Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 180 (2007) 107–121
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Visual Specification of Systems with Heterogeneous Coordination Models[1]

David Šafránek[2]

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic

Abstract

In this paper a prototype of a visual specification language called Visual Coordination Diagrams (VCD) for high-level design of concurrent systems with heterogeneous coordination models is presented. The key property of VCD is the separation of behavioral aspects from coordination aspects. We also highlight the heterogeneity of VCD which has two levels. At first, it allows different coordination models to be mixed in a particular specification. Secondly, different formalisms can be incorporated to VCD for specification of behavioral aspects. This paper contains an overview of the language followed with its formal definition. An example of using the language is also given.

Keywords: Visual Coordination Diagrams, visual specification, concurrent systems, exogenous coordination model

1 Introduction

Visual formalisms are very popular in specification and design of software and hardware systems. The most wide-spread of them are included in the Unified Modeling Language (UML). Examples of others are Statecharts [6], Message Sequence Charts [7], Petri Nets [12],... Visual languages have the advantage of being simple to use for system designers. A difficult problem is to find a compromise between the richness of syntactic constructs and the comprehensible formal semantics of a visual language. Ideally, a visual language should be capable of suitable handling both coordination and behavioral aspects of concurrent systems.

Importance of universal modeling languages such as UML is very significant in the domain of software engineering. Desired properties of such an universal design language, which would be suitable for concurrent systems, are heterogeneity, hierarchy and component-based structure. Nowadays, typical computerised-systems are

[1] This work has been supported by the Grant Agency of Czech Republic grant No. 201/03/0509 and FRVS grant No. 504/2004.

[2] Email: xsafran1@fi.muni.cz

doi:10.1016/j.entcs.2006.10.048

composed of hardware and software components with different models of computation, some of them may be transformational, while the others can be interactive or reactive. We call such complex systems *heterogeneous*.

In this paper we present Visual Coordination Diagrams (VCD) – a visual *design* formalism for specification of component-based concurrent systems, based on the idea of GCCS [5] and its extensions [15]. Note that this is not a programming language in the sense of Linda, Manifold, or other coordination languages. VCD is aimed to be a design formalism.

VCD employs a coordination model in which coordination aspects are semantically separated from the behavioral aspects. This is called *exogenous* model [4]. VCD can be also viewed as static architecture diagrams specifying connections among components. The key property of VCD is its two-level heterogeneity. The first level of this heterogeneity is based on the possibility of combination of various coordination models (both synchronous and asynchronous) in a particular system specification. The second level of heterogeneity is the variability of specification of behavioral aspects. This can be done in various notations which have to be, in some well-defined sense, compatible with the coordination models supported by the language.

1.1 Background and Related Work

There is a group of visual languages for concurrent systems in which the classical state transition diagrams have been extended to fulfil the needs of design of complex systems. Combining the concept of geometric inclusion with the concept of hypergraphs, the hierarchy of states has been added, leading to Harel's Statecharts [6]. The complexity of the syntactic richness of Statecharts has shown that reaching a compositional formal semantics for such a powerful language is tedious. Various sub-dialects of Statecharts have been defined to achieve required properties of their formal semantics [9]. The concept of Statecharts was also incorporated in UML [14].

Another group of visual languages is based on the concept of message flow graphs. They are employed to visually describe partial message passing interaction among concurrent processes. The high level message flow diagrams called Message Sequence Charts are based on this concept [8]. This formalism does not support hierarchical design. For its simple nature, it is widely used in telecommunication industry and it is also a part of UML.

Graphical calculus of communicating systems (GCCS) [5] and its synchronous extension SGCCS [15] adopt the process algebraic approach as the formal underlying semantic model. These languages have component-based hierarchical architecture. Because of too tight relation to the underlying process algebraic semantic model, the heterogeneity of both coordination and behavioral layers is limited.

There is another architecture language, which is, similarly to VCD, based on the idea of GCCS. It is called Architectural Interaction Diagrams (AID) [13]. VCD and AID both achieve some level of heterogeneity by avoiding the tight relation with the CCS process algebra. One of the significant differences between these two formalisms is in the underlying semantic model. AID is aimed to be used for specification of interactive systems while in VCD the interactive aspects can be additionally mixed with reactivity. At the behavioral layer, VCD supports more expressive formalisms than AID, and thus allows more heterogeneity at this level. On the other hand, AID allows more non-deterministic modeling at the coordination layer than VCD.

In the community of coordination languages, there is a large group of languages which have properties of architecture languages. The most significant languages from this domain are Manifold [10] and ToolBus [2]. These languages support control-driven exogenous coordination. In contrast to VCD, these languages are complex programming languages. VCD is aimed to be a simple visual formalism for higher-level design of concurrent systems. Moreover, unlike in Manifold, there is currently no support for dynamic changes of component connections in VCD. Static coordinators are represented in VCD as buses. What is similar to both Manifold and VCD is the concept of ports. Also the Manifold coordinator hierarchy has in some sense its counterpart in the VCD coordination layer hierarchy.

The main reason for developing VCD is our believe in importance of building a framework for coordination of various kinds of Statecharts and other visual formalisms for specification of component behavior. We would like to establish a simple syntactic visual notation with suitable underlying formal semantics. The chosen semantic model is based on composition of local transition systems, which represent particular components, resulting in a global transition system.

2 Overview of VCD

VCD is aimed to be a formal language for the specification of communication relationships in component-based concurrent systems. A simple system specified in VCD is depicted in Fig. 1. Basic elements of the language are component *interfaces*. A particular interface contains input and output *ports* which serve as gates for an encapsulated component with the effect of offering its services to the environment. Interfaces are grouped into *networks* and can be interconnected with *buses*.

Buses are used for specification of various types of coordination mechanisms. Different types of buses can be mixed together in a particular network. Consequently, systems with heterogeneous coordination mechanisms can be effectively specified using a single uniform formalism. From the semantical point of view, buses are based on the similar state transition paradigm as computational components. In contrast to computational components, buses cannot be refined with a network. Note that the fact of having buses flat does not limit the possibility of building the meta-coordinators known from Manifold. Buses are aimed to be primitives which encapsulate the behavior of various coordination media. By mixing buses together with some simple components, we can make networks, which then represent more complicated coordination media. But, unlikely as in Manifold, these "meta-coordinators" have to be taken as ordinary components.

The key concept of VCD is in its hierarchical network structure, which makes the *coordination layer*. This is achieved by the possibility of taking networks as

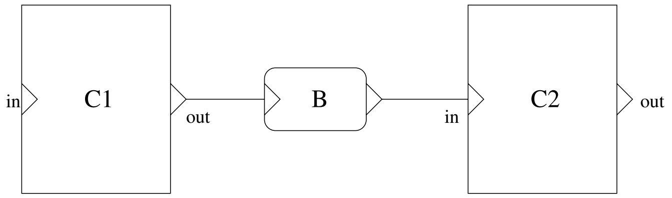

Fig. 1. A network of components *C1* and *C2* interconnected by the bus *B*

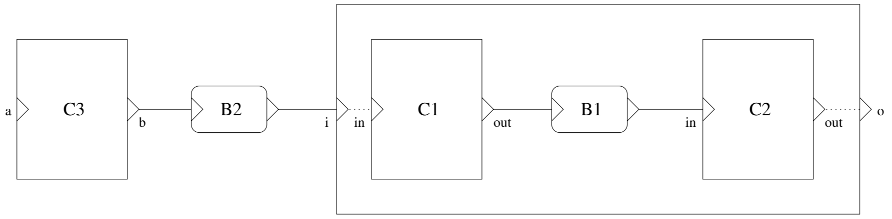

Fig. 2. Network hierarchy

components of other networks (higher level networks). An example of the network nesting is given in Fig. 2.

Semantics of VCD is based on state transition model. By traversing the network hierarchy, it relies on a formal mechanism of combining component state transition models into one resulting state transition model of the top-most network. This is done with respect to the communication relationships specified by busses in networks. Semantics of a particular bus type represents behavior of a specific communication media.

One can imagine a particular VCD network as a graph in which subsystem interfaces create vertexes and busses with links create edges. Any such graph can be partitioned into strongly connected components. The VCD semantics combines subsystem transitions of these strongly connected components synchronously (product) or asynchronously (interleaving) into resulting network transition. The asynchronous or synchronous coordination model results from the semantics of included buses. The semantic relationship among transitions combined from different strongly connected network components is always asynchronous.

At the bottom-most level, behavior of system components can be specified in any VCD-compatible formalism. This is called *behavioral layer* of VCD. The semantic model behind the behavioral layer is an input/output labeled transition system (LTS) with sets of input and output actions taken as labels. This allows any language with semantics derived in the domain of LTS to be used for behavioral specification of system components. This property makes VCD heterogeneous also at the behavioral layer. Heterogeneity of the behavioral layer is achieved with respect to the set of semantically compatible, but notationally different languages, which are used for behavioral description. As examples of supported languages we can mention variants of Statecharts or Petri-Nets.

An example of a more complex network is in Fig. 3. It is an abstract simplification of a design specification of a hardware accelerated network router [1]. This

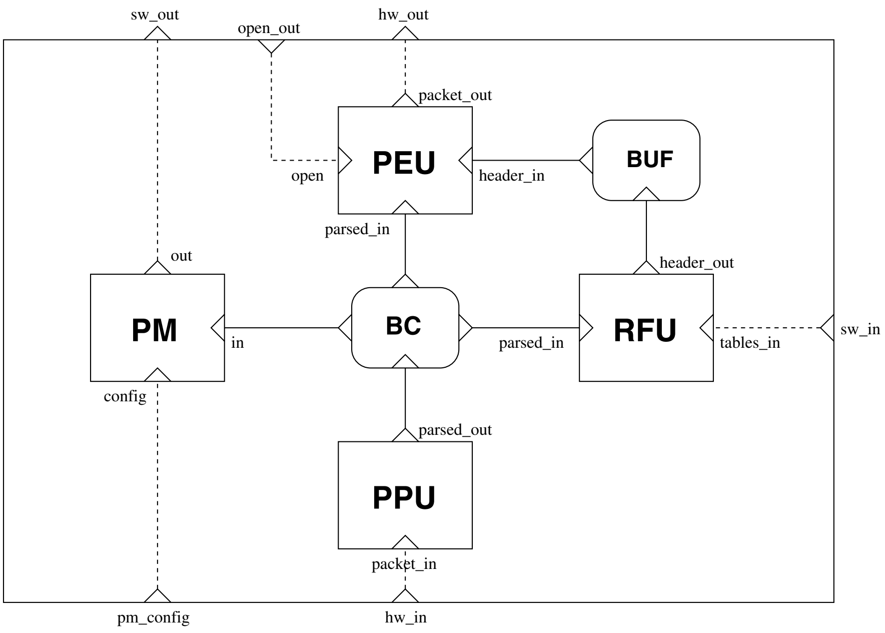

Fig. 3. Network modeling a routing and firewalling system

example is interesting to be presented here because it highlights some features of VCD. The system is being designed for programmable hardware platform, for that reason it can be taken as a software architecture. The network in the figure describes the basic components of the router system and connections among them. The role of the whole system is to take input network packets on its hardware input (port *hw_in*), reading the routing and firewalling tables on its software input (port *sw_in*) and produce the packets modified according to these tables on the *hw_out* output port. Additionally, the system manages and makes some statistics about packets which flow through it. To achieve this functionality, it takes some software configuration on its *pm_config* port and produces the relevant software output on the *sw_out* port.

Refining the system into components, there are four different units interconnected by two buses. The unit *PPU* (Packet Parsing Unit) identifies the information important for routing and firewalling from input packets. The extracted information is synchronously sent to three other units. Note that the synchronous communication is modeled by a broadcast bus *BC*. In the *PEU* (Packet Editing Unit) unit the sent information is stored in some kind of memory for later use. In the *PM* (Packet Manager) unit it is used for computing some statistics. Finally, in the *RFU* (Routing and Firewalling Unit) it is compared with information in the tables, modified and sent asynchronously to *PEU* unit. Here this asynchronous communication is modeled using an asynchronous message passing bus *BUF*.

In Fig. 4 there are Statecharts which model the behavior of individual units. For simplicity we have slightly abstracted from their complex state spaces. We also abstract from data flow and model the binary signal flow (presence or absence of events [3]) only. In this example, statecharts of the *PM* and *RFU* unit have

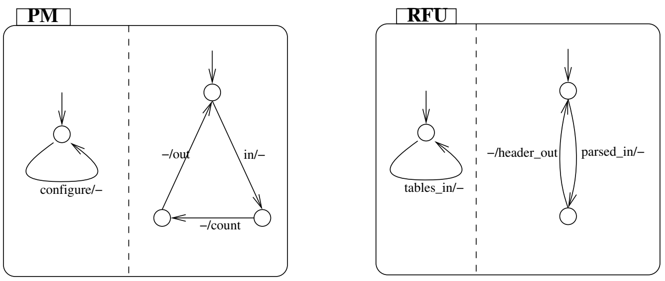

Fig. 4. Statecharts modeling behavior of components

the form of concurrent compound states known as AND-states in the Statechart terminology. More specifically, in the case of *RFU* this means that the unit can accept the input *tables_in* in whatever state the right substatechart is. So from the initial state of that statechart, there are three transitions possible – $tables\_in/-$, $parsed\_in/-$ or $tables\_in, parsed\_in/-$. One of them (chosen w.r.t. presence of a *tables_in* input event) can take its part in synchronisation with *PEU* and *PM* units by the *BC* bus. Note that all of the events in the *RFU* statechart are related by their name with relevant input and output ports of *RFU* interface. This is not the case of the *PM* statechart which could perform an internal input event *configure*. The interface of the *PM* unit abstracts from this event.

3 Formalising VCD

In this part we give formal syntax for the visual constructs of VCD, which have been introduced in the previous section. Later on in this section, the structural operational semantics of VCD is defined.

3.1 Syntax

VCD networks are formally represented as VCD *terms*. Before we will define them, we build some basic notation.

3.1.1 Ports and Interfaces

The most basic element of the coordination layer is a *port*. We fix $\mathcal{W}$ a countable set of *write ports* and $\mathcal{R}$ a countable set of *read ports*. We require $\mathcal{W} \cap \mathcal{R} = \emptyset$. *Interface* I is then defined as any non-empty set of ports $I \subseteq \mathcal{W} \cup \mathcal{R}$. We mark $I^R = \{r \in I \,|\, r \in \mathcal{R}\}$ the *read-interface*, $I^W = \{w \in I \,|\, w \in \mathcal{W}\}$ the *write-interface*, respectively.

3.1.2 Buses and Bus Classes

The key construct of the coordination layer is a *bus*. As it has been mentioned in the previous section, buses represent coordination mechanisms. They are used for modeling various types of coordination models, such as bi-party handshake message passing, synchronised broadcast, or asynchronous types of component coordination. Different buses can be mixed in the specification of a particular network, which makes the coordination layer heterogeneous. Particular types of coordination mechanisms are represented as *bus classes*, which are formally defined as input/output labeled state transition systems (I/O LTS).

Definition 3.1 *Bus class* $\mathcal{B}$ is a tuple $\langle Q, T, q_0\rangle$ where

- Q is a finite set of states,
- $q_0 \in Q$ an initial state,
- $T \subseteq Q \times 2^{\mathcal{W}} \times 2^{\mathcal{R}} \times Q$ a transition relation.

Any bus class can be instantiated as a particular bus and used for specification of coordination of components in a network. The bus interface is always given by the set of links which connect the bus to the ports of surrounding components. Formal definition of a bus, given by its interface and its class, is the following.

Definition 3.2 *Bus* B of a bus class $\mathcal{B}$ is a tuple $B = \langle I, \mathcal{B}\rangle$, where I is an interface and $\mathcal{B}$ a bus class.

The interface of the bus B will be denoted as $I(B)$, $I(B) \subseteq \mathcal{W} \cup \mathcal{R}$.

3.1.3 Gates, Networks and Leaves

Now we define terms which formally represent VCD network diagrams. Note that in the network depicted in Fig. 2 there are dashed lines which connect ports of subsystem interfaces to ports of the surrounding network interface. Later on in this subsection, these connections will be formalised as the notion of a *gate*.

Definition 3.3 A *VCD term* is:

(i) *VCD leave* – behavioral model specified in any LTS-compatible language

(ii) *VCD network* $N = \langle \bar{C}, \bar{M}, L\rangle$, where

  (a) $\bar{C} = \langle C_1, \ldots, C_n\rangle$ – vector of components

  (b) $\forall i : C_i = \langle S_i, I_i, G_i\rangle$
  - S_i …VCD term
  - I_i …$I_i \subseteq \mathcal{W} \cup \mathcal{R}$ interface
  - G_i …gate (see definition below)

  (c) $\bar{M} = \langle M_1, \ldots, M_k\rangle$ – vector of busses

  (d) $\forall j : M_j = \langle I_j, \mathcal{B}_j\rangle$
  - I_j …interface of a bus M_j
  - $\mathcal{B}_j$ …class of a bus M_j

  (e) $L \subseteq (\{1, \ldots, n\} \times (\mathcal{W} \cup \mathcal{R})) \times (\{1, \ldots, k\} \times (\mathcal{W} \cup \mathcal{R}))$ a set of *links* satisfying: if $\langle\langle i, p_1\rangle, \langle j, p_2\rangle\rangle \in L$ then:
  $p_1 \in \mathcal{W} \Leftrightarrow p_2 \in \mathcal{R}$
  $p_1 \in I_i$
  $p_2 \in I^W(M_j) \cup I^R(M_j)$
  $\langle\langle l, p_1'\rangle, \langle j, p_2\rangle\rangle \in L \Leftrightarrow l = i \wedge p_1' = p_1$
  $\langle\langle i, p_1\rangle, \langle l, p_2'\rangle\rangle \in L \Leftrightarrow l = j \wedge p_2' = p_2$

The set of all VCD terms will be denoted by $\mathcal{S}$.

We define a function ϵ_R (ϵ_W) which for any VCD network returns a set of all its read (write) ports which have no connection with any bus. We call such ports *free ports*. To overcome ambiguity of port names in the context of a network, we mark each port in the network with a natural number. This number is given by the index of the component (from the component vector $\bar{C}$) to which the relevant port belongs.

Definition 3.4 Let $N = \langle \bar{C}, \bar{M}, L \rangle$ be a network.

- $\epsilon_W(N) = \{\langle i, w \rangle \mid w \in I_i^W \wedge \forall j, w' : \langle \langle i, w \rangle, \langle j, w' \rangle \rangle \notin L\}$
- $\epsilon_R(N) = \{\langle i, r \rangle \mid r \in I_i^R \wedge \forall j, r' : \langle \langle i, r \rangle, \langle j, r' \rangle \rangle \notin L\}$

For the interface of N we will use the notation $I(N) = \epsilon_W(N) \cup \epsilon_R(N)$.

A *gate* is defined as a partial bijection relating ports of a network to free ports of a particular component in the network. The gate formalises the port mappings depicted in VCD as dashed lines. For simplicity reasons we define gate also for VCD leaves. In this case it is an identity which maps ports of a component to eponymous events of a particular VCD leave. Note that the leave gate restricts the allowed set of leaves which can be embedded in a particular component interface to only those leaves which have the relevant events in labels of transitions. Similarly, the definition of the network gate restricts the possible candidates for embedding of networks into other networks. Only a network with enough free ports can be embedded.

Definition 3.5 Let $I \subseteq \mathcal{W} \cup \mathcal{R}$ be an interface.

(i) Let S be a VCD leave encapsulated in the interface I. Let $ports(S) \subseteq \mathcal{W} \cup \mathcal{R}$ be a set of all ports of S. We define a *gate* of the leave S as the identity function $G : I \rightarrow ports(S)$, $\forall x \in I. G(x) = x$.

(ii) Let $S = \langle \langle \langle S_1, I_1, G_1 \rangle, ..., \langle S_n, I_n, G_n \rangle \rangle, \langle M_1, ..., M_k \rangle, L \rangle$ be a VCD network embedded in interface I. We define a *gate* of the network S as the partial function $G : I \rightarrow I(S)$ satisfying:
- $\forall w \in I^W. G(w) = \langle i, w' \rangle \wedge \langle i, w' \rangle \in \epsilon_W(S)$
- $\forall r \in I^R. G(r) = \langle i, r' \rangle \wedge \langle i, r' \rangle \in \epsilon_R(S)$

Before we start to define the semantics of VCD, we establish some notation. Let N be a network containing just $n > 0$ components. Further let I_i be the interface of the ith component of N and $\Gamma \subseteq I_i$ some set of its ports. We will denote $\langle i, \Gamma \rangle = \{\langle i, w \rangle \mid w \in \Gamma\}$ the set of ports indexed by the ith component in the network N. Note that if $\Gamma = \emptyset$ then also $\langle i, \Gamma \rangle = \emptyset$.

3.2 Semantics

Here we give the VCD terms a precise operational semantics. As a semantic domain we use a class $\mathcal{L}$ of input/output labeled transition systems (I/O LTS) with sets of input and output events in transition labels. Formally the semantics is defined as a mapping ψ of the type $\psi : \mathcal{S} \rightarrow \mathcal{L}$ which assigns an I/O LTS to each VCD term.

First of all, we formally define I/O LTS, which makes the semantic domain for both the behavioral and coordination layer.

Definition 3.6 Let S be a VCD leave. An *I/O LTS* is a tuple $\langle Q, T, q_0 \rangle$ where

- Q is a finite set of states,
- $q_0 \in Q$ an initial state,
- $T \subseteq Q \times 2^{\mathcal{R}} \times 2^{\mathcal{W}} \times Q$ a transition relation.

At the behavioral layer, the state transition semantics captures the dynamics of system components. Note that VCD does not include any predefined syntactic constructs for behavioral layer, but relies on other formalisms which respect the supported state transition semantics of VCD leaves.

Dealing with the coordination layer, we would like to define semantics of VCD networks. In principle, the semantics of a VCD network is defined as a global I/O LTS which combines transitions of local I/O LTSs representing the semantics of network components. This combination is done with respect to the coordination model encoded in buses used in the network.

A network term contains a vector of components and a vector of buses. The semantics of the network term respects this structure. States and transitions of the I/O LTS which represents the network term are constructed by composition of states and transitions of component I/O LTSs. To construct the resulting I/O LTS formally, we define the notion of a *network configuration.*

Definition 3.7 Let $N = \langle\langle C_1, ..., C_n\rangle, \langle M_1, ..., M_k\rangle, L\rangle$ be a network. We define its *configurations* $\langle \bar{s}, \bar{b}\rangle$ as vectors of component and bus states $\langle\langle s_1, ..., s_n\rangle, \langle b_1, ..., b_n\rangle\rangle$ where $\forall i \in \{1, ..., n\}.\, s_i$ is a state of a component C_i and $\forall j \in \{1, ..., k\}.\, b_j$ is a state of a bus M_j.

A network configuration contains a vector of current states of components and a vector of current states of buses. Such network configurations make states of the resulting I/O LTS. Transitions of this I/O LTS are defined by the inference rules given in the remaining part of this section.

Let $N = \langle \bar{C}, \bar{M}, L\rangle$ be a network term. We define its semantics $\psi(N)$ as an I/O LTS $\psi(N) = \langle Q_N, T_N, Q_{0_N}\rangle \in \mathcal{L}$ in which:

- The set of states Q_N is given by all the network configurations.
- Q_{0_N} is a set of initial states – these are the configurations in which at least one of the substates is initial state of some network component.
- The transition relation $T_N \subseteq Q_N \times 2^{\mathcal{N}\times\mathcal{R}} \times 2^{\mathcal{N}\times\mathcal{W}} \times Q_N$ is defined using Plotkin-style inference rules, which combine transitions of subsystems with respect to the network hierarchy.

Let $C = \langle S, I, G\rangle$ be a component of the network N. We suppose that T_S is a transition relation of the VCD term S. We define the transition relation $T_C \subseteq Q_C \times 2^{I^R} \times 2^{I^W} \times Q_C$ of the network component C. It is derived from T_S with respect to the interface I and the gate G. There are two cases of which type the subsystem S can be. With respect to this situation, the derivation of T_C from T_S

is defined by one of the following inference rules.

1. In the case when S is a VCD leave, $S = \langle Q_S, T_S, q_{0_S} \rangle$, the transition relation T_C is derived directly from T_S as stated in the following rule:

$$\frac{T_S : \qquad q \xrightarrow[\Delta]{\Gamma} q'}{T_C : \qquad q \xrightarrow[I^W \cap \Delta]{I^R \cap \Gamma} q'}$$

   The only difference between T_S and T_C is that events of T_S which are not in the component interface are abstracted in T_C by deleting them. Note that this rule also lifts internal leave $q \xrightarrow[\emptyset]{\emptyset} q'$ transitions to internal component transitions.

2. For $S = \langle \bar{C}, \bar{M}, L \rangle$ a network term we have the rule:

$$\frac{T_S : \qquad \langle \bar{s}, \bar{b} \rangle \xrightarrow[\Delta^\times]{\Gamma^\times} \langle \bar{s}', \bar{b}' \rangle}{T_C : \qquad \langle \bar{s}, \bar{b} \rangle \xrightarrow[G^{-1}(\Delta^\times)]{G^{-1}(\Gamma^\times)} \langle \bar{s}[i := q_i'], \bar{b} \rangle}$$

   Notation $\Gamma^\times \subseteq \{\langle i, w \rangle \,|\, , i \in \mathcal{N}, w \in \mathcal{W}\}$ denotes a set of indexed input events. Similarly, $\Delta^\times \subseteq \{\langle i, r \rangle \,|\, i \in \mathcal{N}, r \in \mathcal{R}\}$ denotes a set of indexed output events. $G^{-1}(\Gamma^\times)$ stands for the set of ports in network interface I with which events in $\Gamma^\times$ are related by the gate G. Analogously, similar notation is also used for the indexed output events $\Delta^\times$. $\bar{s}[i := q_i']$ is the state vector which was constructed from $\bar{s}$ by replacing its ith component with the state q_i'.

   In the same way like the previous rule, this rule also propagates the internal events and abstracts from those events of the network S which are not assigned to any port of the interface I.

Now we are going to establish rules which define the transition relation T_N. It will be derived from the component transition relations T_{C_i} and the transitions of buses. The key feature of these rules is building of network configurations (global state vectors) from component configurations (local state vectors).

At first, we add to T_N all the component transitions which are totally independent of any bus interconnections. The following rule defines interleaving behavior of components in the network N.

$$3. \quad \frac{T_{C_i} : \qquad \bar{s}[i] \xrightarrow[\Delta]{\Gamma} q_i' \quad \langle i, \Gamma \rangle \subseteq \epsilon_R(N), \langle i, \Delta \rangle \subseteq \epsilon_W(N)}{T_N : \qquad \langle \bar{s}, \bar{b} \rangle \xrightarrow[\langle i, \Delta \rangle]{\langle i, \Gamma \rangle} \langle \bar{s}[i := q_i'], \bar{b} \rangle}$$

Notation $\bar{s}[i]$ denotes the state configuration of the ith component of N. Note that internal component events are lifted by this rule too.

Finally, we are approaching to the last inference rule, which is the most complex one. It puts together transitions of buses and transitions of components and evaluates their relationships given by the network links. According to the evaluated result it can then coordinate some components by firing their transitions synchronously with transitions of some buses. Before we will define such a coordination rule, we have to look deeper into the structure of the network.

Let $N = \langle \langle C_1, ..., C_n \rangle, \langle M_1, ..., M_m \rangle, L \rangle$ be a network for some $m, n \in \mathcal{N}$. With respect to the link relation L some strongly connected blocks of components may be distinguished in the network. For each such a block of components we will define a synchronising coordination rule. From the semantical point of view, any such a separated block of components can internally synchronise while different blocks put together may only mutually interleave. In other words, these blocks are the maximal groups of components with potential synchronous behavior.

To capture the partitioning idea formally, we define a relation R, $R \subseteq \{1, ..., n\} \times \{1, ..., n\}$:

$$\begin{aligned} \langle i, j \rangle \in R \overset{df}{\Leftrightarrow} i = j \vee \exists k \in \{1, ..., m\}, p_i \in I_i, p_k \in I(M_k), p_j \in I_j . \\ \langle \langle i, p_i \rangle, \langle k, p_k \rangle \rangle \in L \wedge \langle \langle j, p_j \rangle, \langle k, p_k \rangle \rangle \in L \end{aligned}$$

It is worth noting that R is an equivalence. We will note $\{1, ..., n\}_{|R} \subseteq 2^{\{1,\cdots,n\}}$ set of all classes of equivalence over the set of component indeces $\{1, ..., n\}$.

Let $\Omega \in \{1, ..., n\}_{|R}$. We will denote $\Omega' \subseteq \{1, ..., m\}$ a set of indeces of buses which are connected to components indexed by Ω. Precisely,

$$\Omega' = \{i \in \{1, ..., m\} \mid \exists k \in \Omega, p_k \in I_k, p_i \in I(M_i). \langle \langle k, p_k \rangle, \langle i, p_i \rangle \rangle \in L\}.$$

Now let $q \equiv \langle \bar{s}, \bar{b} \rangle$ be an actual configuration of network N. We define sets $ET_\Omega(q)$ and $ET_{\Omega'}(q)$ of all transitions starting in q and indexed by their component (respectively bus) indeces:

$$\begin{aligned} ET_\Omega(q) = \{\langle i, t \rangle \mid \forall i \in \Omega. t \in T_{C_i}, \, src(t) = \bar{s}[i]\} \\ ET_{\Omega'}(q) = \{\langle i, t \rangle \mid \forall i \in \Omega', t \in T(M_i). \, src(t) = \bar{b}[i]\} \end{aligned}$$

The notation $src(t)$ denotes the source state of the transition t and $T(M_i)$ denotes the transition relation of the bus M_i.

To precisely characterise the set of all component transitions which can be synchronised with buses resulting in the one global network transition, we have to put some constraints on $ET_\Omega(q)$ and $ET_{\Omega'}(q)$. Firstly, we require that for each source state only one transition is included. Formally, we say $ET_\Omega(q)$ is *consistent* if and only if $\forall i, j, t, t'. \langle i, t \rangle \in ET_\Omega(q) \wedge \langle j, t' \rangle \in ET_\Omega(q) \Rightarrow i \neq j$.

Further we define a triggering relation among component and bus transitions of a particular partition of current network configuration. Firstly we extract some sets of events from the sets of component (bus) transitions $ET_\Omega(q)$ and $ET_{\Omega'}(q)$. In the following definitions, the notations $\Delta(t)$ and $\Gamma(t)$ denote all the output (input) events which occur in the label of the transition t.

- $\mathcal{E}_\Delta(\Omega) = \{\langle i, w \rangle \mid \exists \langle i, t \rangle \in ET_\Omega(q). w \in \Delta(t) \wedge \langle i, w \rangle \in \epsilon_W(N)\}$
- $\mathcal{E}_\Gamma(\Omega) = \{\langle i, r \rangle \mid \exists \langle i, t \rangle \in ET_\Omega(q). r \in \Gamma(t) \wedge \langle i, r \rangle \in \epsilon_R(N)\}$
- $\mathcal{F}_\Delta(\Omega) = \{\langle j, w' \rangle \mid \exists i \in \Omega, w \in \mathcal{W}, \langle i, t \rangle \in ET_\Omega(q). \langle \langle i, w \rangle, \langle j, w' \rangle \rangle \in L$
  $\wedge w \in \Delta(t)\}$
- $\mathcal{F}_\Gamma(\Omega) = \{\langle j, r' \rangle \mid \exists i \in \Gamma, r \in \mathcal{R}, \langle i, t \rangle \in ET_\Omega(q). \langle \langle i, r \rangle, \langle j, r' \rangle \rangle \in L$
  $\wedge r \in \Gamma(t)\}$
- $\mathcal{A}_\Delta(\Omega') = \{\langle i, w \rangle \mid \exists \langle i, t \rangle \in ET_{\Omega'}(q). \, w \in \Delta(t)\}$

- $\mathcal{A}_\Gamma(\Omega') = \{\langle i, w\rangle \mid \exists\langle i, t\rangle \in ET_{\Omega'}(q).\, w \in \Gamma(t)\}$

We say $ET_\Omega(q)$ *triggers* $ET_{\Omega'}(q)$ iff the following two conditions hold:

(i) $\mathcal{A}_\Gamma(\Omega') = \mathcal{F}_\Delta(\Omega)$

(ii) $\mathcal{A}_\Delta(\Omega') = \mathcal{F}_\Gamma(\Omega)$

For each partition Ω we now define the final coordination rule:

$$4.\quad \frac{ET_\Omega(\langle \bar{s}, \bar{b}\rangle) \text{ and } ET_{\Omega'}(\langle \bar{s}, \bar{b}\rangle) \text{ consistent, } ET_\Omega(\langle \bar{s}, \bar{b}\rangle) \text{ triggers } ET_{\Omega'}(\langle \bar{s}, \bar{b}\rangle)}{T_N : \quad \langle \bar{s}, \bar{b}\rangle \xrightarrow[\mathcal{E}_\Delta(\Omega)]{\mathcal{E}_\Gamma(\Omega)} \langle \bar{s'}, \bar{b'}\rangle}$$

where:

$\bar{s'}[i] = s'_i$, if $\exists t \in T_{C_i}.\, t \in ET_\Omega(\langle \bar{s}, \bar{b}\rangle)$ so that $trg(t) = s'_i$
$\bar{s}[i]$, otherwise

$\bar{b'}[i] = b'_i$, if $\exists t \in T(M_i).\, t \in ET_{\Omega'}(\langle \bar{s}, \bar{b}\rangle)$ so that $trg(t) = b'_i$
$\bar{b}[i]$, otherwise

The notation $trg(t)$ denotes the target state of the transition t.

4 Examples of Bus Classes and Instances

In this section we will demonstrate how the classes of busses can be defined in the VCD framework and how they can be instantiated in concrete cases of system specification. Heterogeneity of the coordination layer is also highlighted.

The coordination layer of the model in the figure 3 contains two types of interaction – asynchronous bi-party message passing and synchronous broadcast. Thus, the coordination model of this network is heterogeneous. Each of these two coordination mechanisms is represented in the network as a relevant bus (*BUF* for buffered coordination and *BC* for broadcast). In the following subsections we show how we can formally define classes of these buses and how they can be instantiated in the context of other components of the network.

4.1 Synchronous multicast coordination model

In this subsection we will focus on modeling of synchronous multicast interaction in the coordination layer of the VCD formalism. More specifically, we are going to define a bus class for this mechanism of component interaction. First of all we will recall the situation described in section 2. In Fig. 3 there is a bus instance *BC*. Its intended behavior is to accept the event *parsed_out* and to synchronously resent it to *PM*, *RFU* and *PEU* units to their *parsed_in* ports (*in* in the case of the *PM* unit). We can model the semantics of this bus as a state transition diagram depicted in Fig. 5. The conflict of ambiguous port names in the context of the bus is solved by indexing them. Formally, in our semantic framework developed in section 3.2 components in the network are indexed and so are their ports. Injective matching of them to relevant bus ports along the network links avoids from any port-name conflicts.

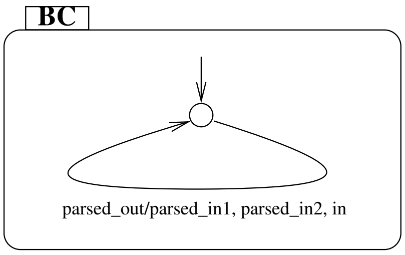

Fig. 5. Semantics of multicast bus instance

In general, the role of synchronous multicast bus is to non-deterministically choose an event involved on one of its input ports, replicate that event and synchronously transfer it to all its output ports. Note that this behavior does not depend on the number of components connected to the bus. Hence, we can abstract from the number of ports the bus contains in its interface. This abstraction is the key knowledge which allows us to construct a bus class. Its instances then have the number of ports in the bus interface bounded (as the number of links to other components is finite). Instantiation is done with respect to the context in which bus instances are placed. In the example of the BC bus referred in the previous paragraph the BC bus is an instance with one input and three output ports.

Formally we define class $\mathcal{B}_{mc}$ of synchronous multicast buses as the following one-state I/O LTS:

$$\mathcal{B}_{mc} = \langle \{q_0\}, T, q_0 \rangle$$

Transition relation T is infinite countable set defined by the following expression:

$$\forall w \in \mathcal{R}, \Delta \subseteq \mathcal{R}, \Delta \neq \emptyset.\ \langle q_0, \{w\}, \Delta, q_0 \rangle \in T$$

4.2 Asynchronous message passing coordination model

As another example of bus class definition we present here a coordination mechanism of asynchronous bi-party message passing. The function of this coordination model is to receive an event from one component and store it in memory until it is taken by another component. It can be taken as an one-cell buffer. Note that for simplicity we have defined non-value-passing version of VCD in this paper, so the memory here handles only the information about the occurrence of the input event. See the BUF bus in Fig. 3. The exact behavior of this bus is showed in the state transition diagram in Fig. 6.

Using the power of the VCD coordination layer we now would like to generalise the notion of the asynchronous message passing bus. More precisely, similarly as in the previous subsection we define a new bus class for this purpose. In this definition we abstract from the concrete number of input and output ports and we base the relevant infinite state transition model on non-determinism of possible asynchronous bi-party interactions. In the bus instances this number is then bounded with respect to the number of surrounding components.

Formally we define class $\mathcal{B}_{amp}$ of asynchronous message passing buses as the

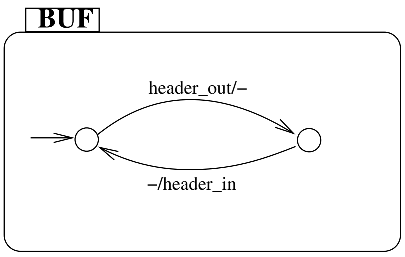

Fig. 6. Semantics of asynchronous message-passing bus instance

following I/O LTS:

$$\mathcal{B}_{amp} = \langle Q, T, q_0 \rangle$$

- $Q = \{q_w \mid w \in \mathcal{W}\} \cup q_0$
- T is defined by the following expression:

$$\forall w \in \mathcal{W}.\ \langle q_0, \{w\}, \emptyset, q_w \rangle \in T \wedge \forall q_x \in Q.\ \langle q_x, \emptyset, \{x\}, q_0 \rangle \in T$$

More complex types of bus classes modeling asynchronous bounded and unbounded coordination models can be defined following the scenario presented above. Together with the possibility of instancing different bus classes in the context of a particular network it demonstrates the heterogeneity of the VCD coordination layer.

5 Conclusions and Future Work

In this paper we have presented the language VCD for hierarchical specification of component-based concurrent systems with heterogeneous models of coordination. The key concept of the language are buses which represent coordination models used in system architectures. Due to its heterogeneous character, VCD can be taken both as an extension of classical software architecture modeling notations and also as a framework for specification of coordination in reactive systems.

We see the main contributions of our work in three ways. First of all, the component-based character of the language together with its hierarchical structure underlied with precise operational semantics allows to join the traditional software and hardware design methods with the formal methods known from the theory of process algebras (e.g., refinement, equivalence or model checking). On the other hand, the both syntactical and semantical separation of modeling the coordination aspects from modeling the behavioral aspects makes it possible to define a static communication infrastructure of a system independently of modeling the behavioral parts. Finally, heterogeneity supported in both behavioral and coordination layers of the language allows not only mixing of various coordination models in one specification, but also using of different models for behavioral description of components. For example, it is possible to put components defined as Statecharts together with components defined as Petri Nets and specify coordination relations among them using the constructs of the VCD coordination layer.

We are currently implementing a graphical tool which allows VCD diagrams to be simply created and modified. In our future work, we would like to add the typed value-passing support to VCD. We also aim to make a precise analysis of relations of our language with other formalisms, especially with process algebras. The key aspect to be investigated here is the language expressiveness and some properties of the semantics, mainly the compositionality. We would like to bring the notion of equivalences known from process algebraic theories and adapt them to VCD. In the future work on tool support, we aim to connect the editor of VCD with the distributed verification environment DiVinE [11].

References

[1] Barnat, J., T. Brázdil, P. Krčál, V. Řehák and D. Šafránek, *Model checking in IPv6 Hardware Router Design*, Technical Report 07, CESNET (2002).

[2] Bergstra, J. A. and P. Klint, *The discrete time ToolBus — a software coordination architecture*, Science of Computer Programming **31** (1998), pp. 205–229.

[3] Berry, G., *The Foundations of Esterel*, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner* (1998).

[4] Ciancarini, P., *Coordination Models and Languages as Software Integrators*, ACM Computing Surveys **28(2)** (1996), p. 300.

[5] Cleaveland, R., X. Du and S. A. Smolka, *GCCS: A Graphical Coordination Language for System Specification*, in: *Proceedings of COORD'00* (2000).

[6] Harel, D., *Statecharts: A Visual Formalism for Complex Systems*, Technical report, The Weizmann Institute (1987).

[7] ITU-TS, "ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)," ITU-TS, Geneva, 1993.

[8] Leue, S., "Methods and Semantics for Telecommunications Systems Engineering," Ph.D. thesis, University of Berne (1994).

[9] Maggiolo-Schettini, A., A. Peron and S. Tini, *A comparison of statecharts step semantics*, Theoretical Computer Science **290** (2003).

[10] Papadopoulos, G. A. and F. Arbab, *Coordination models and languages*, in: *761*, Centrum voor Wiskunde en Informatica (CWI), 1998 p. 55.

[11] ParaDiSe Lab, Masaryk University Brno, "DiVinE project home page," (2004). URL http://anna.fi.muni.cz/divine

[12] Pezze, M. and S. Shatz, *Software Engineering and Petri Nets*, in: *Proceedings of the Workshop on Software Engineering and Petri Nets*, 2000.

[13] Ray, A. and R. Cleaveland, *Architectural Interaction Diagrams: AIDs for System Modeling*, in: *Proc. of the 25th International Conference on Software Engineering (ICSE 2002)* (2003).

[14] von der Beeck, M., *Formalization of UML-Statecharts*, in: *Proceedings of UML 2001*, LNCS (2001).

[15] Šafránek, D., *SGCCS: A Graphical Language for Real-Time Coordination*, in: *Proceedings of FOCLASA'02*, ENTCS **68** (2002).