SADM) aim at giving a specification of software which is independent of, and considerably more abstract and readable than, the code eventually produced. Their goal is to provide in this way a specification which:

a) exposes inconsistencies in the requirement document describing what a client 'thinks' she/he wants, as opposed to the finally debugged hopefully consistent requirement specification describing what she/he 'actually' wants, and

b) provides a consistent requirement specification and independent description of the task of the software to be written by the implementor.

Obviously, this process looses a lot of its potential value once the SAD methods used contain in their definition bugs and inconsistencies themselves. This happens, e.g., in case of an executable specification language, when the execution of a specification does not faithfully represent the semantics of that specification as laid down in the document defining the method. This would endanger point a) above. As to point b), such inconsistencies might result in a specification of dubious value, since an implementor would not know exactly what to implement, when the meaning of the requirement specification is ambiguous or even inconsistent.

One of the well known SAD methods is that of Ward & Mellor. Although widely used, its description [12, 11] contains a number of such inconsistencies. Yet W&M's method contains at least sufficient indications for us to try to *reconstruct* its intended meaning. We show that with the formal methods developed for the definition and analysis of so-called *synchronous* languages (see [1] for an overview) a consistent and precise semantics can be reconstructed for the W&M method. Incompleteness in description and downright contradictions in claimed 'definitions' can be identified and removed, and the rather remote link with timing can be built upon to form a foundation for what is promised by the method which is at least consistent. This is one important purpose of the present paper, in which we give an example of the main flaws in W&M's definition of the semantics of transition diagrams, our suggestions to resolve them (Sec.2) and sketch a precise semantics for the Essential Model of W&M's method (Sec.3). Also a formal definition of W&M's semantics enables the development of a symbolic interpreter to animate TS, which is of great importance for point a) above. The other important purpose of this paper is to argue the need for a family of semantics for different application areas using a 'real-world' example from the field of fault tolerance (See.4).

1.2 Main Technique

The method of W&M uses Transformation Schemas (TS) to represent a system. These are based on data flow diagrams, but can also represent the control aspect of a system. Therefore a TS consists of data and control components, which are both divided into transformations (centers of activities), stores and flows.

Some of the basic flaws in the description of the semantics of Transformation Schemas in the Essential Model of W&M's method given in [11] are the following (also see Sec.2):

1. The method lacks a consistent description of *when* a transformation can start computing upon its input. E.g., one interpretation of Ward's definition may lead to an unnecessary loss of data.

2. The description of the time dependent behaviour of TS is ambiguous. For the life-span of a data-item depends on the interpretation of a 'discrete point in time', but a clear definition is missing of what a 'discrete point in time' is.

We resolve these flaws technically in section 3 by defining a formal operational semantics for that part of TS whose interpretation causes the above mentioned flaws. To be more precise we define a family of formal semantics. Its members are called *recursive causal chain*, *weakly fair* and *full interleaving semantics*. All of these semantics are interleaving semantics defined by transition systems. Referring to [7] for a non graphical syntax of TS and their specifications, these semantics consist of *macro steps*, describing the observable behaviour as seen by the outside world, which in their turn are made up out of (sequences of) *micro steps*, describing the internal processing steps of a TS which describe the internal execution of its transformations. Depending on the family of the particular member of semantics it is belonging to, an internal sequence of micro steps can be characterized by properties such as *maximal* or *recursive-causal-chain* (defined in Sec. 3.2).

The Essential Model is characterized by an abstract notion of time. Every transformation needs zero time to react on input and to produce an output ([12] p.94). The abstract notion of time involved here is such that micro steps take no time for their execution. However a macro step takes a positive amount of external time (as can be interpreted from Table III [11] p.206). This division between micro and macro steps is characteristic for the semantics of synchronous languages, in which the following idealization is adopted: synchronous systems produce their output synchronously with their input (Berry's Synchrony Hypothesis [2]). Of course this hypothesis does not hold for our usual notion of time. It merely expresses that the time taken by a finite number of internal steps of the system should be negligible in comparison with the time between successive external stimuli.

The formal technique dealing with these two notions of step (due to Pnueli [9], Huizing [4], and others) had not been sufficiently formalized around 1985 for W&M to be able to realize its consequences for a worked out semantics. Our contribution is that we adapt these techniques to define a family of semantics for TS, especially for W&M's model. Though a formalization and animation of TS is possible by translating a TS into a Petri net [10], the notions of interleaving semantics, micro and macro steps introduced in the present paper enable a discussion of the *different* views of the dynamic behaviour of a TS, for instance regarding timing, which is not feasible using the tools of [10].

1.3 Application Area: Fault Tolerant Systems

We define a *family* of semantics, because, as we shall argue, every application area imposes its own criteria for being satisfactorily modelled. In particular, Ward's semantics, represented by our recursive causal chain semantics, is appropriate for modelling multitasking and single processor systems, but it turns out that Ward's semantics is not suitable for modelling fault tolerant processing. In this case we find that fault tolerant systems require our weakly fair interleaving semantics (see Sec.4).

1.4 Future Work

Building a symbolic interpreter for W&M's method, based on the formal semantics sketched in this paper, is part of a project in cooperation with a local industry (DST). As next stages, we intend to integrate another real-time model as also described in [11], to give a similar formalization of W&M's implementation model, described in vol. 3 of [12], and investigate its link (in terms of possible notions of refinement) with our formalization of the essential model in the present paper. Apart from the flaws mentioned in chapter 1.2 one might object that the method lacks any yardstick for determining correctness of data refinement, or even flow refinement. Therefore integrating possibilities of formally founded refinement is a next important stage.

2 W&M's Method and an Examples of an Unsolved Ambiguity

In this section we discuss briefly some of the ambiguities in the interpretation of Transformation Schemas as defined in [12] and [11]. (See [6], [7] for details.)

2.1 A Short Introduction to Transformation Schemas

Transformation Schemas (TS) consist of data and control components. We give here just a short introduction to their main constituent parts, called 'transformation','flow' and 'store', which may be labelled by identifiers. We restrict ourselves in these pages to that part of W&M which is formally characterized in the present paper. For example, we assume that all flows are time–discrete, i.e., that they are not continuous.

Example 1 ((Transformation Schema)) *In figure 1 P denotes a data transformation and K a control transformation. The data flow a is an input flow and the data flow b is an output flow of transformation P. Flows which start from 'nowhere' (as flow a) and flows which end 'nowhere' (as flow b) are connected to the outside world of the Transformation Schema. The flows c, d, e and $Prompt(P)$ are control flows. Data flows carry values and control flows carry events. An event is a special value which just indicates that something has happened. The control flow $Prompt(P)$ (which is a special notation of ours) carries the events 'ENABLE(P)' and 'DISABLE(P)'. Such control flows are called* Prompts. *Their meaning is explained below.*

If there is no value on the output flows b, e and the transformation P is not stopped by the control transformation K, then the transformation P computes an output along b or e as soon as an input arrives along a. Such behaviour is called data–triggered. *The flow e is called* data condition *and represents the possibility that control signals can be fed back from a data transformation to a control transformation. The control transformation K stops the data transformation P by sending a value 'DISABLE(P)' to P along the flow $Prompt(P)$. If P is stopped, it can not compute outputs and throws arriving inputs away. The control transformation K starts P up again by sending a value 'ENABLE(P)'. When a transformation has no Prompt as an input flow, the transformation is never stopped.*

□

For every transformation of a Transformation Schema there must exist a *specification*. In our formal semantics which we sketch in chapter 3, data transformations are specified by a relation which also takes values of data stores into account. Data stores are equivalent of a memory within our formalism.

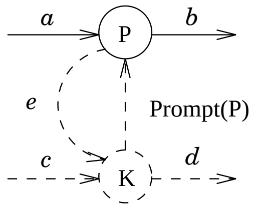

Figure 1: Transformation Schema

2.2 Behaviour of a Transformation

According to [12] p.97 it is impossible for a transformation to output a new value along an output flow as long as some old output value (due to a previous computation) has not been 'cleared' from that flow. As a consequence, W&M's model implies that flows have a buffering capacity of 1. On the other hand [11] p.200 states that as soon as an input arrives it will be processed. A model which meets both requirements may lead to a loss of output data of the transformation. Therefore we list below all possible alternatives we can think of for defining the behaviour of a transformation and discuss which one is best.

1. a) The input is thrown away, if there is still an old value on an output flow. (This option seems to be implied in the implementational model of [11] p.208).
   b) The output is calculated, but its placement on output flows is restricted to flows which are not occupied by old values.
   c) Old outputs are overwritten by new ones.
2. An arrived input value of a transformation is processed only after consideration of the output flows.
   a) The calculation is only started, when the resulting output values are going to appear on flows which are free before the calculation.
   b) The transformation waits with the computation until all output flows are free.

All options under 1) lead to an arbitrary loss of data and are therefore useless for modelling data processing systems. An example is given in [6]. Option 2a) requires foreknowledge and is therefore rejected. This leaves us option 2b) since we do not want arbitrary loss of data.

3 Sketch of a Family of Formal Semantics

In this section we sketch a family of formal operational semantics of TS referring to a non-graphical syntax of TS. (In [6], [7] a more complete definition of the semantics is given). One member of this family closely reflects Ward's original ideas as described in [11]. All members of this family of semantics for TS consist of *macro steps* describing the observable behaviour of a TS as seen by the outside world. A macro step is made up of a sequence of internal processing steps called *micro steps*. Each member of our family of formal semantics is characterized by restrictions on the sequence of internal micro steps and restrictions on the macro steps. The internal sequence represents the reaction of a TS on information sent along its flows by the outside world, and the macro step represents the abstract view of this sequence as presented to the outside world.

3.1 Micro Step

A micro step represents an internal processing step of a data or control transformation belonging to a Transformation Schema T. It is defined formally as a labelled transition

$$(\mathrm{T}, fl, \sigma) \rightarrow_{in}^{out} (\mathrm{T}, fl', \sigma')$$

in the style of Plotkin [8]. Here the flow *in* carries the value that causes the internal processing step that is represented by the micro step and the quantity *out* consists of flows getting new values as a result of the processing step. The tuple (T, fl, σ) is called a micro configuration and is defined as follows:

- T stands for a syntactic representation of a TS. Note that T is not changed in the transition.
- fl denotes a state of the flows of T. It is a function mapping the names of flows to the values they are carrying, where the symbol '$\perp$' represents a formal value indicating that the flow does not carry a processable value.
- σ denotes the state of the transformations of T, e.g., it maps any name from a data transformation of T to the set {DISABLE, ENABLE}, where DISABLE expresses that the transformation has stopped and ENABLE expresses that the transformation may process depending on values on flows. Also σ denotes the state of the stores of T, which maps the names of the stores to the values of the variables which they carry.

A micro configuration (T, fl, σ) induces a micro configuration $(\mathrm{T}^*, fl^*, \sigma^*)$ for every transformation diagram T^* contained in T, where fl^* and σ^* denote corresponding restrictions of fl and σ to, respectively, the flows and transformations of T^*.

A transformation schema T is made up out of data and control transformations. So in order to capture the meaning of a micro step of TS formally, one first defines these steps on the level of its constituting data and control transformations and then introduces the individual micro step on the level of the overall TS, i.e., micro step itself is defined inductively over the non-graphical syntactic structure of a TS. Therefore we need two axioms, one for a data transformation step and one for a control transformation step, and a micro rule to describe the processing step of the overall TS which contains these transformations using these axioms. Below we sketch the axiom for data transformations and the micro rule.

3.1.1 Axiom for Data Transformations

A data transformation is represented syntactically by $\mathrm{Dtra}(A, I, O, Sp)$, where A is the identifier of the transformation, I, O denote the sets of its input and output flows and Sp denotes the set of all stores, which can be written or read by the data transformation.

With every data transformation A a relation f_A and a state are associated. The relation f_A specifies the relation between input and output data. The state of a data transformation A is a tuple (dt, ds), where $dt(A)$ can be either ENABLE or DISABLE and $ds(A)$ maps every store of Sp to its value.

Definition 3.1 ((Axiom for data transformations)) *Assume fl, fl' are states of the set of flows $I \cup O$, $\sigma = (dt, ds)$ and $\sigma' = (dt', ds')$ are states of the data transformation and a flow $in \in I$ and a set of flows $out \subseteq O$, so that one of the following two conditions holds:*

1. *The input flow 'in' is a data flow and the following holds:*
   (a) *The precondition for the processing of the transformation is met:*
   $$fl(f) \begin{cases} \neq \bot & , \text{ if } f = in, \\ = \bot & , \text{ if } f \in O. \end{cases}$$
   (b) *The result of the processing of the transformation is:*
      i. *$dt' = dt$ and*
      ii. *if $dt(A) = ENABLE$ then*
      $((fl, ds), (fl', ds')) \in f_A$ *and* $out = \{o \,|\, fl(o) = \bot \; \wedge \; fl'(o) \neq \bot\}$,
      otherwise, if $dt(A) = DISABLE$ then
      $(fl', ds') = (fl[\bot/in], ds)$ *and* $out = \emptyset$.
2. *The input flow 'in' is a Prompt, and the following holds:*
   (a) *The precondition is met:*
   $$fl(in) \in \{ \text{ENABLE, DISABLE} \}.$$
   (b) *The result is:*
   $$dt'(A) = fl(in), (fl', ds') = (fl[\bot/in], ds) \text{ and } out = \emptyset.$$

The data transformation step is now defined as follows :

$$\boxed{(DTra(A, I, O, Sp), fl, \sigma) \rightarrow_{in}^{out} (DTra(A, I, O, Sp), fl', \sigma')}$$

□

The step is called *enabled* if its precondition as mentioned under (1.a) or (2.a), holds.

Condition 1 models what happens when transformation A performs a processing step, i.e., the process is data-triggered (see exp.1). This step is only started if all output flows are free prior to processing (see sec.2.2). The result depends on whether the state of the transformation is ENABLED or DISABLED. Condition 2 models what happens when the transformation is enabled or disabled, i.e., its possible change of state.

In the following we describe how the whole TS behaves if a transformation performs a processing step.

3.1.2 Parallel Composition

A Transformation Schema is a network of $n \in \mathbb{N}$ components T_k, $k \in \{1, ..., n\}$, each one of which has I_k as its set of input flows, and O_k as its set of output flows. The TS is represented non graphically by $\mathrm{T} = (\mathrm{T}_1 \mid\mid \ldots \mid\mid \mathrm{T}_n)$. If a flow f is element of O_k and I_l, where $k, l \in \{1, ..., n\}$ then flow f 'connects' T_k with T_l. A micro configuration $((\mathrm{T}_1 \mid\mid \ldots \mid\mid \mathrm{T}_n), fl, \sigma)$ induces by convention micro configurations $(\mathrm{T}_i, fl_i, \sigma_i)$ for the components T_i of $\mathrm{T}_1 \mid\mid \ldots \mid\mid \mathrm{T}_n$, for $i \in \{1, \ldots, n\}$.

If a transformation does a processing step, so does the whole TS. Formally the micro rule determines how to get a labeled transition with two micro configurations for the whole TS from a labeled transition with two micro configurations for a transformation. We adopt an interleaving semantics, i.e., only one transformation performs a processing step in one micro step.

3.2 Internal Sequence of Micro Steps

Internal sequences of micro steps represent the way the input from the outside world is processed by a TS. Members of our family of semantics can be characterized by properties of the internal sequence of micro steps which we define as *maximal* or *recursive-causal-chain*. These properties are closely related to statements made in [11].

3.2.1 Maximal

One statement describing the internal processing of a TS in [11] is as follows: '*the consequences of the arrival of a value on a flow from outside the schema are worked out before any other value from outside the schema is accepted, and the execution of simultaneously arriving values on flows from outside the schema is sequential but in indeterminate order.*'

In terms of our formal semantics, this statement is represented by the restriction that every internal sequence of micro steps must be maximal.

Definition 3.2 (Maximal) *Given a particular set of values (produced by the outside world) on the input flows of a TS, the resulting internal sequence of micro steps is called* maximal *when:*

1. *the internal sequence is infinite, or*
2. *the internal sequence is finite, and no micro step due to that set of input values is possible at the end of the internal sequence; i.e., no data or control transformation step is anymore enabled.*

$\square$

If a maximal sequence is finite and consists of $n-1 \in \mathbb{N}$ micro steps, we write $(\mathrm{T}, fl_1, \sigma_1) \rightarrow_{in_1}^{out_1} \ldots \rightarrow_{in_{n-1}}^{out_{n-1}} (\mathrm{T}, fl_n, \sigma_n) \not\rightarrow$. If a maximal sequence is infinite, we write $(\mathrm{T}, fl_1, \sigma_1) \uparrow_{in_1}$.

3.2.2 Recursive Causal Chain

Another statement concerning the further internal processing of input in [Wa86] is: '*in case of simultaneous placement of a number of tokens, the execution rules specify carrying out the interactions sequentially but in an arbitrary order.*' ...'*each branch of the interaction is carried out till its conclusion before returning to the next one. If subbranches are encountered during an interaction, another arbitrary sequencing decision is made and the procedure is applied recursively.*'

In terms of our formal semantics this statement is modelled by the restriction that the internal sequence of micro steps must be a sequence of specially ordered causal chains. In a *causal chain* every micro step except the first one depends causally upon the previous step in the sequence, i.e., the input of a micro step is an output of the previous micro step. The order in which these causal chains are composed is that obtained by backtracking the following tree *recursively*: its edges are the flows along which data values or events occurred during the computation, and its nodes the transformation executed. The formal definition of when a sequence of micro steps forms a *recursive causal chain* is given in [6].

3.3 Macro Step

A macro step represents a reaction on an input sent by the outside world of a Transformation Schema T(I, O), where I is the set of flows of the TS coming from the outside world and O the set of flows of the TS directed towards the outside world. Correspondingly, a macro configuration (T,fl, σ) is defined similarly as a micro configuration, except that fl is a mapping of just $I \cup O$ (and not of all the flows of T) to the values carried on these flows. We define two kinds of macro steps for a semantics. The first one is defined as a labelled transition between macro configurations, which is derived from a finite internal sequence of micro steps. The second kind of macro step is derived from an infinite internal sequence of micro steps. Therefore an 'end' macro configuration does not exist. For the recursive causal chain semantics the first kinds of macro step are defined formally below. Depending on the different properties which the internal sequences of micro steps should satisfy, different macro rules and a *family of semantics* for TS are defined.

3.3.1 Recursive Causal Chain Semantics

The *recursive causal chain semantics* semantics most closely reflects Ward's original ideas described in [11], which are mentioned in section 3.2.2. Each internal sequence of micro steps must be maximal and must be a recursive causal chain. After each internal sequence of micro steps all values left on flows which could not be consumed are cleared before a new internal sequence of micro steps starts. Formally this is represented by:

Definition 3.3 ((Macro rule)) *Let $\mathcal{F}$ be the set of flows of T. Assume*

1. *(T,fl_1, σ_1) $\rightarrow^{in_1}_{out_1} \ldots \rightarrow^{in_{n-1}}_{out_{n-1}}$ (T,fl_n, σ_n) with $n \in \mathbb{N}$ is an internal sequence of micro-steps of T, where*

   (a) the chain is maximal and a recursive causal chain,

(b) fl_1 satisfies $\forall_{z \in \mathcal{F} \setminus (I \cup O)} : fl_1(z) = \bot$,

2. *$in, out \subseteq (I \cup O)$, where $in = \{ x \in (I \cup O) \mid fl_{in}(x) \neq \bot \}$ and $out = O \cap \bigcup_{i=1}^{n} \{out_i\}$,*

3. *fl_{in} is a state of $(I \cup O)$ which meets $\forall_{x \in (I \cup O)} : fl_{in}(x) = fl_1(x)$,*

4. *fl_{out} is a state of $(I \cup O)$, which meets $\forall_{x \in (I \cup O)} : fl_{out}(x) = fl_n(x)$.*

Given the assumptions above, the first kind of macro rule is defined as follows:

$$\boxed{\frac{(\mathrm{T}, fl_1, \sigma_1) \rightarrow_{in_1}^{out_1} \ldots \rightarrow_{in_{n-1}}^{out_{n-1}} (\mathrm{T}, fl_n, \sigma_n) \not\rightarrow}{(\mathrm{T}, fl_{in}, \sigma_1) \Rightarrow_{in}^{out} (\mathrm{T}, fl_{out}, \sigma_n)}}$$

□

3.3.2 Weakly Fair Interleaving Semantics

This semantics does not reflect Ward's statement mentioned in section 3.2.2, but the statement mentioned in 3.2.1. It has the same initialization and termination assumptions as the recursive causal chain semantics (conditions 1.b, 2, 3, 4 of the macro rule), but drops the recursive-causal-chain condition 1.a of the macro rule by allowing *any* possible transition to be taken for each micro step. The condition 1.a is replaced by 'maximal holds'. Consequently, no transformation able to make a step is left at the end of the premise of the macro rule. The name given to this semantics is motivated by this fact. For full discussion of this topic see [5].

3.3.3 Full Interleaving Semantics

This semantics drops the input restrictions (condition 1 of the macro rule) by allowing new inputs from the environment to be placed and processed at each micro step. As a result no observable difference between macro and micro steps remains, and therefore macro steps are identified with micro steps. Note that there is no situation where values placed on flows are cleared because they are left after an internal sequence of micro steps, i.e., condition 1.b of the macro rule is dropped.

4 Application Area: Fault Tolerant Systems

Our opinion is that every member of our family of semantics has its own application area in the 'real-world'. An example from the field of fault tolerant systems is sketched below to investigate the practical applicability of the various semantics defined above. The example is part of a typical problem of hardware redundancy having two mutually duplicating computers $CP1$ and $CP2$ to prevent system failure. The complete specification is given in [5].

The TS given in figure 2 represents the internal structure of $CP1$. Transformation $P1$ processes input from flow $a1$ and produces output on flow $B1$. If flow $B1$ gets a value at the same moment flow $b1$ and $wrB1$ get a value, respectively an event. If process $CCP1$ gets an event $wrB1$ and has not consumed an event $CRASH1$ before, it produces an event $NEXT1$. If the process gets an event $CRASH1$ it disables process P1.

The following *fault hypothesis* must be modelled: Computer $CP1$ can be stopped by a failure event $CRASH1$, though $CP1$ is processing an input. The input from flow $a1$ is consumed and an output on flow $b1$ produced, but an event NEXT1 will not be produced.

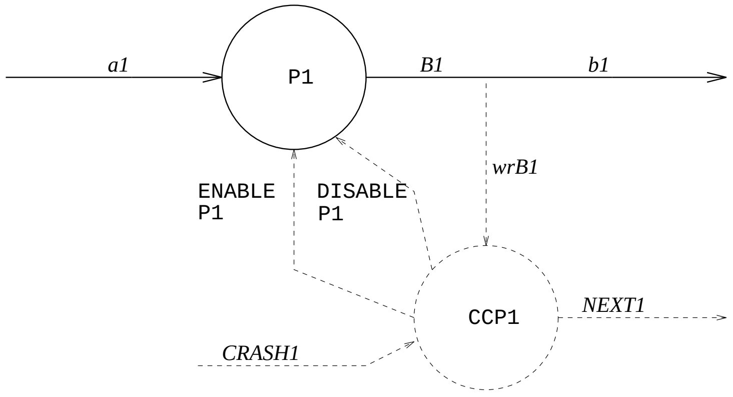

Figure 2: Internal structure of $CP1$

With the *recursive causal chain semantics* it is not possible to model the fault hypothesis. To model the fault hypothesis, $CRASH1$ and a value on flow $a1$ must be input of one macro step, because flows $a1$ and CRASH1 are connected to the outside world. Now only two internal processing sequences are possible. The first possibility is that the input from $a1$ is processed and an output on flows $B1, b1$ and $wrB1$ is produced. Because of the recursive causal chain condition $CCP1$ must consume the event on $wrB1$ and produce an event on $NEXT1$ *before* $CRASH1$ is processed by $CCP1$. The other possibility is that first $CRASH1$ is taken into account. Then $P1$ can not process the value on $a1$ and no output on $b1$ is produced. So the fault hypothesis is not modelled.

The most abstract semantics for this purpose within our setting is the *weakly fair interleaving semantics*. It models the following internal processing sequence: Event $CRASH1$ and a value on flow $a1$ are input of one macro step and output on flows $B1, b1$ and $wrB1$ is produced. Now the choice of processing input event $wrB1$ or $CRASH1$ is made non deterministically by $CCP1$. Therefore $CCP1$ can consume $CRASH1$ before $wrB1$ and no event $NEXT1$ will be produced.

With the *full interleaving semantics* it is possible to model the fault hypothesis, too, but with an inappropriately low level of abstraction. The following processing sequence is possible: Both $CRASH1$ and a value on flow $a1$ are input of one macro step. The value on flow $a1$ is processed. Now a new value on flow $a1$ placed before $CRASH1$ is taken into account. Therefore a situation where in spite of the occurrence of CRASH1 two inputs are processed by computer $CP1$ is modelled.

Acknowledgements

The research of Carsta Petersohn has been partially supported by DST. The research of Willem-Paul de Roever has been partially supported by ESPRIT BRA2 projects "SPEC" (no. 3096) and "REACT" (no. 6021).

References

[1] A. Benveniste and G.Berry. The synchronous approach to reactive and real-time systems. In *IEEE-Proceedings : Another look at Real-Time Programming*, 1992.

[2] G. Berry and G. Gonthier. The esterel synchronous programming language : Design, semantics, implementation. Technical report, Ecole Nationale Supérieur des Mines de Paris, 1988.

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

[4] C. Huizing and R. T. Gerth. Semantics of reactive systems in abstract time. In G. Rozenberg J.W. de Bakker, W.-P. de Rover, editor, *Real-Time: Theory in Practice, proceedings of a REX workshpo, June 1991*, LNCS 600, pages 291–314. Springer Verlag, Berlin, Heidelberg, 1992, June 1991.

[5] J . Peleska, C. Huizing, and C. Petersohn. A comparison of Ward&Mellor's TRANSFORMATION SCHEMA with STATE-&ACTIVITYCHARTS. Technical report, Eindhoven University of Technology, 1994.

[6] C. Petersohn, C. Huizing, J. Peleska, and W.-P. de Roever. Formal semantics for Ward & Mellor's TRANSFORMATION SCHEMAS. In D. Till, editor, *Sixth Refinement Workshop of the BCS FACS Group*. Springer Verlag, 1994.

[7] Carsta Petersohn. Formalisierung reaktiver Systeme mit Transformationsschemata sowie ein Vergleich mit Activity- und Statecharts. Master's thesis, Christian–Albrechts–Universität zu Kiel, 1992.

[8] G. Plotkin. An operational semantics for csp. In *In Proceedings of the IFIP Conference on the Formal Description of Programming Concepts II, North Holland*, pages 199–225, 1993.

[9] A. Pnueli and M. Shalev. What is in a step: On semantics of statecharts. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lect. Notes in Comp. Sci.*, pages 244–264. Springer-Verlag, 1991.

[10] G. Richter and B. Muffeo. Towards a Rigorous Interpretation of ESML – Extended Systems Modeling Language. *IEEE Transaction on Software Engineering*, 19(2):165–180, February 1993.

[11] Paul T. Ward. The Transformation Schema: An extension of the data flow diagram to represent control and timing. *IEEE TSE*, SE-12(2):198–210, February 1986.

[12] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*, volume 1-3 of *Yourdon Press Computing Series*. Prentice Hall, Englewood Cliffs, 1985.