Uniform representation of transition concepts in Domain Specific Visual Languages [1]

Paolo Bottoni[2], Maria De Marsico, Paolo Di Tommaso, Stefano Levialdi, Domenico Ventriglia

Department of Computer Science, University of Rome "La Sapienza", Italy

{bottoni,marilena,levialdi}@dsi.uniroma1.it, ventriglia1@yahoo.it, paolodt@yahoo.com

Abstract

Domain specific visual languages are generally used to specify significant configurations and behaviours of systems of interest for the users and diagrams of different types can be used to specify different components of a single system. At an abstract level, all these diagrams express some form of transformation of the system, which can be characterised by its pre- and post-conditions and by a policy presiding at its execution. We propose a uniform approach to the management of these transitions, independently of the adopted diagrammatic notation, which can be used for bidirectional binding between modifications of the visual representation and the transformations of the underlying model.

1 Introduction

Domain specific visual languages are generally used to specify significant system configurations and behaviours (i.e. possible configuration changes) of interest for the users. The choice of the model to use for behaviour specification is often bound to characteristics of the problem at hand. We restrict our study to discrete event models, among the most common in the field of visual languages.

Complex models may require the composition of systems specified in different ways, e.g at different layers. For example, a producer-consumer system can be modelled as a Petri net when one wants to focus on the dependence between the two activities, while the specific behaviours of the systems acting as producers or consumers can be defined in completely different ways, maybe using different visual languages. In these cases, one is either forced to adopt one single model of visual specification, thus renouncing to domain-specific notations, or one has to connect the different processes at some abstract level.

In this paper, we propose the WIPPOG (from the initials of WHEN, IF, PROCESSES, PRODUCES, OUTS, and GETS) language and computational model, which is based on an abstract notion of *transition*. WIPPOG is able to accommodate a set of different concrete models of diagrammatic transformation, so as to decouple the visual appearance of the specification from its semantical content in terms of the represented transition, also decoupling the specification of a transition from the application mechanism defining a transformation step. The combination of a WIPPOG interpreter and of a policy manager, plus mechanisms to manage import and export of resources, constitute a WIPPOG machine.

Section 2 presents the basic concepts at the heart of WIPPOG in the context of literature on linear logic programming and meta-model approaches. Section 3 illustrates the WIPPOG rationale and syntax and Section 4 discusses various aspects related to the implementation and use of WIPPOG specifications. Some conclusions are drawn in Section 5.

[1] Partially supported by Italian Ministry of University and Research and ESPRIT Working Group SEGRAVIS

[2] Corresponding author

2 Related work

The WIPPOG computational model is tied to the LO framework [AP91, ACP93], which is based on the multiplicative fragment of linear logic [Gir95, And92]. In a nutshell, a computational step consumes and produces a finite set of resources in a pool, with the possible transmission of another finite set of resources in a broadcast way to other agents. Transmitted resources are internalised to the agents in the same way as those internally produced. Differently from the LO model, we do not allow creation of new pools, and we consider a registration mechanism through *Import* and *Export* interfaces, rather than using broadcast communication. Such interfaces were also used in Forumtalk [And95], where resources could be transmitted both to other agents in the same virtual location, and to agents whose locations had registered for import.

WIPPOG exploits these ideas to provide an underlying language to express discrete event systems, independent of any specific visual language. In this sense, our work is in the line of metalevel approaches, which offer frameworks for the creation and definition of language syntax and semantics.

Moses [EJ01] follows the approach of [Erw98] to work on an abstract syntax description of visual sentences through attributed graphs. From this specification a syntax-checker is defined. Moses provides an environment in which a user can define a sentence, with the syntax-checker operating in the background. Semantics is dealt with by producing specific interpreters in the form of Abstract State Machines for any given visual language. In our approach, we use WIPPOG both to express the legal actions in a syntax-directed editor generated from the definition of a visual language, and to express the behavioural semantics attached to a diagram in the language, as both are seen as discrete event systems.

The problem of interconnecting components (for system simulation) expressed with different visual formalisms is particularly addressed in the AToM^3 environment [dLV02]. Here, general meta-meta-models generate meta-models in which to define the type of systems to be simulated. Finally, a model defines a specific instance of system. All models are expressed in some visual formalism and model transformations are defined through graph grammars. Component interconnection is solved by mapping different components to a common formalism. In our approach, component interconnection is automatically given by their WIPPOG expression independently from the original visual formalism. The translation from domain specific languages to their WIPPOG translation has to simultaneously consider the metalevel definitions of both the semantics and the syntax of the language. An explicit and automated management of these definitions, exploiting the meta-meta-level approach could be beneficial to WIPPOG as currently the translation of operational semantics from the visual formalism to WIPPOG has to be explicitly coded case by case.

Referring to discrete event systems, the DEVS framework has to be considered [ZPK00]. This framework distinguishes between internal and external transitions, rather than resources, and allows a system to maintain an internal definition of its timebase, and of its connections to other systems. In the WIPPOG approach, transitions are defined as WIPPOG rules, and external connections are managed at the WIPPOG machine level, without having to revise the rule definition.

While WIPPOG is based on multiset rewriting, graph rewriting is used in several environments as a uniform way to express syntax and semantics [BMST99]. Each tool is however tied to a specific application policy and has to recur to hybrid (textual and visual) forms, to steer rule application in some desired way.

3 A presentation of WIPPOG

WIPPOG exploits an abstract notion of *transition* performed by a discrete event system, and details different aspects of its pre- and post-conditions. A set of rules defines the set of the system's possible behaviours, and a pool of resources describes its state at any given moment. A *resource* is a typed item, of the form

```
Item = TypeName '('['id' = Ident ';'] CSLOFAVP ')'
```

where `TypeName` is a string from a set TN identifying the resource *type* in a set T, `Ident` is a string uniquely defining the item, `CSLOFAVP` is a comma-separated list of attribute-value pairs. The resource type determines the names of the attributes from a set Nms, while values are taken from domains associated with the attributes. No attribute can appear twice in the same item. In the current implementation, values can be taken from the domains of the basic types, `int`, `string`, `boolean`, `Id`, and from lists of these types. A type `Object` is provided for user-defined types. The set of all resources definable on T is called $W(T)$.

A transition is conditional resource consumption and production. WIPPOG distinguishes internal resources, to be present in the system, from external ones, received from other systems, or in general, from the environment (this may include resources representing user inputs). In a similar way, resources can be produced to remain in the agent state, or to be diffused in the environment, possibly to be utilized by other agents. Transition preconditions may include checks on the values of attributes of the internal or external resources to be consumed. Values for the attributes of the resources to be produced are computed in a dedicated component of the transition. To meet these needs, variables can be used in the transition to refer to the value of an attribute, and matching is performed against values of existing resources. Variables with the same name are bound to assume the same value.

A WIPPOG transition is therefore formed by six components:

- **WHEN:** resources which must be available internally to the agent. Attributes can be mentioned, with either a constant value, or a variable name.
- **GETS:** externally produced resources which must be available to the agent. Attributes can be mentioned, with a constant value or a variable name.
- **IF:** conditions on variables in the WHEN or GETS components.
- **PROCESSES:** computational activities associated with the transition and considered to be always successful. Assignments can be specified here for attribute values for the resources to be created. Variables introduced in the WHEN or GETS components can be used in the expressions.
- **PRODUCES:** resources to be created by the transition. If variables appear, their names must occur either in the WHEN or GETS components, or in the left-hand side of an assignment in the PROCESSES component.
- **OUTS:** resources to be made externally available. If variables appear, their names must occur either in the WHEN or GETS components, or in the left-hand side of an assignment in the PROCESSES components.

The application of a WIPPOG transition: a) removes from the resource pool available to an an agent (internal or external) those mentioned in the WHEN and GETS components, if the conditions in the IF component are satisfied; b) executes the activities specified in the PROCESSES component; and c) produces (either for internal or external use) the resources specified in the PRODUCES and OUTS components.

As an example, a transition in a finite state machine, going from one state A to a state B when receiving the input c, as shown in Figure 1a), would be encoded by the rule[3]:
WHEN: current(id=A)
GETS: input(value=c)
PRODUCES: current(id=B)

In another example, a transition in a Place/Transition Petri Net removing two tokens from a place $pl1$, one token from a place $pl2$, and inserting a token in place $pl3$, as specified in Figure 1b), would be encoded by the rule:
WHEN: place(id=1; tkns=X) place(id=2; tkns=Y) place(id=3; tkns=Z)
IF: X $\geq$ 2 Y $\geq$ 1
PROCESSES: X1 := X -2 Y1 := Y -1 Z1 := Z +1
PRODUCES: place(id=1; tkns=X1) place(id=2; tkns=Y1) place(id=3; tkns=Z1)

[3] Actually, the correct syntax of WIPPOG is expressed in XML through a DTD. For the sake of conciseness we adopt a simplified syntax here.

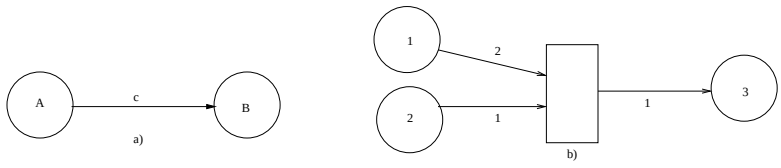

Figure 1: a) A transition in a finite state machine. b) A transition in a PT Petri Net

4 WIPPOG in application contexts

We discuss here some aspects relative to the management of WIPPOG specifications, in particular in the context of visual languages.

4.1 WIPPOG Machines

A *WIPPOG machine* (WM) provides a computational environment for the execution of WIPPOG rules. It consists, as portrayed in Figure 2, of a WIPPOG *Interpreter*, applying one WIPPOG rule at a time, from those present in the *Rule* base, coherently with the role of the different components and according to an activation policy managed by the WIPPOG *Manager*. The process acts on both the *Resource Pool* which contains the resources describing the state of the system, and the *Input* and *Output* compartments of the WM, which represent the system's connection with the other systems.

Several WMs can communicate among them. In particular, let T_1 and T_2 be the set of types for the rules associated with two machines WM_1 and WM_2, called the sets P_1 and P_2. A machine WM_i can declare a set $I_i \subset T_i$ of types as `Import` and a set $E_i \subset T_i$ as `Export`. Now, for $i, j \in \{1, 2\}$, each resource of a type $t \in I_i \cap E_j$ which is declared in the OUTS component of a rule in P_j and produced during a transition, will be placed in the *Output* compartment of WM_j. The communication mechanism removes them from there and places them in the *Input* compartment of WM_i. A rule in P_i, exploiting a resource of type t in its GETS component, will be able to use such resource and remove it from the *Input* component.

The WIPPOG Manager is responsible for activating the interpreter, according to an activation policy, thus realising the notion of a *step*. Three basic activation policies may be defined: *sequential*, *concurrent*, and *maximally concurrent*.

In the sequential policy, the interpreter selects only one set of resources satisfying the preconditions of at least one rule for application. In the concurrent policy, a set of non conflicting sets of resources satisfying the preconditions for some rule for application is selected. The maximal concurrency policy applies all possible concurrent rules in such a way that any other application would conflict with the selected instances. All the resources appearing in the sets of preconditions for the applied rules are removed from the pool, while all resources in the sets of postconditions for such rules are added. The completion of a step can trigger some specific activity by other objects in the system. In particular, it triggers the movement from the *Export* of one machine to the *Import* compartment of another machine, and enables possible external observers to refer to the new content of the *Resource Pool*.

4.2 From domain specific languages to WIPPOG specifications

In order to use a WIPPOG machine to support execution of visual specifications, we consider that behaviour specifications are generally of three types.

In a first case, diagrammatic notations are used to specify transformations following some prototypical model such as finite state automata, dataflow diagrams (and their derivations like Prograph [CGP89]), or Petri nets. These models use a diagram to implicitly define the content of the transformation (which can be represented as diagram animation), while adopting an explicit representation of the transition elements, but rely on an external definition to specify its actual operational semantics.

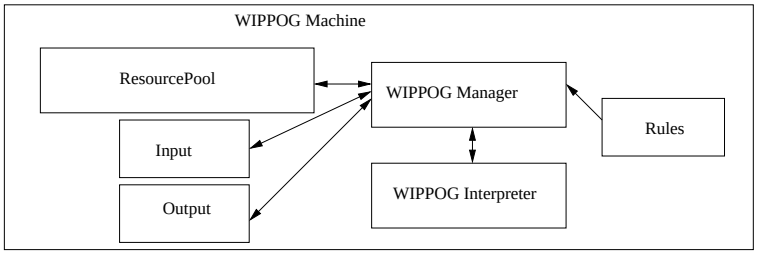

Figure 2: The architecture of a WIPPOG machine

The second family groups those specifications which express transformations as before-after visual rules, following the pioneering works of BITPICT [Fur91], Agentsheets [RS95] or KidSym [SCS94]. These are based on a simple operational semantics (remove the antecedent and substitute it with the consequent), but cannot provide much variance in the application policy, generally using either a sequential or a concurrent one.

Finally, visual rules can be associated with sophisticated application policies or embedding mechanisms, as in graph grammars (for the different approaches to this field, see [Roz97]). In this case, however, rules operate more on an abstract representation of the diagram, rather than on the actual diagram. Transformations can be mapped back to the actual diagram, generally by way of programmed routines [Tae00], or layered transformations [Bar02] (for a survey on applications see [BTMS99]).

For all these cases, a style of translation to a WIPPOG specification can be devised, exploiting metamodel definitions of both the semantics and the syntax of the visual language of interest. The metamodels can be expressed in the UML style via an abstract syntax, typically in the form of class diagrams, and a collection of constraints on the relations among elements of the abstract syntax. The metamodel characterisation of the syntactical aspects of the visual language, as defined in [PG02], assigns the language to a family, such as containment-based, graph-like, adjacency-based, etc. The translation from visual to WIPPOG elements thus depends on the families of models as discussed above, and the specific way of representing relations among elements.

The translation is based on a metalevel definition of the semantics of visual specifications. This relies on the identification of *Semantic Elements*, i.e. of those visual elements through which semantics is defined. In this set, one can distinguish between *Configuration Support Elements* and *Transition Elements*. *Configuration Support Elements* can be in turn divided into *Holders* and *Tokens*. A *Configuration* is a set of *Semantic Elements* and a *Transition* expresses a relation between two *Configurations* (pre- and post- *Configuration*).

For the first family of visual languages, this metamodel is reflected at the visual level, by expressing *Configuration Support Elements* as *Identifiable Elements*, so that a *Visual Configuration* is a set of such elements (possibly with specific values of their appearance attributes, e.g. colour may identify the position of the `current` *Token* in the representation of a finite state automaton). *Transitions* are mapped to peculiar *Identifiable Elements*, which can be *Referrable Elements*, as happens in Petri nets, or *Connections*, as in Finite State Automata. Attributes of *Configuration Support Elements* relevant to define their state are mapped into visual attributes of the *Identifiable Elements*. The last mapping is now to the WIPPOG level. *Identifiable Elements* corresponding to *Configuration Support Elements* are mapped into *WIPPOG Elements*, i.e. resources. In any case, a *WIPPOG Rule* is created for each *Identifiable Element* corresponding to a *Transition*.

For the second family of visual specifications, the main difference at the visual level is the lack of a visual element that explicitly represents a transition. This is anyway expressed by the differences between the pre- and post- *Visual Configurations*. *WIPPOG Rules* are directly derived from these two *Visual Configurations*. The third family relies on a similar mapping, but in addition, it requires a specific coding of the different execution policies.

4.3 From WIPPOG execution to visual representations

The transitions in a WM need, of course, to be mapped back to the pictorial level. On the other hand, the types of attributes involved in the management of a WIPPOG transition may be different from those relevant to the management of the graphical appearance of an item representing a resource. Moreover, one could be interested in observing global properties of a collection of resources, for instance how many resources exist for each different type. Finally, some specific values of an attribute for a resource may need to be reflected through some special appearances of the graphical items.

To this end, we rely on the distinction between *Observers* and *Executors* as defined in the CVE architecture [BBMP93]. In this architecture, Executors are responsible for managing computational resources and activities, while Observers are responsible to format views to be displayed by *Presenters*. Observers are also able to interpret the events generated by the user on the presentation elements and to consequently generate requests to the executors, or adjust presentation aspects, according to the meaning of the event. A notification mechanism ensures that state changes in the Executor are reported to the registered Observers to update the connected presentations.

An Observer for a WM (i.e. an instance of `WIPPOGObserver`) needs to be aware of the set of types T used by the WM, and to define a set of *visual types* VT, together with an injective mapping $\mu : T \rightarrow VT$ connecting the WIPPOG type to the visual type. As the attributes in a type $v = \mu(t)$ may differ for any $t \in T$, the observers must also be equipped with partial mappings $\mu_t : D_{a_i} \rightarrow D_{a_j}$ for some $a_i \in atts(t)$, $a_j \in atts(\mu(t))$. In this way, an Observer, once informed of the completion of a step, can update the pictorial presentation of the current configuration of the system, which may involve removal or insertion of items, or modifications of the appearance of modified items, according to the outcomes of a step.

It is to be noted that in a WIPPOG transition the modification of a resource is possible only if it appears with the same identifier in both the WHEN and the PRODUCES components. In this case, the resource is first removed and then reinserted with the same identifier. Attributes whose value has not been computed in the PROCESSES component maintain the same value. At the graphical level, we do not need to show deletion and reinsertion (and generally we do not want), so that a `WIPPOGObserver` generally waits for the completion of a step to perform the updating, thus being able to show only the net result. Such a mechanism can however be overriden. As a `WIPPOGObserver` needs to implement the `WIPPOGListener` interface, which declares methods to capture the different `WIPPOGEvent`s, one can choose to perform (partial) updates when resources are removed or inserted in any compartment of a WM.

4.4 Using WIPPOG to specify dialogue control in a syntax directed editor

In [BCM99], a construction is presented to translate a set of conditional attributed visual rewriting rules, defining a visual language, into an interaction control automaton enabling or disabling rule application, according to the user selection of possible rule antecedents. The GenIAL system provides an interactive environment where users can define visual alphabets, collections of visual rules and sets of axioms. The user can then construct correct sentences in the corresponding visual language, in a way governed by the generated control automaton. GenIAL exploits the CVE architecture, so that it is possible to integrate it with executors embedding WMs into them and with `WIPPOGObserver`s.

As shown in Figure 3, an `EACBuild` executor translates the specification of a visual language alphabet and rules into two WIPPOG specifications. One, managed by the `automaW` WM, defines the control automaton; the other, managed by the `sentenceW` WM, is an adaptation of the original rules to the WIPPOG syntax. While visual rules may contain attributes pertaining to both the graphical and the semantical aspects, `sentenceW` is only involved in the semantics of the visual language. Graphical attributes are managed by an observer `OACesec`, responsible for the visualisation of the current state of the sentence, according to the criteria discussed in Section 4.3. `OACesec` captures the user gestures relevant to the evolution of the sentence and translates them into an alphabet of actions understood by `automaW`.

The observer `OACesec` is connected to an executor `EACesec` encapsulating `automaW` and `sentenceW`. User actions are inserted in the `Input` pool of `automaW` and consumed by its transitions. In particular,

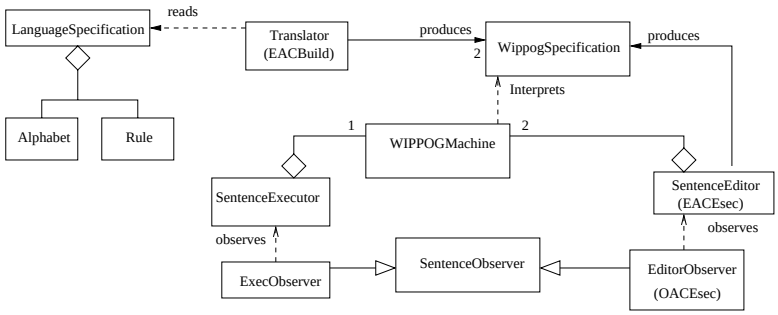

Figure 3: A class diagram defining the GenIAL conceptual architecture

each transition in `automaW` presents exactly one *action* resource in its GETS component, while the current state of the automaton (consumed in WHEN and regenerated in PRODUCES) is described by one *current* resource. Selection actions may be valid or not. In either case, a notification is reflected back to `OACesec`, possibly starting an error dialogue. Visualisation is again managed by `OACesec`, while the control logic is managed in `automaW`. If the action is a request for activation of an enabled rule, then `automaW` exports to `sentenceW` two resources: one specifying the requested rule and one describing the current list of selected elements. The `sentenceW` WM tries to execute the corresponding transition and exports the boolean resource `conditionRule` back to `automaW` to communicate the outcome. According to the value of `conditionRule`, one of two transitions will take place in `automaW`. In either case, the outcome will be notified to `OACesec`, which visualises the new configuration, or starts an error dialogue.

The visual language rules can in turn be annotated with WIPPOG rules defining the behaviour of the transition. This way, the creation of transitions in a visual sentence is associated with the creation of a WIPPOG specification connected with the visual sentence created. The latter can be executed in a new WM (called a `SentenceExecutor`) and its behaviour observed by an observer managing suitable graphical representations of the resources in the sentence.

5 Conclusions

The paper has presented the executable language WIPPOG for specification of discrete event systems, and shown how to connect its execution environment to a visual environment in which a user can specify processes based on some notion of transition. The WIPPOG language can have several applications in the visual language area. A generic system for generation and use of domain specific visual languages can support different visual languages in a uniform way, without having to attach different execution mechanisms for any language. Moreover, a syntax-directed editor can be generated, which allows the creation of sentences in a visual language specified through *before-after* rules. The editor results from the coupling of two WIPPOG specifications, one for an automaton enabling or disabling user actions, and one for the actual rules for sentence evolution.

References

[ACP93] M. Andreoli, P. Ciancarini, and R. Pareschi., *Interaction abstract machines*, Research Directions in Concurrent Object Oriented Programming (G. Agha, A. Yonezaw, and P. Wegner, eds.), MIT Press, 1993, pp. 257–280.

[And92] J.-M. Andreoli, *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), no. 3, 297–347.

[And95] J-M. Andreoli, *Programming in forumtalk*, Tech. Report CT– 003, Rank Xerox Research Centre, Grenoble Lab., France, 1995.

[AP91] J.-M. Andreoli and R. Pareschi, *Communication as Fair Distribution of Knowledge*, Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications, 1991, pp. 212–229.

[Bar02] R. Bardohl, *Genged - a visual environment for visual languages*, Int. Journal Science of Computer Programming **44** (2002), no. 2, 181–203.

[BBMP93] N. Bianchi, P. Bottoni, P. Mussio, and M. Protti, *Cooperative visual environments for the design of effective visual systems*, Journal of Visual Languages and Computing **4** (1993), no. 4, 357–381.

[BCM99] P. Bottoni, M. F. Costabile, and P. Mussio, *Specification and dialogue control of visual interaction through visual rewriting systems*, ACM TOPLAS **21** (1999), no. 6, 1077–1136.

[BMST99] R. Bardohl, M. Minas, A. Schurr, and G. Taentzer, *Application of graph transformation to visual languages*, Handbook of Graph Grammars and Computing by Graph Transformation. Volume II: Applications, Languages and Tools (H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds.), World Scientific, 1999, pp. 105–180.

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, and A Schrr, *Application of graph transformation to visual languages*, Handbook on Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools,, World Scientific, 1999.

[CGP89] P.T. Cox, F. R. Giles, and T. Pietrzykowski, *Prograph: A step towards liberating programming from textual conditioning*, Proc. IEEE Workshop on Visual Languages, IEEE CS Press, 1989, Also appearing in Visual Object-Oriented Programming: Concepts and Environments, M.M. Burnett, A. Goldberg, T.G. Lewis (Eds), Prentice-Hall (1995), pp.45-66, pp. 150–156.

[dLV02] J. de Lara and H. L. Vangheluwe, AToM^3*: A tool for multi-formalism and meta-modelling.*, European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE), LNCS, no. 2306, Springer-Verlag, 2002, pp. 174–188.

[EJ01] R. Esser and J. W. Janneck, *Moses - a tool suite for visual modelling of discrete-event systems*, Symposium on Visual/Multimedia Approaches to Programming and Software Engineering, HCC01, 2001, pp. 272–279.

[Erw98] M. Erwig, *Abstract syntax and semantics of visual languages*, Journal of Visual Languages and Computing **9** (1998), no. 5, 461–483.

[Fur91] G. W. Furnas, *New graphical reasoning models for understanding graphical interfaces*, Proc. CHI '91, April 1991, pp. 71–78.

[Gir95] J.-Y. Girard, *Linear logic: Its syntax and semantics*, Advances in Linear Logic (J.-Y. Girard, Y. Lafont, and L. Regnier, eds.), Cambridge University Press, 1995, pp. 1–42.

[PG02] P.Bottoni and G.Costagliola, *On the definition of visual languages and their editors*, Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002 (M. Hegarty, Bernd Meyer, and N. H. Narayanan, eds.), LNAI, no. 2317, Springer-Verlag, 2002, p. 305319.

[Roz97] G. Rozenberg (ed.), *Handbook of graph grammars and computing by graph transformation, volume 1: Foundations*, World Scientific, 1997.

[RS95] A. Repenning and T. Sumner, *Agentsheets: A medium for creating domain-oriented visual languages*, IEEE Computer **28** (1995), no. 3, 17–26.

[SCS94] D. Smith, A. Cypher, and J. Spohrer., *Kidsim: Programming agents without a programming language*, Communications of the ACM **37** (1994), no. 7, 54–67.

[Tae00] G. Taentzer, *Agg: A tool enviroment for algebraic graph transformation*, Application of Graph Transformation with Industrial Relevance (M. Nagl, M. Mnch, and A. Schuerr, eds.), LNCS 1779, Berlin: Springer-Verlag, 2000.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G Kim, *Theory of modeling and simulation, 2nd edition*, Academic Press, 2000.