Visual Formalisms Revisited*

R. Grosu, Gh. Stefănescu[‡], M. Broy

Institut für Informatik, TU München, D-80290 München, Germany

[‡]Department of Informatics, Kyushu University 33, Fukuoka 812-81, Japan

email: grosu,broy@informatik.tu-muenchen.de, ghstef@i.kyushu-u.ac.jp

Abstract

The development of an interactive application is a complex task that has to consider data, behavior, intercommunication, architecture and distribution aspects of the modeled system. In particular, it presupposes the successful communication between the customer and the software expert. To enhance this communication most modern software engineering methods recommend to specify the different aspects of a system by visual formalisms.

In essence, visual specifications are directed graphs that are interpreted in a particular way for each aspect of the system. They are also intended to be compositional. This means that, each node can itself be a graph with a separate meaning. However, the lack of a denotational model for hierarchical graphs often leads to the loss of compositionality. This has severe negative consequences in the development of realistic applications.

In this paper we present a simple denotational model (which is by definition compositional) for the architecture and behavior aspects of a system. This model is then used to give a semantics to almost all the concepts occurring in ROOM. Our model also provides a compositional semantics for or-states in statecharts.

1 Introduction

Recent advances in telecommunication and hardware technology made distributed, interactive applications into an important domain of concern of software construction. The development of an interactive application is, however, a complex task that has to consider data, behavior, intercommunication, architecture and distribution aspects of the modeled system. In particular, it presupposes the successful communication between the customer and the software expert.

To enhance this communication, most modern software engineering methods, such as Rhapsody [16], ROOM [22], SDL [20] and UML [21], recommend to specify the different aspects of a system by visual formalisms. In essence, all visual specifications are directed graphs that are interpreted in a particular context. In the data context the nodes define data-entities and the arcs define data relationships (e.g. entity-relationships diagrams [11]). In the behavior context the nodes define states and the arcs define state transitions (e.g. statecharts [14], ROOM-charts [22]). In the intercommunication context the nodes define processes and the arcs define events (e.g. message sequence charts [17]). In the architecture context the nodes define components and the arcs define data-flow paths (e.g. data-flow diagrams [25]). Finally, in the distribution context the nodes define components and computation resources and the arcs define the placement of components on resources (e.g. UML deployment diagrams).

All these visual specifications are *intended* to be *compositional*, i.e., each node can be a graph with a separate meaning. However, the lack of a denotational model for hierarchical graphs often leads to the loss of compositionality. This has severe negative consequences in the development of realistic applications.

In this paper we present a simple denotational model (which is by definition compositional) for the architecture and behavior aspects of a system. This model is then used to give a semantics to almost all the concepts occurring in ROOM. Our model also provides a compositional semantics for or-states in statecharts. In comparison with the compositional semantics for or-states given in [19], this semantics retains the full power of the higraph semantics [15].

To better appreciate the importance of a compositional model, let us give a small example showing the problems arising when using the hierarchical statechart notation.

A telephone switch (which resides in a telephone exchange) is supposed to control the function of an associated telephone. Its simplified overall behavior is

*This research was partially supported by the ESPRIT basic research action 8533 NADA

given in Figure 1. The state onHook consists of two sub-states: idle and ring. Idle is responsible for call initiation.

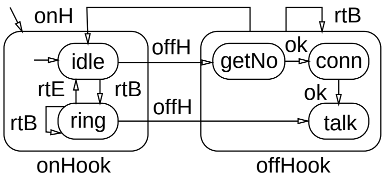

Figure 1: The telphone switch behavior

If the switch receives the signal off-hook offH while it is in state idle, then it moves to the state getNo and subsequently to conn and talk, if everything goes right. If the switch receives the signal ring-tone-begin rtB from another switch while it is in the state idle, then it moves to the state ring. Further signals rtB from another switch leave the switch in ring. The receipt of the signal ring-tone-end rtE from the original switch determines the switch to move back to the state idle. However, the receipt of the signal offH initiates a talking session by moving the control to the state talk. At any time, if the switch is in the state offHook and it receives the signal on-hook onH, then it moves back to the state idle. Moreover, any signal rtB received by the switch while it is in the state offHook leaves the state unchanged.

The problem with this hierarchical diagram is that the transitions labeled by offH and onH connect states inside onHook and offHook. If we want to reason in a compositional way, for instance, if we want to hide the states inside these two composed states, we loose the starting and the ending points of the corresponding transitions.

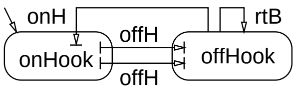

Figure 2: The switch behavior with stubs

A single transition labeled by offH and connecting the composed states onHook and offHook is clearly undesirable, because the behavior associated to the original transitions is definitely different. Moreover, a transition labeled by onH and connecting the composed states offHook and onHook is also problematic without some default assumptions, because the original transition ended in a particular sub-state of onHook, namely idle. As a consequence, one either draws the diagram as shown above, or uses a patch, the so called *stubs*. With stubs, the top level diagram looks as shown in Figure 2. The semantics of these stubs is, however, not clearly defined. Moreover, in this case we have a new kind of nodes, the stubs, and two new kinds of arcs, arcs connecting stubs and arcs connecting states and stubs.

The main problem with this notation is the absence of a clearly defined interface notion. Using a graph formalism that explicitly supports interfaces, stubs get a very natural interpretation. They are *interface points*.

The rest of the paper is organized as follows. In Section 2 we describe our graph formalism in an abstract setting. This formalism is then instantiated to the architecture and to the behavior aspects in Sections 4 and 5. The instantiation is guided by the computation model described in Section 3. Finally in Section 6 we summarize the results of this paper and relate them to the literature. To introduce and explain our formalism in an intuitive way we use a running example throughout the paper: the specification of a telephone exchange. The paper also contains two appendices giving an additive and a multiplicative interpretation of graphs.

2 Hierarchical Graphs

A *hierarchical graph* consists of a set of *nodes* connected by a set of *arcs*. For each node, the incoming and the outgoing arcs define the node's *interface*. In general the arcs have associated some *type information*.

Suppose T is a set of *type names* and D is a *type function* mapping each name $t \in T$ to an associated *domain* of values D_t. Since we want to speak about incoming and outgoing arcs collectively we assume given a binary (monoidal) operation $\star$, with neutral element $\mathbf{e}$, both on type names and on the corresponding domains. For types, we obtain the set of terms defined by the following grammar:

$$a \quad ::= \quad t \in T \mid \mathbf{e} \mid a \star a \mid (a)$$

The $\star$ operation on domains is assumed to be compatible with the $\star$ operation on type terms, hence:

$$D_{a\star b} \;=\; D_a \star D_b \qquad D_{a\star \mathbf{e}} \;=\; D_{\mathbf{e}\star a} \;=\; D_a$$

Now a node N with incoming arcs that collectively have type a and outgoing arcs that collectively have type b can be interpreted as a *relation* $N \subseteq D_a \times D_b$. Visually, this is represented by a box labeled by N and with an incoming arrow labeled by a and an outgoing arrow labeled by b. We write $N : a \to b$. If we define $|t|$ as the number of names occurring in the term t, then $|a|$ gives the number of incoming arcs and $|b|$ gives the number of outgoing arcs of N.

Operators on nodes. In order to form graphs, we put nodes one next to another and interconnect them by using the following operators on nodes: *star composition, sequential composition* and *feedback.* Their visual representation is given in Figure 3.

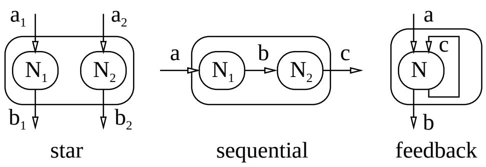

Figure 3: The composition operators

The *star composition* is achieved by extending $\star$ to an operation over nodes. Given $N_1 : a_1 \to b_1$ and $N_2 : a_2 \to b_2$ we define $N_1 \star N_2$ to be of type $a_1 \star a_2 \to b_1 \star b_2$.

The *sequential composition* corresponds to the usual composition of relations. Given $N_1 : a \to b$ and $N_2 : b \to c$ we define $N_1 \,;\, N_2$ to be of type $a \to c$.

The *feedback* operation allows to connect the output of a node to the input of the same node, if both have the same type. Given $N : a \star c \to b \star c$ we define $N{\uparrow}^c_\star$ to be of type $a \to b$.

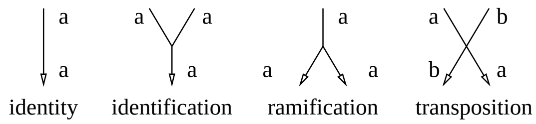

Figure 4: The connectors

Operators on arcs. Beside operators on nodes, we also need some operators on arcs, that we call *connectors.* We consider the following connectors: *identity, identification, ramification* and *transposition.* Their visual representation is given in Figure 4.

The *identity* connector I_a simply copies its input to the output. Hence, it has the type $a \to a$.

The binary *identification* connector $\vee_a$ joins two inputs together. Hence $\vee_a$ has the type $a \star a \to a$. This operator is naturally extended to k inputs. For $k \geq 1$ it is written $\vee^k_a$. For $k = 0$ it is written either $\top_a$ or $\vee^0_a$.

The binary *ramification* connector $\wedge^a$ distributes the input information on two outputs. Hence $\wedge^a$ has the type $a \to a \star a$. This operator is naturally extended to k outputs. For $k \geq 1$ it is written $\wedge^a_k$. For $k = 0$ it is written $\perp^a$ or $\wedge^a_0$.

Finally the *transposition* connector ${}^a\mathsf{X}^b$ exchanges the inputs. Hence ${}^a\mathsf{X}^b$ has the type $a \star b \to b \star a$

Symmetric feedback. Using the above basic operators and connectors one can define, as shown in Figure 5, left, a *derived* composition operator, the *symmetric feedback.* If $N : a \star c \to b \star d$ and $M : d \star e \to c \star f$ then $N \circledast M$ has type $a \star e \to b \star f$. Its simplified visual representation is given in Figure 5, right. The formal definition corresponds one to one to the visual representation in Figure 5, left:

$$\mathrm{N} \circledast \mathrm{M} = ((((\mathsf{I}_a \star {}^e\mathsf{X}^{c \star d}); (\mathrm{N} \star \mathrm{M}); (\mathsf{I}_b \star {}^{d \star c}\mathsf{X}^f); (\mathsf{I}_{b \star f} \star {}^d\mathsf{X}^c)){\uparrow}^d_\star){\uparrow}^c_\star$$

The symmetric feedback operator often simplifies both the visual notation and the associated graph expression. Moreover, it plays a central role in Abramski's semantics of interaction [2, 1].

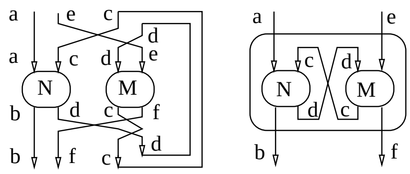

Figure 5: The symmetric feedback

To be a precise formalization of graphs, the above basic operators and connectors have to satisfy a set of laws, which intuitively express our visual understanding of graphs. These laws correspond to *symmetric monoidal categories with feedback* enriched with branching constants, see e.g. [23][1]. The very basic setting without branching constants, but extended from symmetric to balanced categories is given in [18]. A category obeying these laws is called a *trace monoidal category* there. Such a category also contains *associativity isomorphisms* for $\star$. To simplify notation, they are never written explicitly and assumed present, when necessary.

In this work we are interested in two particular interpretations for $\star$ that are highly relevant in computer science and satisfy the graph laws: the additive interpretation and the multiplicative interpretation[2].

The additive interpretation. This interpretation corresponds to *control-flow* in sequential programs as follows. At any moment of time, the control resides in *exactly one* node. The node receives the control on one of its *disjoint entry points* and gives the control back on one of its *disjoint exit points.* The arcs of the graph then forward the control to another node. The intended disjointness of nodes, entry/exit points and branches of the connectors is obtained by interpreting $\star$ by the disjoint sum $+$ and by defining the other

[1]The basic laws were given by Stefanescu (1986) and were extended to various branching constants by Bergstra, Cazanescu and Stefanescu.

[2]They are sometimes called the temporal and the spatial interpretation respectively.

operators and connectors consistently with this interpretation (see Appendix A). The additive connectors are written as $\mathsf{I}_a, {}_k{>\!\!\bullet}_a, {}_a{\bullet\!\!<}_k$ and ${}^b_a\mathsf{X}$.

The multiplicative interpretation. This interpretation corresponds to *data-flow* in parallel programs as follows. At any moment of time, *all* nodes of the graph are active and computing the output data based on the input data. A node receives the input data along a *tuple of input channels* and sends the computed data along a *tuple of output channels*. The arcs of the graph, i.e., the channels, forward the data to the other nodes in the graph. The intended parallelism of nodes, input/output channels and branches of the connectors is obtained by interpreting $\star$ by the product $\times$ and by defining the other operators and connectors consistently with this interpretation (see Appendix B). The multiplicative connectors are written as $\mathsf{I}_a, \curlyvee^k_a, \curlywedge^a_k$ and ${}^a\mathsf{X}^b$.

Both the additive and the multiplicative interpretations were already studied in isolation by the authors, e.g. in [6, 8, 10, 12, 13, 23]. and also by other researchers like Abramski, Bartha, Bergstra, Bloom, Cazanescu, Elgot, Esik, Joyal, Milner, Stark, Street, Verity etc.

The combination of data-flow networks with state-transition diagrams was also studied e.g. in [7, 9]. However, the way we combine the additive and the multiplicative interpretations in this paper to obtain a state-based description of reactive systems is new and it is guided by the computation model presented in the next section.

3 The Computation Model

We model an *interactive system* by a network of autonomous components that communicate time-synchronously via directed channels (Figure 6, left). Time synchrony is achieved by using a *global clock*.

A *component* is modeled by a Moore-machine (Figure 6, right). This machine consists of three parts: a *combinational* part, a *register* and a *feedback loop*.

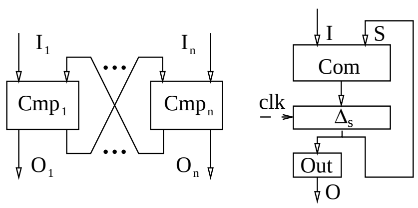

Figure 6: The Moore-machine computation model

The combinational part. The combinational part consists of two elements: *Com* and *Out*. Both have *no memory*. *Com* is concerned exclusively with *control*. It *instantaneously* and (possibly) *nondeterministically* maps the current state and the current input to the next state. If $\mathcal{S}$ is the set of states and $\mathcal{I}$ is the set of inputs then *Com* is a relation,

$$Com \in \mathcal{I} \times \mathcal{S} \to \mathcal{P}(\mathcal{S})$$

which is *total*, i.e., $Com(i, s) \neq \emptyset$, for all s, i. To emphasize input/output and totality, we write the relation in a functional style. In Section 5 we define *Com* by an additive *control-flow graph* with nodes of the form $N \subseteq (\mathcal{I} \times \mathcal{S}_a) \times \mathcal{S}_b$ where a and b are sum terms. The only apparent mismatch of this additive interpretation with the abstract graph-definition is the additional input component $\mathcal{I}$. However, by considering that $\mathcal{I}$ is the same for all the nodes representing the combinational part, the extension to relations of this type is trivial. Note that in contrast to state transition diagrams, our control-flow graphs model instantaneous computation.

The output relation $Out \in \mathcal{S} \to \mathcal{P}(\mathcal{O})$ is usually a function that (instantaneously) projects the current state to the component's output interface.

The component. The register Δ_s and the feedback loop model the *memory*, i.e., they add the *temporal dimension*. As long as the global clock does not tick, the current state, input and output remain stable. This leaves enough time for the combinational part to perform its computation. Moreover, it assures a stable input for the components receiving the outputs of this component. The arrival of a clock-tick *clk* updates the current state and input and a new computation cycle begins.

Associating a natural number with each computation cycle and assuming that a component never stops, the input and the output of the component are infinite sequences in $\mathcal{I}^{\mathbb{N}}$ and $\mathcal{O}^{\mathbb{N}}$ respectively, which we call the *input* and the *output histories* of the component.

For *any* initial state, we formally define a component *Cmp* as a *total* relation between its input and output communication histories. As before, to emphasize input/output and totality, we write this relation in functional notation. Let us define the timed extension $R^{\dagger}$ of a combinational relation R and the delay relation Δ_s (the register) as follows:

$$\begin{aligned} \Delta_s(x) \quad &= \{y \mid y(0) = s \wedge \forall t > 0 : y(t) = x(t)\} \\ R^{\dagger}(x) \quad &= \{y \mid \forall t : y(t) \in R(x(t))\} \end{aligned}$$

Then using the multiplicative graph-operators, Δ_s, $Com^{\dagger}$ and $Out^{\dagger}$ the formal definition of *Cmp* corresponds one-to-one to the Figure Figure 6, right:

$$\begin{aligned} Cmp \quad &\in \quad \mathcal{S} \to \mathcal{I}^{\mathbb{N}} \to \mathcal{P}(\mathcal{O}^{\mathbb{N}}) \\ Cmp(s) \quad &= (Com^{\dagger}; \Delta_s; \curlywedge_2; (Out^{\dagger} \times \mathsf{I}))\uparrow_{\times} \end{aligned}$$

Hence, Cmp is the extension in time and with memory of Com. This matches Abramski's slogan: *processes are relations extended in time.* A relation like Cmp whose output depends only on the input received *before* is called *strongly time guarded.* This property makes the multiplicative feedback operator well defined.

The system. For a given initial state s_0, the relation $Cmp(s_0)$ defines a component with an encapsulated private state. In Section 4 we define a system by a multiplicative *data-flow graph* where each node represents a component, and each arc represents a data flow, i.e., a communication history. We also call such a diagram the *architecture specification* of the system.

Note on semantics. A very important characteristic of our semantic model is its uniform use of the relational framework. This has two beneficial consequences. First, it considerably simplifies the semantic definition. Second, it allows us to apply the well established set of graph operators to compose relations.

Another important aspect is the totality of Com, which also implies the totality of Cmp. Totality is also called *reactivity* and it assures that composing two components does not lead to an empty relation, provided that the components are not empty. This is essential both for modular system development and for modular proofs about a system.

4 Architecture Specification

The architecture specification is given by a hierarchical data-flow graph. Each node in the graph is a component acting in parallel with the other components and each arc in the graph is a channel describing the data flow from the source component to the destination component.

The data-flow graphs are constructed by using the operators formally defined in Appendix B. In the example below, we shall only use parallel composition $\times$ and symmetric multiplicative feedback $\otimes$. As with associativity isomorphisms, to simplify notation we assume the necessary transpositions, too.

Example 1 (Telephone Exchange) *Suppose we want to specify a telephone exchange, whose architecture is given in ROOM-notation in Figure 7.*

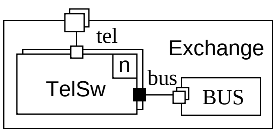

Figure 7: The architecture of the exchange

It consists of n telephone switches, each monitoring an associated telephone and communicating with the other switches along an internal bus. The main task of the exchange is to connect a caller to an idle callee upon receipt of a four digit number. We consider the following sets of input and output messages on the telephone interface of the switch:

$$\begin{array}{l} \texttt{TelI ::= tk} \mid \texttt{onH} \mid \texttt{offH} \mid \texttt{dig}(\mathbb{I}) \\ \texttt{TelO ::= tk} \mid \texttt{dtB} \mid \texttt{dtE} \mid \texttt{rtB} \mid \texttt{rtE} \mid \texttt{rbB} \mid \texttt{rbE} \end{array}$$

where | denotes as usual alternatives, i.e., it is the same as +. A telephone may send the signals on hook `onH`*, off hook* `offH`*, a digit n* `dig`*(n) and an abstract talking signal* `tk`*. The switches enclosed in the exchange send pairs of signals of the form* signal`B` *(signal Begin) and* signal`E` *(signal End). The pairs are dial tone* `dt`*, ring tone* `rt` *and ring back tone* `rb`*. The switches may also forward the abstract talk signal* `tk`*. Along the bus interface the switches exchange the sets of messages* `BusI = BusO` *where:*

$$\texttt{BusI ::= tk}(\mathbb{I}) \mid \texttt{onH}(\mathbb{I}) \mid \texttt{rtB}(\mathbb{I}) \mid \texttt{rtE}(\mathbb{I}) \mid \texttt{bsy}(\mathbb{I})$$

The integer denotes the number of the destination switch when the message is sent by the switch and the number of the source switch when the message is sent by the bus. Given the above types, the switches and the bus are components of the following types:

$$\begin{array}{lll} \texttt{TelSw} & \in (\texttt{TelI} \times \texttt{BusI})^{\mathbb{N}} & \rightarrow \mathcal{P}((\texttt{BusO} \times \texttt{TelO})^{\mathbb{N}}) \\ \texttt{BUS} & \in (\texttt{BusO}^n)^{\mathbb{N}} & \rightarrow \mathcal{P}((\texttt{BusI}^n)^{\mathbb{N}}) \end{array}$$

The telephone exchange is then defined by their composition as below:

$$\begin{array}{ll} \texttt{Exchange} & \in (\texttt{TelI}^n)^{\mathbb{N}} \rightarrow \mathcal{P}((\texttt{TelO}^n)^{\mathbb{N}}) \\ \texttt{Exchange} & = (\times_{i=1}^{n} \texttt{TelSw}) \otimes \texttt{BUS} \end{array}$$

Note the close relation between the formula and the visual representation given in Figure 7. For the product of the n telephone switches one can use as in ROOM a distinct visual notation that emphasizes that all the components in the product are identical. □

5 Component Specification

In many modern software-engineering methods the behavior of a component is given by a state-chart-like diagram. The semantics of such a diagram is, however, complicated by the use of hierarchy in absence of interfaces and by the use of additional features like entry/exit actions and history variables. The absence of interfaces also severely reduces their usefulness. In this section, we give a denotational semantics to ROOM-charts in a simple and modular way by using hierarchical control-flow graphs. ROOM-charts are the only visual notation we know that explicitly supports interfaces[3].

Control-flow graphs describe the behavior of the combinational part of a component. Since this behavior is extended in a canonical way to component behavior, specifying the combinational part amounts to specifying the component itself. Let us first be more precise about state and input.

State and input. A *state* consists of a mapping of *latched* (or *controlled*) variable names to values of corresponding type. Let S denote the set of controlled variable names with associated domains $\{D_v \mid v \in S\}$. Then the set of all associated states is given by $\mathcal{S} = \prod_{v \in S} D_v$.

The variables occurring in the state can be further split in two disjoint sets: a set P of *private* variables and a set O of *output* (or *interface*) variables. We write $\mathcal{S}_P$ for $\prod_{v \in P} D_v$ and $\mathcal{S}_O$ for $\prod_{v \in O} D_v$. Clearly, $\mathcal{S} = \mathcal{S}_P \times \mathcal{S}_O$.

The input is also a mapping of *input* variable names to values of corresponding type. Let I denote the set of input variable names with associated domains $\{D_v \mid v \in I\}$. Then the set of all associated inputs is given by $\mathcal{I} = \prod_{v \in I} D_v$.

We are now prepared to define the semantics of ROOM-charts. As a consequence of this semantics, we can safely use ROOM-charts to define components. Moreover we can reason about ROOM-charts in an abstract mathematical setting.

Arrows with common source or common destination. Arrows with common source are modeled by using the additive ramification connector. Arrows with common destination are modeled by using the additive identification connector.

Computation units. A "simple state" is drawn in ROOM as shown in Figure 8, right. It may have an entry action which is marked as $\rightarrow\circ$, an exit action which is marked as $\circ\rightarrow$ and may perform the transitions $\mathtt{ac}_1, \ldots, \mathtt{ac}_n$.

We model a simple ROOM-state by a *computation unit* whose flow-graph is shown in Figure 8, left. The computation unit gets the *control* along one of its *entry points* and gives the control back along one of its *exit points.*

After getting the control along an entry point en_i, the computation unit forgets the entry point information (it is not relevant in this case) and then it executes an *entry action.* Then it evaluates a set of *action guards.* If one of the guards is true, than the corresponding *action* is said to be *enabled* and its *body* may be executed. After finishing its execution, the computation unit may also execute an *exit action.* Finally, the control is given to another computation unit along the exit point corresponding to the executed action.

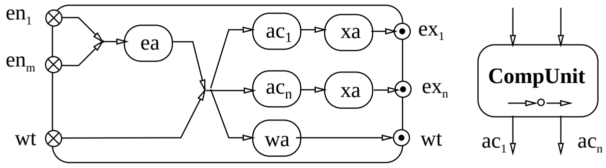

Figure 8: The architecture of a computation unit cU

If more than one guards are true, then the computation unit *nondeterministically* chooses one of them. If none of the guards is true, i.e., if the computation unit would block, then `wa` is true and the discrete computation is completed by leaving not only the computation unit but also the enclosing combinational part along the exit point `wt`. The computation unit $\mathtt{cU} \subseteq \mathcal{I} \times \mathcal{S}_{en_1+\ldots+en_m+wt} \rightarrow \mathcal{P}(\mathcal{S}_{ex_1+\ldots+ex_n+wt})$ is defined formally by the following formula:

$$\mathtt{cU} \stackrel{\text{def}}{=} ({}_m\!\succ\!\bullet; \mathtt{ea} + \mathsf{I}); {}_2\!\succ\!\bullet; \bullet\!\prec_{n+1}; ((+_{i=1}^{n} \mathtt{ac}_i; \mathtt{xa}) + \mathtt{wa})$$

Waiting. The control passed by `cU` along the wait exit point `wt` to the register has the form $\mathtt{wt}.s$. It is the injection of the state $s \in \mathcal{S}$ in the sum $\mathcal{S}_{ex_1+\ldots+ex_n+wt}$. Since the injections are unique both inside a computation unit and inside the enclosing combinational part, the multiplicative part gives the control $\mathtt{wt}.s$ to the computation unit associated with `wt` in the next time slice back. Hence the combinational part resumes execution exactly where the computation was suspended. Passing control between the additive and the multiplicative parts models *waiting.* Since waiting usually happens because of interaction (if all guards involve the input and all of them are false), we are able to model interaction in a purely denotational setting. Contrast this with the structural operational semantics, where the state of the transition relation has to contain additionally the program expression.

Note that adding an implicit wait entry/exit point `wt` and an implicit wait action `wa` assures the totality of *simple* computation units. However, it does not assure the totality of computation units involving feedback. Feedback control is either the specifier's responsibility or the responsibility of a compiler, like in Esterel [5].

Actions. An *action* a is a relation between the current state, the current input and the next state:

$$a \subseteq (\mathcal{I} \times \mathcal{S}) \times \mathcal{S}$$

[3]The usefulness of interfaces was actually recognized and implemented in Statemate, the official tool for statecharts, too.

We specify actions by their characteristic predicate. We use *back-primed variables* to denote the current input, *primed variables* to denote the next state and *plain variables* to denote the current state. Moreover, we mention only the changed variables and always assume the necessary equalities stating that the other variables did not change.

Events and message passing. Latched variables allow us to model many different communication styles. Particularly interesting are events and message passing. We model *events* by toggling boolean variables. The occurrence of an event is detected by testing if the current input value for that variable is different from the latched value of that variable, i.e., $e` \neq e$, where $e \in \mathbb{B}$ signals the occurrence of the event e. We abbreviate the above expression by $e?$. Similarly, sending an event is given by the following expression $e' = \neg e$ which is abbreviated by $e!$.

To model *message passing* we associate with each channel c a pair (e, m) consisting of an event variable e and a message variable m. The arrival of a message a on the channel c is given by the following expression: $e? \wedge m` = a$. We abbreviate it by $c?a$. Similarly sending the message a on the channel c is given by $e! \wedge m' = a$ which is abbreviated by $c!a$.

Example 2 (Computation Units) *Let us define the behavior of the telephone switch in the state* `ring`*. The switch has four unidirectional channels which we denote by* `teli`*,* `telo`*,* `busi` *and* `buso` *and model them as explained above.*

An interesting aspect of this computation unit is the way it handles events that take time. Intuitively, as long as control is in the ringing "state", the telephone should be ringed. This is accomplished in a message passing formalism with a pair of messages `rtB` *and* `rtE` *as follows. Whenever control arrives at the computation unit* `ring` *the message* `rtB` *is sent. This is performed by an entry action. Whenever control leaves the computation unit* `ring` *the message* `rtE` *is sent. This is performed by an exit action.*

The top level specification of this computation unit is given below. Its visual representation inside the state `onHook` *is given in Figure 9, left. Let us abbreviate $k\bullet$ by k. Then:*

$$\mathtt{ring} \subseteq (\mathcal{I} \times \mathcal{S}_2) \ \times \ \mathcal{S}_3$$

$$\begin{aligned}\mathtt{ring} \equiv\ & (\mathit{ea} + \mathsf{I});{}_2{\succ}\!\bullet;\bullet\!{\prec}_4;\\ & ((\mathit{wa} + \mathtt{rtB});{}_2{\succ}\!\bullet + \mathtt{rtE}; \mathit{xa} + \mathit{ans}; \mathit{xa})\end{aligned}$$

All actions are relations in $(\mathcal{I} \times \mathcal{S}) \ \times \ \mathcal{S}$. The entry and the exit actions of the computation unit `ring` *are described as follows:*

$$\mathit{ea} \equiv \mathtt{telo!rtB} \quad \mathit{xa} \equiv \mathtt{telo!rtE}$$

The other actions are as expected. The state variable nr memorizes the destination switch. Note that we often use the same name for an action and the message it is waiting for.

$$\begin{aligned}
\mathit{wa} &\equiv \neg(\mathtt{teli?offH} \vee \mathtt{busi?rtB}(n) \vee \mathtt{busi?rtE}(nr))\\
\mathtt{rtB} &\equiv \mathtt{busi?rtB}(n) \wedge nr \neq n \wedge \mathtt{buso!bsy}(n)\\
\mathtt{rtE} &\equiv \mathtt{busi?rtE}(n) \wedge nr = n\\
\mathit{ans} &\equiv \mathtt{teli?offH}
\end{aligned}$$

The above computation unit defines the ROOM-charts in Figure 9, right. Note that in ROOM transitions like `rtB`*, which give control to the enclosing multiplicative structure, are drawn as a loop. Moreover,* `rtB` *is drawn inside the "state"* `ring` *to emphasize that no entry and exit actions are performed for* `rtB`*.* □

Hierarchical states. Hierarchical states of ROOM-charts are obtained by composing computation units. For this purpose, we use the symmetric feedback operator and implicitly assume the necessary transpositions.

Example 3 (The state onHook) *The hierarchical computation unit* `onHook` *consists, as shown in Figure 9, of two interconnected computation units,* `idle` *and* `ring`*.*

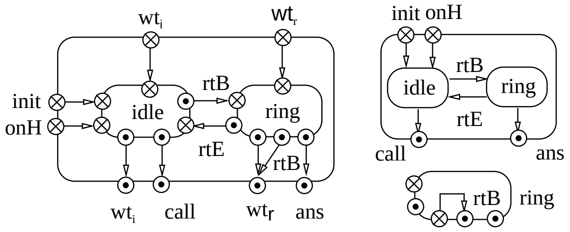

Figure 9: The computation unit onHook

The mathematical equivalent is the following relational expression:

$$\mathtt{onHook} \equiv \mathtt{idle} \oplus \mathtt{ring}$$

Note that the semantics of the sum automatically assures that upon entry, the control is either given to `idle` *or to* `ring` *according to the control's injection.* □

Entry and exit actions of composed states. A composed computation unit may also have entry and exit actions. This is especially useful when dealing with pairs of events. For example if we would like to limit the connection and talking time to 60 minutes, we could set a timer when entering the composed state `offHook` and reset it when leaving this state.

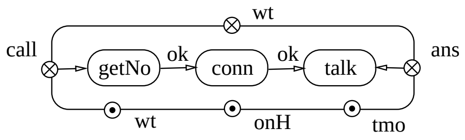

Fig 10: Entry/exit actions for offHook

In this case, as shown in ROOM-notation in Figure 10, the computation unit `offHook` has three entry points, `wt, call` and `ans` and three exit points, `wt, onH` and `tmo`. Entry and exit actions for composed computation units are dealt with similarly to simple computation units. Care is necessary if we do not want to identify different entry points. In that case, we use a technique similar to the exit actions in the previous section. For `offHook` we have:

$$\mathtt{offHook} = (2\,\mathsf{ea} + \mathsf{I}); (\mathtt{getNo}\oplus\mathtt{conn}\oplus\mathtt{talk}); (2\mathsf{xa} + \mathsf{I})$$

where $+_{i=1}^{n} R$ is abbreviated by $n\,R$. At the beginning and at the end of `offHook`, the identity handles the transitions `wt`. If `to` is the considered timer, than the entry and the exit actions may have the following form:

$$\mathtt{ea} = \mathtt{to!set}(60), \quad \mathtt{xa} = \mathtt{to!reset}$$

Transitions to composed states. The transition `onH` from the composed state `offHook` to the composed state `onHook` as shown in Figure 2 was problematic, because on that level we did not have enough information to unambiguously determine the destination state. Using sums, the complete information is contained in the interface of the composed computation units. Defining:

$$\begin{aligned} \mathtt{offHook} &= \mathtt{getNo} \oplus \mathtt{conn} \oplus \mathtt{talk} \\ \mathtt{telSw} &= \mathtt{onHook} \oplus \mathtt{offHook} \end{aligned}$$

we can safely abstract from the internal structure of these computation units and draw the corresponding diagrams as shown in Figure 11.

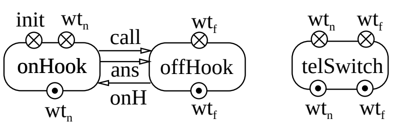

Figure 11: Top level graph of the switch

The interface of `offHook` automatically directs `call` to `getNo` and `ans` to `talk`. Similarly, the interface of `onHook` directs `onH` to `idle`. The `wt` transitions are dealt with similarly on the top-level defined by `telSwitch`.

Transitions from composed states (preemption). The transition `onH` is a transition from the composed state `offHook` to the composed state `onHook`. The usual meaning is that from each substate, that transition is taken, if it was not overridden. In terms of computation units this means that each computation unit has the exit point `onH`.

Although this semantics clearly does its job, it has a big disadvantage: it is *not modular*. Adding a common transition automatically implies the change of code of all the computation units involved in the sum giving the composed computation unit. Fortunately, in our formalism there are three ways to deal with common transitions in a modular way. They are shown in Figure 12. The first one can be understood as an *entry action*; the second one as an *exit action*; the third one as an *alternative action*. Let the actions corresponding to the transition `onH` be given as follows:

$$\mathtt{onHa} \;\equiv \mathtt{teli?onH}, \qquad \mathtt{onHw} \;\equiv \neg\,\mathtt{teli?onH}$$

The three ways of achieving a common transition `onH` correspond to the following expressions:

$$\begin{aligned} \mathtt{OffE} &\equiv ({}_3{\bullet\!\!<}_2; (3\mathtt{onHw} + ({}_3{>\!\!\bullet}_1; \mathtt{onHa}))); (\mathtt{offHook} + \mathsf{I}) \\ \mathtt{OffX} &\equiv \mathtt{offHook}\,; ({}_1{\bullet\!\!<}_2\,; (\mathtt{onHw} + \mathtt{onHa})) \\ \mathtt{OffA} &\equiv {}_3{\bullet\!\!<}_2\,; (\mathtt{offHook} + ({}_3{>\!\!\bullet}_1\,; \mathtt{onHa})) \end{aligned}$$

The first expression corresponds to the strong preemption semantics for statecharts: the common transition `onH` has greater priority than any nested transition `onH`. The second expression corresponds to the weak preemption semantics for ROOM-charts: the common transition `onH` has lower priority than any nested transition `onH`. It is therefore better suited to incremental OO-design. The third expression allows nondeterministic choice between the common and the nested transitions `onH`. Each of these semantics may be useful in different contexts.

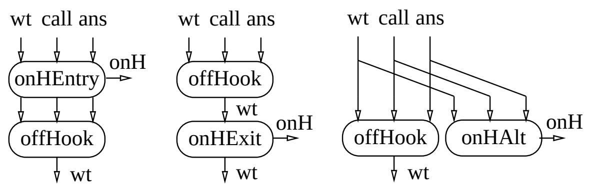

Figure 12: The preempting transition onH

The above expressions can be generalized to arbitrary composed states in the obvious way.

History variables (interrupts). Suppose the transition `int` interrupts the computation unit `offHook` on the occurrence of some special event and gives the control to the computation unit `admin`. The computation unit `admin` does some processing (which can consume time and require interaction with other components) and then returns the control with the transition `ret` to the same state, at which `offHook` was interrupted. Then, the interrupt processing unit `admin` has to be, similarly to exit actions, generic with respect to the current state, i.e., it should not destroy the information about the current exit point. Hence, `admin` must have the following form:

$$\mathtt{admin} = +_{i=1}^{n}\ \mathtt{intProcessing}_i$$

where n gives the number of components in the interrupted computation unit. This interface information, is already contained in the `wt` exit points ($n = |wt|$) and can be directly used if one chooses the weak preemption semantics for the transition `int`.

Now, by requiring that `int` and `ret` have the same type, after the interrupt processing is finished, control is returned to the right computation unit inside `offHook`. This gives a very concise, formal meaning to history variables.

Embedding in the multiplicative part. The top level flow-graph of the combinational part has only one entry and one exit point: the composed wait point. The multiplicative part receives the control on this point and gives it back to the additive part.

Let $n = |wt|$. Then the the function `Out` from Section 3 is defined by $\mathtt{Out} = {}_{n}{\succ\!\bullet}\,;\,|_{O}$. It first identifies the control points and then projects them on the output interface O.

6 Conclusions

In this paper we showed how to combine control (the additive graph interpretation) with data-flow (the multiplicative graph interpretation) by using the well established Moore-machine computation model. The result is a mixed additive-multiplicative graph interpretation that opens new perspectives both from practical and from theoretical point of view.

From the practical point of view, the mixed interpretation provides a very concise compositional semantics for ROOM-charts. It not only justifies the informal use of interface points in ROOM-charts but also shows how to define or-states in a compositional way in statecharts. In fact, we put on a solid, formal basis the original idea of Harel about or-states. Finally, the mixed interpretation also shows how to integrate architectural specifications with state-based component specifications.

From the theoretical point of view, the mixed interpretation extends the additive and the multiplicative algebras that were studied in isolation in [23] and in a categorical setting in [18]. While the additive algebra gives a semantics to control by ignoring interaction, and the multiplicative algebra gives a semantics for interaction by ignoring control, the mixed algebra gives a semantics to both control and interaction. An axiomatization of this algebra was already started in [24].

An important aspect of our algebra is that it uses the operators advocated by Abramski in [2, 1] for a formal foundation of interaction. Hence, our algebra has deep connections to game theory and linear logic. It is also connected to the reactive modules and the automata of Alur and Henzinger [4, 3]. Hence, we expect to be able to extend the associated model checking algorithms for hierarchical automata.

Because of space limitation we have omitted in this paper two important concepts: parallel composition inside the combinational units and network-architecture manipulation.

Parallel composition inside the combinational unit corresponds to the parallel composition inside statecharts and Esterel modules. Although one can argue (like in ROOM) that such an operator is not needed in practice, we believe that such an operator is useful as a handsome abbreviation of more involved diagrams. In fact, for the combinational part, parallel composition reduces to additive symmetric feedback. The reason is, that the combinational part performs computation instantaneously, i.e., for the combinational part the equation $a\times b = (a\times I);(I\times b) + (I\times b);(a\times I)$ is true. Hence, in order to give a semantics to parallel composition inside statecharts, we do not need to modify our model.

The model proposed so far can be understood as the alternation $\Sigma\,\Pi\,\Sigma\,\Pi$ of additive and multiplicative interpretations. The first Σ corresponds to data-type definitions. The first Π corresponds to states. The second Σ defines hierarchical state transition diagrams (or sequential programs). Finally, the second Π defines components and communication. Now, if we add a Σ on top of the second Π, we obtain control over components, i.e., we can express architecture control. This not only allows us to express object-oriented concepts, but also concepts occurring in mobile systems like network reconfiguration and process migration.

References

[1] S. Abramski. Retracing some paths in process algebra. In *Seventh International Conference on Concurrency Theory (Concur'96), Lecture Notes Computer Science 1055*, pages 21–33, 1996.

[2] S. Abramski, S. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concur-

rent programming. To appear in Proc. Marktoberdorf Summer School, 1994.

[3] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. To appear in the Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1997), 1997.

[4] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[5] G. Berry and G. Gonthier. The synchronous programming language ESTEREL: Design, semantics, implementation. *Sience of Computer Programming*, 19(2):83–152, 1992.

[6] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[7] M. Broy. Mathematical system models as a basis of software engineering. *Computer Science Today*, 1995.

[8] M. Broy. Towards a mathematical model of a component and its use. *Software—Concepts and Tools*, 18:137–148, 1997. Also appeared in Proc. of Componentware Users Conference 1996, Munich.

[9] M. Broy, R. Grosu, and C. Klein. Reconciling real-time with asynchronous message passing. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME '97, 4th International Symposium of Formal Methods Europe, Graz, Austria, Lecture Notes in Computer Science 1313*. Springer, September 1997.

[10] M. Broy and Gh. Stefănescu. The algebra of stream processing functions. Technical Report TUM-I9620, Technische Univerität München, 1996.

[11] P. Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[12] R. Grosu and K. Stølen. A Model for Mobile Point-to-Point Data-flow Networks without Channel Sharing. In Martin Wirsing and Maurice Nivat, editors, *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology, AMAST'96, Munich, Germany*, pages 505–519. Lecture Notes in Computer Science 1101, 1996.

[13] R. Grosu and K. Stølen. Specification of Mobile Systems. In M. Haveraaen and O. Owe, editors, *Proceedings of the 8th Nordic Workshop on Programming Theory, Oslo, Norway*, pages 67–76. Department of Informatics at the University of Oslo, 1996.

[14] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[15] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[16] D. Harel and E. Gery. Executable object modeling with statecharts. In *18th International Conference on Software Engineering*, pages 246–257, Berlin - Heidelberg - New York, March 1996. Springer.

[17] ITU-T. *Z.120 – Message Sequence Chart (MSC)*. ITU-T, Geneva, 1996.

[18] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.*, 119:447–468, 1996.

[19] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*. Springer Verlag, LNCS 630, august 1992.

[20] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92*. Elsevier Science, North-Holland, 1994.

[21] Rational. UML Notation guide, version 1.0. Response to the OMG's OOA&OOD RFP, January 1997.

[22] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[23] Gh. Stefănescu. Algebra of flownomials. Technical Report TUM-I9437, Technische Universität München, 1994.

[24] Gh. Stefănescu. Reaction and control I: Mixing additive and multiplicative network algebras. Technical Report 37/1996, Institute of Mathematics of the Romanian Academy, 1996. To appear in Journal of the Interest Group in Pure and Applied Logic.

[25] E. Yourdon. *Modern Structured Analysis*. Englewood Cliffs, Yourdon Press, New Jersey, 1989.

A The Additive Interpretation

This additive behavior is obtained by instantiating the abstract operator $\star$ to the *disjoint sum* operator $+$ and the type mapping D to the state space $\mathcal{S}$. Since $\mathcal{S}$ is independent of $t \in T$, the set of names T reduces to a one element set $\{\bullet\}$. Hence we obtain:

$$\mathcal{S}_e = \emptyset, \quad \mathcal{S}_\bullet = \mathcal{S} \quad \mathcal{S}_{a+b} = \mathcal{S}_a + \mathcal{S}_b$$

where $\mathcal{S}_{a_1} + \ldots + \mathcal{S}_{a_n} = \{1\} \times \mathcal{S}_{a_1} \cup \ldots \cup \{n\} \times \mathcal{S}_{a_n}$. If $x \in \mathcal{S}_a$ and $y \in \mathcal{S}_b$ we write $1.x$ and $2.y$ for their corresponding injections in $\mathcal{S}_{a+b}$. In examples, we often use more suggestive names for the injections.

The disjoint sum interpretation is extended to the composition operators and to the connectors as follows.

A.1 The Composition Operators

The *additive composition* of two nodes $N_1 \subseteq (\mathcal{I} \times \mathcal{S}_{a_1}) \times \mathcal{S}_{b_1}$ and $N_2 \subseteq (\mathcal{I} \times \mathcal{S}_{a_2}) \times \mathcal{S}_{b_2}$ yields, as in statecharts, a new node $N_1 + N_2 \subseteq (\mathcal{I} \times \mathcal{S}_{a_1+a_2}) \times \mathcal{S}_{b_1+b_2}$, such that control resides either in N_1 or in N_2:

$$\begin{aligned} N_1 + N_2 \quad &= \{(x, 1.s, 1.s') \mid (x, s, s') \in N_1\} \\ &\cup \{(x, 2.s, 2.s') \mid (x, s, s') \in N_2\} \end{aligned}$$

The *sequential composition* of two nodes $N_1 \subseteq (\mathcal{I}\times\mathcal{S}_a)\times\mathcal{S}_b$ and $N_2 \subseteq (\mathcal{I}\times\mathcal{S}_b)\times\mathcal{S}_c$ yields, a new node $N_1 ; N_2 \subseteq (\mathcal{I}\times\mathcal{S}_a)\times\mathcal{S}_c$, which is defined as expected:

$$N_1;N_2 = \{(x,s,s') \mid \exists t \in \mathcal{S}_b.(x,s,t) \in N_1 \wedge (x,t,s') \in N_2\}$$

The *additive feedback* is more tricky and it allows the construction of loops. As in programming, feedback has to be used with care in order to ensure termination. Given a relation $N \subseteq (\mathcal{I}\times\mathcal{S}_{a+c})\times\mathcal{S}_{b+c}$ we define the relation $N\uparrow^c_+ \subseteq (\mathcal{I}\times\mathcal{S}_a)\times\mathcal{S}_b$ as follows: the control is received on a and it is either given directly on b or after an arbitrary number of times in which it loops along c. Formally:

$$N\uparrow^c_+ \quad = N_{1,1} \;\cup\; N_{1,2} \,;\, N^*_{2,2} \,;\, N_{2,1}$$

where N^* is the arbitrary but finite iteration of N and $N_{i,j}$ and is defined as below:

$$N_{i,j} = \{(x,s,s') \mid (x,i.s,j.s') \in N\}$$

In this definition 1 and 2 are the injections corresponding to a and c for the input and to b and c for the output.

A.2 The Connectors

The *identity* $\mathsf{I}_a \subseteq (\mathcal{I}\times\mathcal{S}_a)\times\mathcal{S}_a$ is defined as expected:

$$\mathsf{I}_a \quad = \{(x,s,s) \mid s \in \mathcal{S}_a \wedge x \in \mathcal{I}\}$$

The *identification* ${}_k{\succ\!\bullet}_a \subseteq (\mathcal{I}\times\mathcal{S}_{k\,a})\times\mathcal{S}_a$ forgets the entry point on which it gets the control:

$${}_k{\succ\!\bullet}_a \quad = \{(x,i.s,s) \mid 0 \le i \le k \wedge s \in \mathcal{S}_a \wedge x \in \mathcal{I}\}$$

The *ramification* ${}_a{\bullet\!\prec}_k \subseteq (\mathcal{I}\times\mathcal{S}_a)\times\mathcal{S}_{k\,a}$ gives the control on any of its exit points:

$${}_a{\bullet\!\prec}_k \quad = \{(x,s,i.s) \mid 0 \le i \le k \wedge s \in \mathcal{S}_a \wedge x \in \mathcal{I}\}$$

The *transposition* ${}^b_a\!\chi \subseteq (\mathcal{I}\times\mathcal{S}_{a\,+\,b})\times\mathcal{S}_{b\,+\,a}$ commutes the entry point information:

$$\begin{aligned} {}^b_a\!\chi \quad &= \{(x,1.s,2.s) \mid s \in \mathcal{S}_a \wedge x \in \mathcal{I}\} \\ &\cup \{(x,2.s,1.s) \mid s \in \mathcal{S}_b \wedge x \in \mathcal{I}\} \end{aligned}$$

Sums clearly have *associativity isomorphisms*. The *symmetric feedback* $\oplus$ is defined as shown in Section 2.

B The Multiplicative Interpretation

B.1 The Operators

Given a type mapping D and a set of primitive type names T such that D_t is a primitive type for each $t \in T$. The $\star$ operation is interpreted as product, hence:

$$\mathcal{D}_{a\times b} \quad = \mathcal{D}_a \times \mathcal{D}_b, \qquad \mathcal{D}_{\mathsf{e}} \quad = \{()\}$$

Since the data-flow relations are defined over infinite streams, we often obtain products of the form $\mathcal{D}^{\mathbb{N}}_{a\times b}$. In this case $\mathcal{D}^{\mathbb{N}}_{a\times b} = \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_b$, because we work in a time-synchronous setting.

As in statecharts, the *parallel composition* of two components yields a new component such that control resides *in both* of the summands. Given two components $N_1 \subseteq \mathcal{D}^{\mathbb{N}}_{a_1} \times \mathcal{D}^{\mathbb{N}}_{b_1}$ and $N_2 \subseteq \mathcal{D}^{\mathbb{N}}_{a_2} \times \mathcal{D}^{\mathbb{N}}_{b_2}$ we define their product $N_1 \times N_2 \subseteq \mathcal{D}^{\mathbb{N}}_{a_1\times a_2} \times \mathcal{D}^{\mathbb{N}}_{b_1\times b_2}$ as follows:

$$N_1 \times N_2 = \{((x,u),(y,v)) \mid (x,y) \in N_1 \wedge (u,v) \in N_2\}$$

Sequential composition allows to pass the data from one component to another component. Mathematically, it is the usual sequential composition of relations. Given two relations: $N_1 \subseteq \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_b$ and $N_2 \subseteq \mathcal{D}^{\mathbb{N}}_b \times \mathcal{D}^{\mathbb{N}}_c$ we define their sequential composition $N_1 ; N_2 \subseteq \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_c$ as follows:

$$N_1 \,;\, N_2 = \{(x,y) \mid \exists z \in \mathcal{D}^{\mathbb{N}}_b.\ (x,z) \in N_1 \;\wedge\; (z,y) \in N_2\}$$

The *multiplicative feedback* allows to pass the output of a component back to its input. It is this construct which added the memory to our components by passing the state back to the input of the combinational unit. This construct also allows us to model communication between components by passing the output of one component to the input of the other one.

An essential property for the well behaveness of the feedback is that the relation introduces a one tick delay, i.e., that the relation is *strongly time guarded*. In this way, the output of the relation can be computed successively for the whole input stream. This delay was assured by introducing the register. The computation speed of the component is then given by the speed of the clock. Given a relation $N \subseteq \mathcal{D}^{\mathbb{N}}_{a\,\times\,c} \times \mathcal{D}^{\mathbb{N}}_{b\,\times\,c}$ we define the new relation $N\uparrow^c_\times \subseteq \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_b$ as the unique fix-point of the following recursive definition:

$$N\uparrow^c_\times \quad = \{(x,y) \mid \exists z.\ (y,z) \in N(x,z)\}$$

The uniqueness of the relation $N\uparrow^c_\times$ is guaranteed by the strong time guardedness of the relation N.

B.2 The Connectors

The *identity* $\mathsf{I}_a \subseteq \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_a$ is the simplest operator. It simply copies the input to the output:

$$\mathsf{I}_a \quad = \{(s,s) \mid s \in \mathcal{D}^{\mathbb{N}}_a\}$$

The *identification* $\forall^k_a \subseteq \mathcal{D}^{\mathbb{N}}_{a^k} \times \mathcal{D}^{\mathbb{N}}_a$ allows to identify k copies of elements $s \in \mathcal{D}^{\mathbb{N}}_a$:

$$\forall^k_a \quad = \{(s^k,s) \mid s \in \mathcal{D}^{\mathbb{N}}_a\}$$

The *ramification* $\mathsf{R}^a_k \subseteq \mathcal{D}^{\mathbb{N}}_a \times \mathcal{D}^{\mathbb{N}}_{a^k}$ allows to make k copies of the input s:

$$\mathsf{R}^a_k \quad = \{(s,s^k) \mid s \in \mathcal{D}^{\mathbb{N}}_a\}$$

The *transposition* ${}^a\mathsf{X}^b \subseteq \mathcal{D}^{\mathbb{N}}_{a\,\times\,b} \times \mathcal{D}^{\mathbb{N}}_{b\,\times\,a}$ allows to commute the factors:

$${}^a\mathsf{X}^b \quad = \{((x,y),(y,x)) \mid (x,y) \in \mathcal{D}^{\mathbb{N}}_{a\,\times\,b}\}$$

Products clearly contain *associativity isomorphisms*. The *symmetric feedback* $\otimes$ is defined as shown in Section 2.