VCD: A Visual Formalism for Specification of Heterogeneous Software Architectures*

David Šafránek and Jiří Šimša

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
{xsafran1,xsimsa}@fi.muni.cz

Abstract. A visual formalism called Visual Coordination Diagrams (VCD) for high-level design of heterogeneous systems is presented in this paper. The language is based on a state-transition operational semantics, which allows application of formal methods to software design. Formal definition of VCD is included in the paper. Moreover, an example of use of the language is also given.

1 Introduction

The importance of visual modeling languages such as UML [12] is very significant in the domain of software engineering. The desired properties of such an universal visual design language are heterogeneity, hierarchy and component-based structure. Additionally, to be able to analyze the software design using formal methods, some unambiguous formal semantics is required. Unfortunately, there is no formal semantics of UML [6].

In this paper we present Visual Coordination Diagrams (VCD) – a *visual formalism* for specification of component-based distributed systems, based on the idea of GCCS [3] and its extensions [18]. The VCD formalism can be viewed as static architecture diagrams for specification of 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 specification. The second level of the 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 supported coordination models.

The work on VCD is practically motivated by the formal verification project Liberouter [2]. In this project, we have to deal with formal modeling of a complex system composed of heterogeneous SW/HW units [7].

* This work has been supported by the Grant Agency of Czech Republic grant No. 201/03/0509, the FP5 project No. IST-2001-32603 and the CESNET activity Programmable hardware.

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 fulfill the needs of design of complex systems. Combining the concept of geometric inclusion with the concept of hi-graphs, the hierarchy of states has been added, leading to Harel's Statecharts [4]. The complexity of the syntactic richness of Statecharts has shown that reaching a compositional formal semantics for such a powerful language is impracticable. Various sub-dialects of Statecharts have been defined to achieve required semantic properties [10]. The concept of Statecharts was also incorporated in UML [17], [5].

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 notation does not support hierarchical design. For its simple nature, it is widely used in telecommunication industry and it is also a part of UML.

VCD extends and generalizes ideas of the work on Graphical calculus of communicating systems (GCCS) [3] and its synchronous extension SGCCS [18], which adopt the process algebraic approach as the underlying semantic model. In these languages, a very tight relation to the underlying process algebraic semantic model limits the heterogeneity of both coordination and behavioral layers. I.e., it is difficult to incorporate Statechart-like formalisms into GCCS. In VCD we try to overcome these inconveniences by using of a more general semantic model.

There is another architectural language, which is, similarly to VCD, based on the idea of GCCS. It is called Architectural Interaction Diagrams (AID) [15]. VCD and AID both achieve some level of heterogeneity by avoiding the tight relation with the CCS process algebra [11]. 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.

Similarly to some of the classical textual architecture description languages (ADLs) like Wright [1] or UniCON [16], VCD are based on the idea of taking connectors and computational components as different elements of system architecture. Moreover, this concept is further refined in VCD. In contrast to Wright, where the semantics of connectors is defined in terms of CSP processes, which are based on handshake-style coordination, in VCD more complex coordination mechanisms, e.g. multi-cast, can be modeled more conveniently. It is mainly due to the fact of semantic modeling of any coordination event, e.g. a broad-cast communication, as one atomic transition of a connector model. In the domain of ADLs, there are some languages which support dynamic changes of system structure, i.e. Darwin [9] or SOFA [14]. Unlike these languages, VCD does not support dynamism. VCD is aimed to be a simple visual formalism for hierarchical description of coordination of system components.

The main reason for developing VCD is our belief in importance of building a formal 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 one global transition system formally representing the whole architecture description.

2 Overview of VCD

VCD is aimed to be a formal language for specification of communication relationships in component-based systems. An example of a simple VCD is depicted in Fig. 1. Basic elements of the VCD formalism are component *interfaces*. Each interface contains input and output *ports*. Interfaces are organized in so called *networks* in which they can be connected by *links* to *buses*.

Buses represent connectors of components. They 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. In VCD there is a concept of *bus classes*, which allows to specify generic templates for various coordination media.

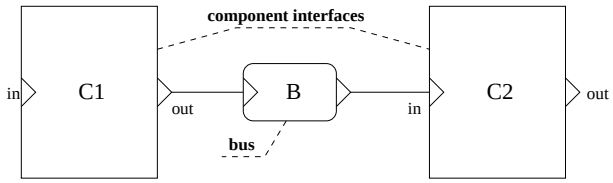

Fig. 1. A network of components *C1* and *C2* connected to bus *B*

The key concept of VCD is in its hierarchical network structure, which unfolds the *coordination layer*. This is achieved by the possibility of taking networks as components of other networks (higher-level networks). The relation between a network and its enclosing interface is defined by a *gate*. Gate maps ports of the lower-level network to ports of the enclosing interface in the higher-level network. An example of a network hierarchy is given in Fig. 2.

At the bottom-most level, behavior of system components has to be specified explicitly. There is no direct visual notation for behavioral specification in VCD. Instead of that, the formal semantic framework of so called VCD *leaves* is defined. It is called *behavioral layer*. The behavioral layer is based on the semantic model given by the notion of input/output labeled transition system (LTS) with sets of input and output actions taken as labels. This allows any language with semantics defined 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 at this level is achieved with respect to the set of semantically compatible, but notationally different languages, which can be incorporated to VCD for the purpose of behavioral description. As examples of supported languages variants of Statecharts or Petri-Nets can be mentioned.

Semantics of VCD is based on a 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 buses. Semantics of a particular bus class represents behavior of a specific communication media.

3 Syntax and Semantics of VCD

In this section, the formal syntax of VCD is defined and its semantics given.

3.1 Syntax

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

Ports and Interfaces The most basic elements of the coordination layer are *interfaces* with *ports*. We fix $\mathcal{W}$ a countable set of *write ports* and $\mathcal{R}$ a countable set of *read ports*. Interface I is defined as a pair consisting of a finite set of input ports and a finite set of output ports — $I = \langle W, R \rangle$, $W \subseteq \mathcal{W}$, $R \subseteq \mathcal{R}$, $W \cap R = \emptyset$. We mark projections $I^W = W$ the *write-interface*, $I^R = R$ the *read-interface*, respectively.

Buses and Bus Classes The key construct of the coordination layer is *bus*. As it has been mentioned in the previous section, buses represent coordination mechanisms. Particular types of coordination mechanisms are represented as *bus classes*, which are formally defined as input/output labeled transition systems (I/O LTS).

Definition 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 (countable) transition relation.*

Any bus class can be instantiated as a particular bus and used for specification of concrete connections among components in a network. The bus interface is determined by the set of links which connect the bus to the ports of surrounding components. The finiteness of the bus interface puts a finite bound to the transition relation of a bus instance. Formal definition of bus instance, given by its interface and its class, is the following.

Definition 2. Bus instance 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 instance B will be denoted as $I(B)$.

Gates, Networks and Leaves Now we are going to define terms which formally represent VCD network diagrams. 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 dashed links will be formalized as the notion of *gate*.

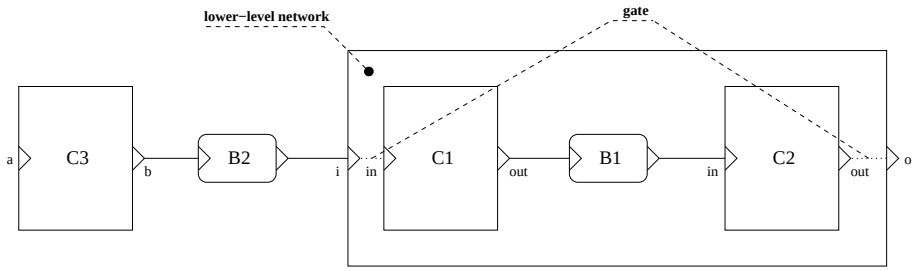

Fig. 2. Network hierarchy

Definition 3. *A* VCD term *is:*

1. VCD leaf – *behavioral model specified in any LTS-compatible language*
2. 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 *. . . 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:*
      i. $p_1 \in \mathcal{W} \Leftrightarrow p_2 \in \mathcal{R}$
      ii. $p_1 \in I_i^W \cup I_i^R$
      iii. $p_2 \in I^W(M_j) \cup I^R(M_j)$
      iv. $\langle \langle l, p_1' \rangle, \langle j, p_2 \rangle \rangle \in L \Leftrightarrow l = i \wedge p_1' = p_1$
      v. $\langle \langle i, p_1 \rangle, \langle l, p_2' \rangle \rangle \in L \Leftrightarrow l = j \wedge p_2' = p_2$

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

To formalize the feature of embedding a network into a higher-level network, we set up a function ϵ_R (ϵ_W) which for any VCD network returns a set of all its read (write) ports which have no connection to any bus. We call such ports *free ports*. To overcome ambiguity of port names in the context of a network, we index all the component interfaces in the scope of a particular network, and mark each port with the index of its interface.

Definition 4. *Let $N = \langle \bar{C}, \bar{M}, L \rangle$ be a network.*

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

We define *interface* of network N as a pair $I(N) = \langle \epsilon_W(N), \epsilon_R(N) \rangle$.

Gate is formally defined as a partial function relating ports of a particular component interface to free ports of the network which is nested in that interface. In the case when the nested structure is a leaf, the gate maps interface ports to eponymous actions of the nested process.

Definition 5. *Let I be an interface.*

1. *Let S be a VCD leaf encapsulated in the interface I. Let $ports(S) \subseteq \mathcal{W} \cup \mathcal{R}$ be a set of all actions of S. We define a* gate *of the leaf S as the identity function $G : I^W \cup I^R \to ports(S)$, $\forall x \in I^W \cup I^R . G(x) = x$.*
2. *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 $G : I^W \cup I^R \to I(S)$ satisfying:*
   - $\forall w \in I^W . G(w) = \langle i, w' \rangle \wedge \langle i, w' \rangle \in \epsilon_W(S)$
   - $\forall r \in I^R . G(r) = \langle i, r' \rangle \wedge \langle i, r' \rangle \in \epsilon_R(S)$

3.2 Semantics

In this subsection, the definition of the structural operational semantics of VCD terms is sketched. Its precise definition is presented in the full version of this paper [19].

As a semantic domain a class $\mathcal{L}$ of input/output labeled transition systems (I/O LTS) with sets of input and output actions in transition labels is used. Formally, the semantics is defined as a mapping $\psi : \mathcal{S} \to \mathcal{L}$ which assigns an I/O LTS to each VCD term.

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

Definition 6. *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 atomic components. As VCD does not include any predefined syntactic construct for the behavioral layer, this I/O LTS is the structure in which the formalisms for behavioral description have to be encoded.

At the coordination layer, the semantics of a VCD network is defined as a global I/O LTS which composes transitions of local I/O LTSs representing the semantics of network components. This composition is realized with respect to the coordination model given by the specific bus classes instantiated in the network. States of the global I/O LTS are represented as *network configurations*. They respect the hierarchical structure of network terms. The formal definition of a network configuration is the following.

Definition 7. *Let* $N = \langle\langle C_1, ..., C_n\rangle, \langle M_1, ..., M_k\rangle, L\rangle$ *be a network. We define its* configuration $\langle \bar{s}, \bar{b}\rangle$ *as a vector 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 determine states of the resulting I/O LTS. Transitions of the global I/O LTS are defined with respect to the network hierarchy using Plotkin-style inference rules.

Fig. 3. Subsystem S embedded in a component C

In figure 3, there is a scheme how the component C is built by embedding of the subsystem S into a component interface. The subsystem S can be either a leaf or a network. Transition system of the component C is derived from the transition system of the embedded subsystem with respect to ports in the interface. Actions of S which have no ports in the interface of component C are hidden. The structure of the relevant inference rule for the situation when S is a leaf is the following.

$$\frac{\text{transition of S:} \quad q \xrightarrow[\Delta]{\Gamma} q' \qquad (\Gamma \subseteq ports(S) \cap \mathcal{R} \text{ and } \Delta \subseteq ports(S) \cap \mathcal{W})}{\text{transition of C:} \quad q \xrightarrow[I^W \cap \Delta]{I^R \cap \Gamma} q'}$$

A similar inference rule captures the case when S is a network. This case is more complicated because of possible ambiguity of port names in the network scope. In appendix, the inference rule for this case is precised.

In figure 4, there is a scheme of a network with n components arbitrarily connected to m buses. For simplification, the links are not depicted. To define a global transition system for the network N, transition systems of the components and buses have to be composed. There are two different situations:

- Stand-alone components — their transitions are interleaved.
- Components connected to buses — their transitions are interleaved or synchronized w.r.t. semantics of instantiated bus classes.

The inference rules for both of the situations above are defined in [19].

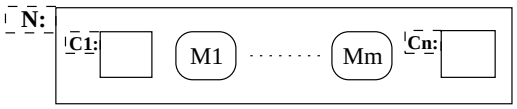

Fig. 4. Components $C_1 \ldots C_n$ and buses $M_1 \ldots M_m$ embedded in a network N

4 An Example of Architectural Specification in VCD

In this section we will demonstrate on a very simple example how the VCD formalism, especially the concept of bus classes, can be used for specification of a distributed software architecture.

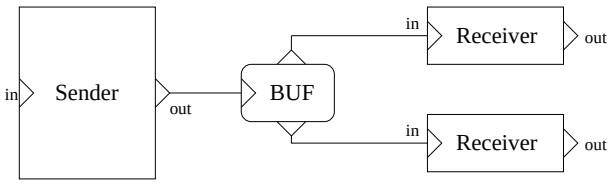

Fig. 5. Example of a concrete VCD network

In distributed software architectures, components of systems interact most typically in asynchronous way. One of the coordination mechanisms which captures this flavor of interaction is asynchronous message passing. In figure 5 there is a simple VCD network which specifies a distributed system with three components. It can be taken as a part of the specification of some communication protocol. There are one sender and two receiver components in the system. The intended behavior of the sender is to pass the output information to the communication media and continue some inner computation. On the other side, the behavior of any receiver is to take the information from the media asynchronously with computations of the sender.

To model this kind of interaction in the VCD framework, we establish a class $\mathcal{B}_{amp}$ of asynchronous message passing buses. It can be formally defined as the I/O LTS $\mathcal{B}_{amp} = \langle Q, T, q_0 \rangle$ where:

- $Q = \{q_w \mid w \in \mathcal{W}\} \cup q_0$
- T is defined by disjunction of the following expressions:
  1. $\forall w \in \mathcal{W}.\ \langle q_0, \{w\}, \emptyset, q_w \rangle \in T$
  2. $\forall q_x \in Q.\ \langle q_x, \emptyset, \{x\}, q_0 \rangle \in T$
  3. $\forall q_x \in Q.\ \langle q_x, \emptyset, \emptyset, q_x \rangle \in T$

The first expression defines the reaction of the bus to incoming write-actions. In the second expression, the interaction with receiver components is solved. The last expression adds the empty self-transitions, which allow the interactions to be potentially asynchronous.

The countable transition relation, which is the part of the bus class defined above, is made finite by the process of instantiation. In the example depicted in figure 5, the bus class $\mathcal{B}_{amp}$ is instantiated and placed in the context of three components. Thus, the number of transitions is bounded by the number of links which interconnect the bus with the surrounding components. In figure 6, there is the resulting transition system which represents this bus instance.

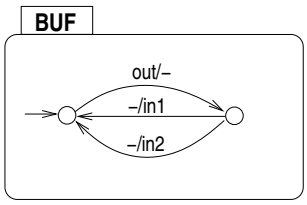

Fig. 6. Instance of asynchronous message-passing bus class

More complex types of bus classes modeling both synchronous and asynchronous coordination models can be defined following the scenario presented above. Together with the possibility of instancing different bus classes in the context of one particular network, this example demonstrates the heterogeneity of the VCD coordination layer.

5 Conclusions and Future Work

In this paper we have presented the formalism VCD for hierarchical specification of heterogeneous system architectures. The key concept of the language are buses which represent coordination models used in system architectures.

We see the main contributions of our work in three ways. First of all, the component-based character of VCD together with its hierarchical structure based on precise operational semantics allows to join the traditional design methods with the formal methods known from the theory of process algebras. 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 implementing a graphical tool [19] 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 would like to bring the notion of equivalences known from process algebraic theories and adapt them to VCD. We also aim to connect the editor of VCD with the distributed verification environment DiVinE [13].

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.
2. D. Antoš, O. Fučík, and J. Novotný. Project of IPv6 Router with FPGA Hardware Accelerator. In *Proceeding of 13th International Conference on Field-Programmable Logic and Applications*, volume 2778, pages 964–967. LNCS, Springer-Verlag, 2003.
3. R. Cleaveland, X. Du, and S. A. Smolka. GCCS: A Graphical Coordination Language for System Specification. In *Proceedings of COORD'00*. LNCS, Springer Verlag, 2000.
4. D. Harel. Statecharts: A Visual Formalism for Complex Systems. Technical report, The Weizmann Institute, 1987.
5. D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *Proc. of 3rd Int. Workshop on Integration of Software Specification Techniques for Applications in Engineering*, volume 3147, pages 325–354. LNCS, Springer-Verlag, 2004.
6. D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Technical Report MSC00-16, Weizman Institute of Science, 2000.
7. J. Holeček Jan, T. Kratochvíla, V. Řehák, D. Šafránek, and P. Šimecek. How to Formalize FPGA Hardware Design. Technical Report 4/2004, CESNET z.s.p.o., 2004.
8. S. Leue. *Methods and Semantics for Telecommunications Systems Engineering.* PhD thesis, University of Berne, 1994.
9. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.
10. A. Maggiolo-Schettini, A. Peron, and S. Tini. A Comparison of Statecharts Step Semantics. *Theoretical Computer Science*, 290, 2003.
11. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
12. OMG. *Unified Modeling Language. Version 2.0.* OMG, 2003.
13. ParaDiSe Lab, Masaryk University Brno. *DiVinE project home page*, 2004.
14. F. Plášil, D. Bálek, and R. Janeček. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the International Conference on Configurable Distributed Systems*, page 43. IEEE Computer Society, 1998.
15. A. Ray and R. Cleaveland. Architectural Interaction Diagrams: AIDs for System Modeling. In *Proc. of ICSE 2003*. IEEE, 2003.
16. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.
17. Michael von der Beeck. Formalization of UML-Statecharts. In *Proceedings of UML 2001*, LNCS. Springer-Verlag, 2001.
18. D. Šafránek. SGCCS: A Graphical Language for Real-Time Coordination. In *Proceedings of FOCLASA'02*, volume 68 of *ENTCS*. Elsevier Science, 2002.
19. D. Šafránek and J. Šimša. Visual Formalism for Specification of Heterogeneous Software Architectures. Technical Report FIMU-RS-2004-11, Faculty of Informatics, Masaryk University Brno, 2004.