Translating a Visual Description Technique to a Synchronous Language: From DiCharts to PURR*

Th. Stauner
Technische Universität München
Institut für Informatik
(Prof. Dr. M. Broy)
Arcisstraße 21, 80290 München
email: stauner@in.tum.de
http://www4.in.tum.de/~stauner

K. Schneider, M. Huhn
Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
(Prof. Dr.-Ing. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
email: {schneide,huhn}@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/hvg

Abstract. *We give a translation of the visual description technique* DiCharts *to the synchronous language* PURR*. As a result, the design of a system can start with a graphical description technique at a very high-level and can then be automatically translated to the synchronous language* PURR*. This translation allows the further synthesis of the system and in particular, its verification.*

1 Introduction

Nowadays, visual description techniques like UML [13], ROOM [16], or Statecharts [8] are widely accepted in the design of all kinds of hardware and software systems. They are used to describe the architecture of a system and the behavior and the interaction of its components in a well-structured manner. In particular, the requirement analysis benefits from visual techniques since it improves the communication with the customer [3].

In the design of hardware circuits, formal verification is already considered as a useful means for the validation of designs in early phases: Thus, essential properties of the system's behavior are formalized and proven to hold on the mathematical model of the system. In the past decade, powerful tools for automatic verification, in particular for model checking temporal logics [12, 9] have been developed. Moreover, various enhancements of these model checking procedures, as e.g. abstractions, allow to handle systems of industrial size and complexity.

To apply formal verification also for designs given in visual description techniques, a formal semantics is indispensable. Moreover, if verification tools shall be used, a semantics according to the system model of such tools is needed.

In this paper, we consider a visual description technique, namely *DiCharts* [6], developed at the Technische Universität München. DiCharts, the discrete time sublanguage of HyCharts [5], are tailored for the specification of discrete time embedded systems. They can be understood as a variant of of Statecharts, ROOMcharts or the state diagrams of UML.

We give a semantics of DiCharts in terms of PURR programs [11, 14]. PURR extends the synchronous programming language Esterel [2] by nondeterminism and abstract data types. It

*Supported with funds of the Deutsche Forschungsgemeinschaft under reference numbers Sch 623/7 and Br 887/9 within the priority program *Design and design methodology of embedded systems*.

was developed at the University of Karlsruhe as a modeling language for reactive embedded systems and is the basis for the new verification system called C@S (read 'Cats') [15]. C@S offers model checking procedures for different temporal logics like LTL, CTL or even CTL*, but also for real-time systems and logics. Additionally, it builds a link to interactive theorem provers as HOL [4]. Based on our translation, DiChart specifications of embedded systems can be subject to all verification procedures provided by C@S, in particular model checking.

The outline of the paper is as follows: In the following two sections, we briefly list the syntax and semantics of DiCharts and PURR. In Section 4, we present the translation of DiCharts to PURR, and Section 5 contains some conclusions of our work.

2 DiCharts

DiCharts [6] is a graphical description technique that is modular and based on a clear computation model. DiCharts regard a system as a network of components communicating over directed channels in a time-synchronous way. DiCharts come in two variants: *DiACharts* for the specification of the system architecture and *DiSCharts* for the specification of the behavior of the system's components. DiSCharts are very similar to the Statechart variant ROOMcharts [16].[1]

Each component is modeled by a Moore machine, consisting of a (time extended) combinational part Com^+, a register $\Delta(z)$, and the part Out that generates the outputs o (see figure 1). As usual in Moore machines, the current state $k.s$ (consisting in DiSCharts of the control state k and the data state s) contains also the current outputs, so that Out is simply a projection. The *combinational part* Com^+ instantaneously (and possibly nondeterministically) maps the current input i and the current state $k.s$ to the next state. The register $\Delta(z)$ is used to store the current state and feeds it back to Com^+ for the computation of the next state.

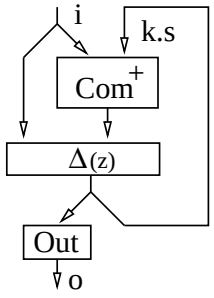


Figure 1: The computation model.

Besides the output, the data state also contains a copy of the input values received in the previous computation, i.e. the *latched inputs*. These latched inputs are updated by the register before it feeds the state back to Com^+. They allow the combinational part to remember the input values from the previous point in time. As long as the global clock does not tick, the state and the output of the machine remain stable. The arrival of a clock-tick updates the register with the next state, which was previously computed by the combinational part, and a new computation cycle begins. This concept ensures that components can only react to their inputs with a delay of one time unit.

The combinational part is specified with DiSCharts, while the diagram in figure 1, which defines the machine model, is a DiAChart. The interconnection of components is also specified with DiACharts. From a syntactic point of view, DiACharts and DiSCharts are both constructed from primitive nodes by the application of node operators and arrow operators, which we also call *connectors*, to build a *hierarchic graph*. For DiACharts, these graphs are then given a *multiplicative interpretation* while for DiSCharts the graphs are interpreted by an *additive interpretation* of the operators [6].

We briefly describe the syntax and the semantics of the operators: The syntax of the node and the arrow operators of DiCharts can be given both in a graphical and in a textual manner

[1] Note that for the specification of hybrid, i.e. mixed discrete and continuous, systems there also is an extension of DiCharts, called *HyCharts*, which is based on a dense time model and also permits the specification of continuous behavior [5].

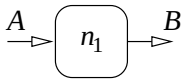
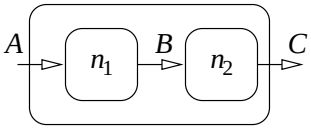
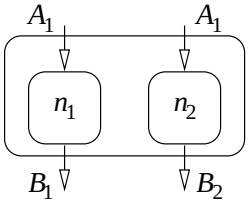
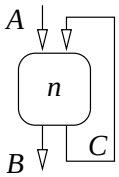
Basic Node	Sequential Composition	Visual Attachment	Feedback Loop
			
$n_1 : A \to B$	$n_1; n_2 : A \to C$	$n_1 \star n_2 : A_1 \star A_2 \to B_1 \star B_2$	$n\uparrow^C : A \to B$

Figure 2: Graphical and Textual Representation of Node Operators

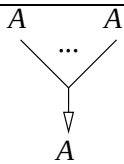
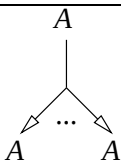
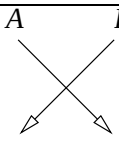
Identity	Identification	Ramification	Transposition
			
I_A	$\vee_A^m$	$\wedge_m^A$	$^A\mathsf{X}^B$

Figure 3: Graphical and Textual Representation of Connectors

and is depicted in figure 2 and figure 3, respectively. For example, the textual representation of the DiAChart given in figure 1 is as follows:

$$MooreM(Com, z) \equiv ((\wedge_2^{\mathcal{I}} \times \mathsf{I}_{m\mathcal{S}}); (\mathsf{I}_{\mathcal{I}} \times Com^+); \Delta(z); \wedge_2^{m\mathcal{S}}; (Out \times \mathsf{I}_{m\mathcal{S}}))\uparrow^{m\mathcal{S}}$$

where we write $\mathcal{I}$ for the input space and $m\mathcal{S}$ for the (control and data) state space. There are two semantics for the graphs: a multiplicative one for the DiACharts and an additive semantics for the DiSCharts.

Multiplicative Semantics. In the semantics based on the multiplicative interpretation, each node is seen as a input/output relation that nondeterministically maps an input stream to a non-empty set of output streams. Relations are used instead of functions to be able to express nondeterminism. In the following, A^B denotes the set of all functions that map elements of the set B to elements of the set A and $A \times B$ denotes the cartesian product of the sets A and B. Then, the semantics $[\![n]\!]_\times^\xi$ of a node $n : A \to B$ is a relation $[\![n]\!]_\times^\xi \subseteq A^{\mathbb{N}} \times B^{\mathbb{N}}$. We require that the relation is total in its input, i.e. there is a $(a, b) \in [\![n]\!]_\times^\xi$ for every input $a \in A^{\mathbb{N}}$. For the following, note that the set $(A \times B)^{\mathbb{N}}$ is isomorphic to the set $(A^{\mathbb{N}} \times B^{\mathbb{N}})$, so that we do not distinguish between them. The semantics of the operators is then given as follows:

- $[\![n]\!]_\times^\xi \stackrel{\text{def}}{=} \xi(n)$ for each primitive node, where $\xi(n)$ is a total relation of type $A^{\mathbb{N}} \times B^{\mathbb{N}}$, if n has input streams in $A^{\mathbb{N}}$ and output streams in $B^{\mathbb{N}}$.
- $[\![n_1; n_2]\!]_\times^\xi \stackrel{\text{def}}{=} \{(a, c) \mid \exists b \in B^{\mathbb{N}}.(a, b) \in [\![n_1]\!]_\times^\xi \wedge (b, c) \in [\![n_2]\!]_\times^\xi\}$
- $[\![n_1 \star n_2]\!]_\times^\xi \stackrel{\text{def}}{=} \{((a_1, a_2), (b_1, b_2)) \mid (a_1, b_1) \in [\![n_1]\!]_\times^\xi \wedge (a_2, b_2) \in [\![n_2]\!]_\times^\xi\}$

- $[\![n\!\uparrow^C]\!]^\xi_\times \stackrel{\text{def}}{=} \{(a,b) \mid \exists c \in C^{\mathbb{N}}.(b,c) \in [\![n]\!]^\xi_\times\ (a,c)\}$, where $n : (A \star C) \to (B \star C)$

Hence, $n_1; n_2$ corresponds to sequential composition of relations, $n_1 \star n_2$ to independent parallel execution, and $n\!\uparrow^C$ to a fixpoint calculation, i.e. recursion. Note that $[\![n_1 \star n_2]\!]^\xi_\times$ is of type $(A_1 \times A_2)^{\mathbb{N}} \times (B_1 \times B_2)^{\mathbb{N}}$, so that $\star$ is interpreted as cartesian product. The semantics of the arrow operators is as expected: We define $[\![\mathsf{I}_A]\!]^\xi_\times \stackrel{\text{def}}{=} \{(a,a) \mid a \in A^{\mathbb{N}}\}$, $[\![\vee^m_A]\!]^\xi_\times \stackrel{\text{def}}{=} \{(a^m, a) \mid a \in A^{\mathbb{N}}\}$ and $[\![\wedge^A_m]\!]^\xi_\times \stackrel{\text{def}}{=} \{(a, a^m) \mid a \in A^{\mathbb{N}}\}$, where a^m denotes the function that maps $t \in \mathbb{N}$ to the m-tuple $(a(t), \ldots, a(t))$. The transposition is also clear: $[\![{}^A\mathsf{X}^B]\!]^\xi_\times \stackrel{\text{def}}{=} \{((a,b),(b,a)) \mid (a,b) \in (A^{\mathbb{N}} \times B^{\mathbb{N}})\}$.

Additive Semantics. In the additive semantics, the graphs are viewed as control flow graphs. The intuition is that at any point of time, the control resides in *exactly one* node of the graph. Each node receives the control at one of its input arrows and forwards it on one of its output arrows. As exactly one node should have the control, the semantics of $n_1 \star n_2$ is now defined additively, as given below. The *control* is represented as $k.s$, where s is the data state in the data state space $\mathcal{S}$ and $k \in \mathbb{N}$ (the program counter) denotes the control state.

As usual, the disjoint sum $A + B$ is defined as $A + B \stackrel{\text{def}}{=} \{l.a \mid a \in A\} \cup \{r.b \mid b \in B\}$, so that elements $a \in A \cup B$ are identified by the prefix l or r as belonging to A or B, respectively. For multiple sums $A_1 + \ldots + A_m$, we use prefixes $0 < k \leq m$ to denote that a in $k.a \in A_1 + \ldots + A_m$ stems from A_k. Moreover, instead of adding m times the same set $\mathcal{S}$, we simply write $m\mathcal{S}$. Note that elements $k.s \in m\mathcal{S}$ can be encoded as tuples (k,s) for $k \in \{1, \ldots m\}$ and $s \in \mathcal{S}$.

A node $n : A \to B$ is interpreted in the additive semantics as a relation $[\![n]\!]^\xi_+ \subseteq (\mathcal{I} \times \ell\mathcal{S}) \times m\mathcal{S}$, where $(x,a,b) \in [\![n]\!]^\xi_+$ means that if the input $x \in \mathcal{I}$ is read in the control a, b can be the next control. The semantics of the node operators is then:

- $[\![n]\!]^\xi_+ \stackrel{\text{def}}{=} \xi(n)$ for each primitive node, where $\xi(n)$ is a relation $\xi(n) \subseteq (\mathcal{I} \times \ell\mathcal{S}) \times m\mathcal{S}$.
- $[\![n_1; n_2]\!]^\xi_+ \stackrel{\text{def}}{=} \left\{(x,a,c) \mid \exists b.(x,a,b) \in [\![n_1]\!]^\xi_+ \wedge (x,b,c) \in [\![n_2]\!]^\xi_+\right\}$
- $[\![n_1 \star n_2]\!]^\xi_+ \stackrel{\text{def}}{=} \{(x, l.a, l.b) \mid (x,a,b) \in [\![n_1]\!]^\xi_+\} \cup \{(x, r.a, r.b) \mid (x,a,b) \in [\![n_2]\!]^\xi_+\}$, where $l.a$ and $r.a$ denote the left and right summand of a, respectively.
- $[\![n\!\uparrow^C]\!]^\xi_+ \stackrel{\text{def}}{=} [\![n_{l,l}]\!]^\xi_+ \cup [\![n_{l,r}; n^*_{r,r}; n_{r,l}]\!]^\xi_+$ where $n : (A \star C) \to (B \star C)$, $[\![n_{i,j}]\!]^\xi_+ \stackrel{\text{def}}{=} \{(x,a,b) \mid (x, i.a, j.b) \in [\![n]\!]^\xi_+\}$ for $i,j \in \{l,r\}$ and $[\![m^*]\!]^\xi_+$ is the arbitrary, but finite iteration of $[\![m]\!]^\xi_+$. Hence, $n\!\uparrow^C$ corresponds to a while-loop: n is repeated until is passes control further on output arrow B.

Note that $A \star B$ is now additively interpreted: Instead of interpreting it as product $A \times B$, we now interpret it as sum type $A + B$.

The semantics of the connectors is then as follows: $[\![\mathsf{I}_A]\!]^\xi_+ \stackrel{\text{def}}{=} \{(x,a,a) \mid x \in \mathcal{I} \wedge a \in A\}$, $[\![\vee^m_A]\!]^\xi_+ \stackrel{\text{def}}{=} \{(x, k.a, a) \mid x \in \mathcal{I} \wedge a \in A \wedge 0 < k \leq m\}$ and $[\![\wedge^A_m]\!]^\xi_+ \stackrel{\text{def}}{=} \{(x,a,k.a) \mid x \in \mathcal{I} \wedge a \in A \wedge 0 < k \leq m\}$, where $k.a \in mA$ as defined above. The transposition is also clear: $[\![{}^A\mathsf{X}^B]\!]^\xi_+ \stackrel{\text{def}}{=} \{((x, l.a, r.a) \mid x \in \mathcal{I} \wedge a \in A\} \cup \{((x, r.b, l.b) \mid x \in \mathcal{I} \wedge b \in B\}$.

3 The Synchronous Modeling Language PURR

Synchronous languages like Esterel [2], graphical variants thereof as SyncCharts [1], and Lustre [7] are well-suited for the description of complex control tasks with discrete time. These

languages have a small and clean formal semantics which lends themselves well for the formal verification of specifications given in various formalisms. The semantics of these languages allows to translate each program to a finite-state machine so that the verification of temporal properties can be performed by means of finite-state machine traversals, which have become known as model checking algorithms for temporal logics.

Recently a variant of Esterel called PURR has been presented [10, 14]. PURR extends Esterel by nondeterminism, the ability to define abstract data types, and the use of time constraints. All these features make PURR more suited for the modeling of systems wrt. to a later verification than Esterel, while Esterel has its focus on the efficient synthesis of hardware/software.

As already a verification system for PURR is under development, it is natural to develop a translation from DiCharts to PURR so that DiCharts can also be verified. In the following, we develop a translation from DiCharts to PURR based on a translation of the multiplicative and additive interpretation of hierarchic graphs.

Due to lack of space, we can only mention some basic concepts of PURR and some basic statements that occur in the paper: the main idea is that most statements do not consume time and are therefore instantaneously executed. The time model is discrete, i.e., time is essentially modeled by the natural numbers. The only (basic) statement that consumes time is the **pause** statement, which consumes exactly one unit of time. Given statements S_1 and S_2, $S_1; S_2$ denotes sequential composition and $S_1 \parallel S_2$ denotes parallel composition. Note that in $S_1 \parallel S_2$, the threads S_1 and S_2 run synchronously to each other, i.e., they synchronize at their **pause** statements. Communication is realized by the broadcast of signals. Signals can be either present or absent and can be made present by an emission statement **emit** x. Signals may carry a value which is denoted as $?x$ and which is modified by an emission of the form **emit** $x(\tau)$ (τ is then the current value $?x$ of x). Signal values are stored unless they are changed by an emission. The signal status (being present or not) is however not stored, instead it is automatically reset at the next point of time (unless there is a further emission which overrides this).

PURR has variables which can change their values more than once in a unit of time (note that signals have for each point of time exactly one status and according to their type a value). Therefore, assignments of the form $x := x + 1$ are legal for variables, while emissions of the form **emit** $x(?x + 1)$ make no sense (causality cycles), since there is no number n that satisfies $n = n + 1$.

In our translation, we moreover use the **loop** S **end** statement that infinitely often executes the statement S. As any statement, such a loop can however be aborted: it may be surrounded by a trap statement of the form **trap** t **in** S **end** if in S the trap t is called via **exit** t.

Beneath these basic statements, we use the local signal and variable declarations and the **sustain** $x(\tau)$ statement which is an abbreviation for **loop emit** $x(\tau)$; **pause end**. For the translation of DiSCharts, we moreover use the nondeterminism of PURR that is given by the Hilbert choice operator: $@x : \alpha.\Phi$ chooses a value of type α that satisfies the property Φ. The nondeterminism of PURR is completely based on the choice expressions.

4 Translating DiCharts to PURR

4.1 DiACharts

In the translation from DiACharts to PURR presented here, we assume that every primitive node in a DiAChart is either a DiAChart as given in figure 1, i.e., a DiSChart and a register connected in the depicted way, or a PURR module. Moreover, the types of the channels on which the nodes

operate all are primitive types of PURR. We also do not consider the identification connectors $\vee_A^m$, which introduce synchronization between channels to DiACharts.[2]

The connectors, I_A, $\wedge_m^A$ and ${}^A\mathsf{X}^B$ are necessary for DiACharts, because DiACharts rely on point-to-point communication without channel names. As PURR uses the signal names to address inputs and outputs and allows broadcast communication, the connectors are obsolete: We handle the connectors by a suitable naming of signals. The translation from a DiAChart to PURR therefore only involves the node operators and is defined recursively on the structure of the DiAChart as follows:

Sequential Composition: Assume $n_1 : A_1 \star \ldots \star A_p \to B_1 \star \ldots \star B_r$ and $n_2 : B_1 \star \ldots \star B_r \to C_1 \star \ldots \star C_q$ are given and we have already computed PURR statements $D(n_1)$ with the inputs $\mathcal{I}_1 = \{i_1,\ldots, i_p\}$ and the outputs $\mathcal{O}_1 = \{g_1,\ldots,g_r\}$, and also $D(n_2)$ with the inputs $\mathcal{I}_2 = \{h_1,\ldots,h_r\}$, and the outputs $\mathcal{O}_2 = \{o_1,\ldots,o_q\}$. Then, we define $D(n_1; n_2)$ with inputs $\mathcal{I}_1$ and outputs $\mathcal{O}_2$ as follows:

$$D(n_1;n_2) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{signal } x_1 : B_1, \ldots, x_r : B_r \textbf{ in} \\ \quad D(n_1)[g_1 \leftarrow x_1, \ldots, g_r \leftarrow x_r] \parallel D(n_2)[h_1 \leftarrow x_1, \ldots, h_r \leftarrow x_r] \\ \textbf{end signal} \end{array} \right)$$

where $x_1, \ldots, x_r$ are fresh local signal names and $[y \leftarrow x]$ denotes substitution of signal names.

Visual Attachment: Assume $n_1 : A_1 \to B_1$ and $n_2 : A_2 \to B_2$ are given and we have already computed PURR statements $D(n_i)$ with inputs $\mathcal{I}_i$ and outputs $\mathcal{O}_i$ for $i \in \{1, 2\}$. Then we define $D(n_1 \star n_2) \stackrel{\text{def}}{=} D(n_1) \parallel D(n_2)$ with inputs $\mathcal{I}_1 \cup \mathcal{I}_2$ and outputs $\mathcal{O}_1 \cup \mathcal{O}_2$.

Feedback Loops: Assume $n : A \star C \to B \star C$ is given and we have already computed $D(n)$ with inputs $\mathcal{I} = \{a, c\}$ and outputs $\mathcal{O} = \{b, c'\}$. Then, we define $D\left(n\uparrow^C\right) \stackrel{\text{def}}{=}$ **signal** x : C **in** $D(n)[c \leftarrow x, c' \leftarrow x]$ **end signal** with input $\{a\}$ and output $\{b\}$.

Primitive Nodes: Every primitive node must either be given by a DiAChart according to fig. 1 or by a PURR module. In the latter case, we set $D(n) = n$. If n is a DiSChart, we define $D(n) \stackrel{\text{def}}{=} D\left(\left(\left(\wedge_2^{\mathcal{I}} \times \mathsf{I}_{m\mathcal{S}}\right); (\mathsf{I}_{\mathcal{I}} \times Com^+); \Delta(z_{Init}); \wedge_2^{m\mathcal{S}}; (Out \times \mathsf{I}_{m\mathcal{S}})\right) \uparrow^{m\mathcal{S}}\right)$ (cf. figure 1), where $z_{Init} = k_{Init}.s_{Init} \in m\mathcal{S}$ stands for the initial control with which the computation starts. Com^+ refers to the (time extended) combinational part of component n. It is defined by the DiSChart that specifies n. In Section 4.2 we will see how $\Delta(z)$, Out and Com^+, the components of our Moore machine model, are translated to PURR.

In the above translations, we always assume that the programs are well-typed, i.e. we choose local signal variables x_i of the corresponding type B_i in PURR.

4.2 Connecting DiACharts and DiSCharts

In this section, we consider the translation of the components of our Moore machine model. Such machines define the behavior of all those primitive node in DiACharts which are given by

[2] Identification connectors require to receive m equal input streams and identify them. It is unclear how systems containing identification connectors are realized operationally, because the connector imposes a very strong requirement on the components sending the input streams to it. In fact, this connector has not been used in any application of DiACharts so far.

DiSCharts (and not directly by PURR code). We provide PURR code for the register $\Delta(z)$, the output component Out, and the time extension of the combinational part Com^+. The PURR code for the hierarchic graph Com which is defined by the DiSChart is given in the next section. The composition of these ingredients is covered by the translation for the primitive DiAChart nodes given in the previous section.

```
module REG
input i : I, s : S, k : ℕ;
output s' : S, k' : ℕ;
   emit s'(s_Init); emit k'(k_Init);
   loop
      emit s'((fst(?s), ?i)) after 1;
      emit k'(?k) after 1;
      pause;
   end loop
end module
```

Figure 4: Code for $\Delta(z_{Init})$.

The *register* $\Delta(z_{Init})$ stores the current input $i \in \mathcal{I}$ and the data it receives from the combinational part for one clock tick, updates the latched inputs, and then passes control and data state on its outputs where the modules Out and Com^+ can take them. The PURR code $D(\Delta(z_{Init}))$ for the register with the initial control $z_{Init} = k_{Init}.s_{Init}$ is given in figure 4. The channels k and k' are used to transmit the control state of the DiSChart, the channels s and s' transmit its data state. As mentioned previously, the latched inputs are a part of the data state $\mathcal{S}$. Therefore, we can write $\mathcal{S}$ as the set product of the state space of local and output variables $\mathcal{L}$ and the latched input variables $\mathcal{I}$, i.e. $\mathcal{S} = \mathcal{L} \times \mathcal{I}$. The projection $fst(s)$ for $s \in \mathcal{S}$ thus yields the data state without the latched inputs. This is used in the **emit** statement for channel s' to perform the update of the latched inputs. Statements **emit** x **after** 1 express that signal x is emitted in the next clock cycle.

```
module Out
input s : S, k : ℕ;
output o : O;
   sustain o(proj(?s))
end module

module Com+
input i : I, s : S, k : ℕ;
output s' : S, k' : ℕ;
   var sv : S, kv : ℕ in
      loop
         sv :=?s; kv :=?k;
         C(n);
         emit s'(sv); emit k'(kv);
         pause;
      end loop
   end var
end module
```

Figure 5: Code for Out and Com^+.

Out performs a projection $proj$. It receives the current control state k and data state s and computes the current output o as a projection of s. The PURR code $D(Out)$ for it is given in figure 5. $proj(s)$ projects the data state stored in s to the output variables.

The time extended combinational part Com^+ receives at each point of time the current input $i \in \mathcal{I}$ and the DiSChart's state (i.e. k and s) from the register. It immediately computes the next state s' and k' by the PURR statement $C(n)$ and forwards it as inputs to $\Delta(z_{Init})$. $C(n)$ consumes no time, i.e., the computation is finished within a clock cycle. The code for $D(Com^+)$ is given in figure 5, where $C(n)$ is the translation of the additive hierarchic graph n into PURR. Note that $C(n)$ only reads $?i$. Variables kv and sv are read and possibly modified.

4.3 DiSCharts

It remains to define the translation $C(n)$ of a DiSChart to PURR. At the user level, DiSCharts look very similar to ROOMcharts or other Statechart variants and support preemption, state entry and exit actions, and similar features of extended state machines. All graphic elements of DiSCharts are macros built from the operators on nodes and arrows (Section 2). These macros

are defined in [6]. When the macros are expanded we end up with a hierarchic graph only consisting of the operators of Section 2 and of primitive nodes which refer to *actions*. Actions consist of a *guard* and a *body*. If the guard is true, the body is executed and possibly changes the values of controlled variables in dependence on the current inputs. Only if the guard is true, the action passes control further. In the context of our PURR translation we demand that all action guards are given as boolean PURR expressions and that all action bodies are sequences of assignments, also given as PURR statements. Note that these expressions and assignments may not contain PURR statements that refer to signals, as the communication concept of DiSCharts is implemented by the machine model.

According to the semantics of DiSCharts, computations that end, because an action is selected which does not pass control further, do not appear in the set of possible executions. Therefore, it would not be correct to translate them to PURR executions which are active for some time and then pause forever. To solve this problem it is important to note that the definition of the macros ensures that actions are nondeterministically selected by the ramification connectors. Without loss of generality we can assume that each outgoing arrow of a ramification is followed by an action. (If necessary dummy actions with guard true and the identity as body can be introduced.) This is used in the following translation of DiSCharts.

Relying on the macros in [6] it now suffices to define the translation of the operators on nodes and arrows from Section 2 and of primitive nodes to PURR. As ramification is always followed by actions, we do not consider $\wedge$ alone, but translate the whole construct $\wedge_m; (+_{i=1}^{m} a_i)$, which means that each outgoing arrow i of $\wedge_m$ is followed by an action a_i.

Let n be a hierarchic graph. Its translation to PURR is defined as follows:

Sequential Composition: $C\,(n_1; n_2) \stackrel{\text{def}}{=} C\,(n_1)\,; C\,(n_2)$.

Visual Attachment: For a visual attachment $n_1 \star n_2$, where n_1 has g input arrows and h output arrows, we define $C\,(n_1 \star n_2)$ as given below, where kv is the variable defined in the translation of Com^+ that encodes the current control state.

$$C\,(n_1 \star n_2) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{if } kv \leq g \textbf{ then } C\,(n_1) \\ \textbf{else} \\ \quad kv := kv - g; \\ \quad C\,(n_2)\,; \\ \quad kv := kv + h \\ \textbf{end if} \end{array} \right)$$

To understand the manipulation of kv in detail, the reader is referred to [5]. The principle is as follows: The name spaces for the input and output arrows of a node in a hierarchic graph are the natural numbers. The names always start with 1. When two hierarchic nodes n_1 and n_2 are composed, the name space of the second one must be shifted in order to still be able to refer to its arrows. This way input arrow i of n_2 gets number $g + i$ in the composed node $n_1 \star n_2$, where g is the number of input arrows of n_1.[3]

Feedback: Given n with $g + i$ input arrows and $h + i$ output arrows, the translation $C\left(n\uparrow^{i\mathcal{S}}\right)$ is given below, where $newT$ is a new identifier. Note that termination of this loop is not guaranteed. A non-terminating loop results if there is an infinite sequences of transient

[3]Note that the reduction of DiSCharts to hierarchic graphs by the macros leads to a mapping of states in the DiSChart to a special kind of arrows in the graph, in [6] these arrows are called *wait-entry* and *wait-exit ports*. Variable kv actually stores arrow numbers.

states in the DiSChart. It is in the responsibility of the user to ensure that the design has no such errors.

$$C\left(n\uparrow^{i}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{trap } newT \textbf{ in} \\ \quad \textbf{loop} \\ \quad\quad C\left(n\right); \\ \quad\quad \textbf{if } kv > h \textbf{ then } kv := kv - h + l \\ \quad\quad \textbf{else exit } newT \\ \quad\quad \textbf{end if} \\ \quad \textbf{end loop} \\ \textbf{end trap} \end{array}\right)$$

Identity: $C\left(\mathsf{I}\right) \stackrel{\text{def}}{=}$ **nothing**

Identification: $C\left(\vee^{m}\right) \stackrel{\text{def}}{=} kv := 1$

(i,j)-ary Transposition: $C\left({}^{i}\mathsf{X}^{j}\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{if } kv \leq i \textbf{ then } kv := kv + j \\ \textbf{else } kv := kv - i \\ \textbf{end if} \end{array}\right)$

Ramification: $C\left(\wedge_{m}; (+_{i=1}^{m} a_i)\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} kv := @i : \mathbb{N}.\, 1 \leq i \leq m \wedge \textit{guard}(a_i); \\ \textbf{if } kv = 1 \textbf{ then } \textit{body}(a_1) \\ \vdots \\ \textbf{elseif } kv = m \textbf{ then } \textit{body}(a_m) \\ \textbf{end if} \end{array}\right)$

where $\textit{guard}(a_i)$ is the PURR code of the guard of action a_i and $\textit{body}(a_i)$ is the code for its body. The Hilbert operator $@x : \alpha.\Phi(x)$ is a primitive of PURR (see Section 3). We use it here to nondeterministically select an enabled action.

Primitive Nodes: For any primitive node, which is an action a as guaranteed by the macros that generate the hierarchic graphs, we define $C\left(a\right)$ as:

$$C\left(a\right) \stackrel{\text{def}}{=} \left(\begin{array}{l} \textbf{if } \textit{guard}(a) \textbf{ then } \textit{body}(a) \\ \textbf{else halt} \\ \textbf{end if} \end{array}\right)$$

Thus, the statement does not terminate, if the action guard is false. Typically, this can occur if an action is followed by a second action which does not have guard true. We require that the user ensures that the second action always is enabled if the first one was taken. An alternative to this strategy is to disallow that the second action has a guard different from true. This e.g. is done in many tools for Statecharts.

5 Conclusion

We gave a modular translation of DiCharts to the synchronous language PURR. By handling DiACharts describing the system's architecture and DiSCharts for the behavioral part separately, the structure of the specification is preserved in the PURR translation. This structure preserving

translation will be very helpful in verification: If errors are detected, they have to be traced in the model, and the original specification, i.e., in the DiCharts.

In future we will compare the PURR semantics developed here to the original semantics of DiCharts [6]. Additionally, we aim to validate our approach by verifying a DiChart specification.

References

[1] Ch. Andre. Representation and analysis of reactive behaviors: A synchronous approach. research report tr96-28, University of Nice, Sophia Antipolis, 1996.

[2] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[3] M. Broy and T. Stauner. Requirements Engineering für eingebettete Systeme. *Informationstechnik und technische Informatik*, 2:7–11, 1999.

[4] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[5] R. Grosu, T. Stauner, and M. Broy. A modular visual model for hybrid systems. In *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[6] R. Grosu, Gh. Stefănescu, and M. Broy. Visual formalisms revisited. In *Proc. International Conference on Application of Concurrency to System Design (CSD'98)*, 1998.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, sep 1991.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, pages 231–274, 1987.

[9] G. J. Holzmann and D. Peled. The state of SPIN. In Rajeev Alur and Thomas A. Henzinger, editors, *Conference on Computer Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 385–389, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[10] T. Kropf, J. Ruf, K. Schneider, and M. Wild. A synchronous language for modeling and verifying real time and embedded systems. In *GI/ITG/GME Workshop: Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen*, pages 11–20. HNI-Verlagsschriften, ISBN 3-931466-35-3, 1998.

[11] T. Kropf, J. Ruf, K. Schneider, and M. Wild. The synchronous system description language PURR. In *Open Project Workshop on System Design Automation (SDA98)*, Dresden, Germany, 1998.

[12] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[13] Rational Software Corporation. Unified modeling language, version 1.1. `http://www.rational.com/uml/documentation.html`, 1998.

[14] D. Schmid, K. Schneider, M. Huhn, G. Logothetis, and V. Sabelfeld. Formale Verifikation eingebetteter Systeme. *Informationstechnik und Technische Informatik (it+ti)*, 2:12–16, March 1999.

[15] K. Schneider and T. Kropf. The C@S system: Combining proof strategies for system verification. In T. Kropf, editor, *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 248–329. Springer Verlag, state of the art report edition, August 1997.

[16] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons Ltd, Chichester, 1994.