View Transformation in Visual Environments applied to Algebraic High-Level Nets

Claudia Ermel [1], Karsten Ehrig [2],

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

Abstract

Graph transformation systems are a well-founded and adequate technique to describe the syntax of visual modeling languages and to formalize their semantics. Moreover, graph transformation tools support visual model specification, simulation and analysis on the basis of the rich underlying theory.

Despite the benefits of model validation by simulation, sometimes it is preferable for users to see the model's behavior not in the abstract layout of the formal model, but as scenarios presented in the layout of the specific application domain. Hence, we propose the integration of a domain-oriented animation view with the model transformation system. An animation view allows to define scenario animations in a systematic way based on the formal model. The specification of the well-known *Dining Philosophers* system as algebraic high-level Petri net serves as running example for the extension of the model by an animation view and the derivation of animation rules from the model transformation system. A scenario animation then is obtained as transformation by applying the animation rules to model states. This visualizes the behavior of the model in the layout of philosophers sitting around a table and eating with chopsticks. A prototypical implementation of the concepts in GenGED, a visual language environment, is presented.

Key words: graph transformation, Petri nets, animation view

1 Introduction

During the last decades the growing complexity of software systems led to a shift of paradigm in software specification from textual to visual modeling techniques which are used to represent aspects like e.g. distribution, components, parallelism and processes in a more natural way.

[1] Email: lieske@cs.tu-berlin.de
[2] Email: karstene@cs.tu-berlin.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The success of visual modeling techniques resulted in a variety of methods and notations addressing different application domains, perspectives and different phases of the software development process. Common visual notations like e.g. the UML [19] are often semi-formal in the sense that the syntax and semantics of the models are defined informally with different, sometimes even incompatible interpretations. The use of graph transformation techniques provides support to improve the preciseness of visual modeling techniques.

With graph transformation systems the concrete and abstract syntax of various visual modeling languages can be described, and the semantics can be formalized. Moreover, graph transformation tools [5] such as AGG [18,1], GenGED [2,11] or DiaGen [14] support visual model specification, simulation and analysis on the basis of the rich underlying theory. By *simulation* we mean to show the before- and after-states of an action as diagrams of the visual language used to define the formal model. *Scenarios* then are given as sequences of actions, where the after-state of one action is the before-state of the next action. In Petri nets, for example, a simulation is performed by playing the token game. Different system states are different markings, and a scenario is determined by a firing sequence of transitions resulting in a sequence of markings. Using graph transformation, simulation is specified by a behavior grammar, and a scenario corresponds to a derivation sequence where the behavior rules are applied to a start diagram (given as graph), and, consequently, to the derived diagrams.

Despite these benefits, often the simulation of abstract visual behavior models (e.g. Petri nets, graphs or statecharts) is not flexible enough and, hence, can be ineffective in the model validation process. The behavior of a model based on (semi-) formal and abstract notations may not always be comprehensible to users, due to several reasons, like e.g.: different aspects like control flow and data flow are represented in a similar way (a common problem in understanding Petri nets); information belonging to one model is distributed in several submodels based on different visual modeling languages (this can also lead to inconsistencies between the submodels); too much detail is integrated in the model representation (e.g. in order to be able to perform a complete formal analysis of the model); if the chosen modeling formalism does not allow to model a distinguished feature by an adequate model element, auxiliary elements (such as stereotypes in UML) or workarounds are used which make it difficult for other people than the modelers themselves to understand what is meant.

To this end, instead of simulating the model behavior, animation has become a popular way to present a model's behavior. *Animation* shows model aspects in a layout visually different from the formal model. Actions are visualized in a movie-like fashion, such that not only discrete steps (a before-state and an after-state) are shown but rather a continuously animated change of the scene. Unfortunately, the step from the formal behavior specification to the animation may introduce new errors or inconsistencies in addition to those

in the model which we want to discover by the validation.

Therefore, this paper proposes the use of formal *views* and *view transformation* on the basis of graph transformation for well-founded model simulation and animation in order to facilitate the validation of model behavior in visual environments. We extend the concepts of typed, attributed graph transformation by means for view integration and view restriction in order to to define an adequate abstraction level for the simulation of visual behavior models. Moreover, additional views for the animation of model behavior are added in a systematic way to the model such that on the one hand customers can easily understand and validate the model behavior, and on the other hand, the additional views do not change the semantics of the modeled system and can easily be replaced by other views. The definition of these so-called *animation views* and the animation of specific model behavior scenarios supports the early detection of inconsistencies and possible missing requirements in the model which cannot always be found by formal analysis only.

An animation view presents a state of the model in the layout of a specific application domain. We call the simulation steps of a model *animation steps* when the states (diagrams) before and after a simulation step are shown in the animation view. By the transformation of a behavior model to an animation view, the view might show only some aspects or parts of the system. Different views can reflect (parts of) the behavior defined in the behavior model. Fig. 1 sketches the relation between formal behavior models and animation views in different application domains. Formal behavior models (e.g. Petri

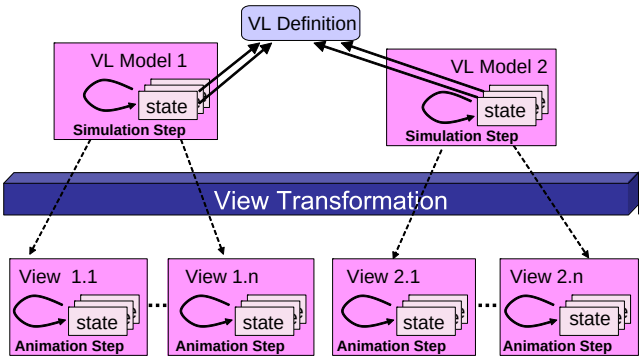


Fig. 1. Different views on formal models

nets or state charts) are elements of a visual language, i.e. diagrams over the corresponding VL. Such a language is given in the graph transformation representation as a combination of a VL type graph and a VL syntax grammar defining the valid diagrams for the VL. The behavior of a model is defined as a graph transformation system where the start graph corresponds to the initial state and the behavior rules specify the valid state transitions (simulation steps). We call the set of valid model states induced by the behavior rules *VL model*. View transformation is applied to the complete VL model on the basis of a *view transformation system*, thus realizing a consistent mapping of simu-

lation steps to animation steps in the respective animation view. Moreover, by adding continuous animation operations to the animation steps the resulting scenario animations are not only discrete-event steps but can show the model behavior in a movie-like fashion. Consequently, requirements and scenarios can be interactively played out and validated in one or more animation views.

The paper is organized as follows. In Section 2, the running example is introduced, the well-known model of *The Dining Philosophers* modeled as algebraic high-level net (AHL net). The formal definition of AHL nets is reviewed. In Section 3 we define the graph transformation system for the *Dining Philosophers* net and give a general construction for mapping the behavior of AHL nets to graph transformation systems. Section 4 formally defines views and their interaction and integration. Moreover, we introduce animation views [10] as a special kind of views and apply the concept of view transformation to realize a consistent mapping from the behavior model to an animation view. This is illustrated by the definition of an animation view for the *Dining Philosophers* net. The main implementation issues of the visual environment GenGED are presented in Section 5 with a focus on the definition of animation viewss. Finally, in Section 6, we summarize the main achievements and outline some open problems and directions for future work.

2 *The Dining Philosophers* modeled as AHL Net

An AHL net is a specific, well-defined high-level net variant, a combination of a place/transition net [17] and an algebraic datatype specification *SPEC*. Arc inscriptions ar *SPEC*-terms and tokens are elements of a corresponding *SPEC*-algebra [7]. In this section, we review the definition of AHL nets and their behavior as given in [15], and present our running example, the specification of the well-known *Dining Philosophers* as AHL net. The pre- and postdomain of a transition are given by a multiset of pairs of terms and places, i. e. as elements of a commutative monoid.

Definition 2.1 (Algebraic High-Level Net)
An *algebraic high-level net* $N = (SPEC, P, T, pre, post, cond)$ consists of an algebraic specification $SPEC = (S, OP, E; X)$ with equations E and additional variables X over the signature (S, OP), sets P and T of places and transitions respectively, pre- and postdomain functions $pre, post : T \to (T_{OP}(X) \times P)^{\oplus}$ assigning to each transition $t \in T$ the pre- and postdomains $pre(t)$ and $post(t)$ respectively and a firing condition function $cond : T \to \mathcal{P}_{fin}(EQNS(S, OP, X))$ assigning to each transition $t \in T$ a finite set $cond(t)$ of equations over the signature (S,OP) with variables X.

Remarks

- $T_{OP}(X)$ is the set of terms with variables X over the signature (S, OP). $T_{OP}(X) \times P$ is defined by $T_{OP}(X) \times P = \{(term, p) | term \in T_{OP}(X), p \in P\}$.

- The pre- and postdomain functions $pre(t)$ (and similar $post(t)$) have the form $pre(t) = \sum_{i=1}^{n}(term_i, p_i) \quad (n \geq 0)$ with $p_i \in P, term_i \in T_{OP}(X)$. This means that $\{p_1, ...p_n\}$ is the predomain of t with arc-inscription $term_i$ for the arc from p_i to t if $\{p_1, ..., p_n\}$ are disjoint (unary case) and arc-inscription $term_{i1} \oplus ... \oplus term_{ik}$ for $p_{i1} = ... = p_{ik}$ (multi case). In our sample AHL net (see Example 2.3) we have the multi case.

Definition 2.2 (Marking and Firing Behavior of AHL Nets)
Let $N = (SPEC, P, T, pre, post, cond)$ be an AHL net according to Def. 2.1, and let A be a (S, OP, E)-algebra.

- A marking m is an element $m \in (A \times P)^{\oplus}$.
- Enabling and firing of transitions is defined as follows: let $Var(t)$ be the set of local variables occurring in $pre(t), post(t)$ and $cond(t)$. An assignment $asg_A : Var(t) \to A$ is called consistent for $t \in T$ if the equations $cond(t)$ are satisfied in A under asg_A. A transition $t \in T$ is enabled under a consistent assignment $asg_A : Var(t) \to A$ and a marking $m \in (A \times P)^{\oplus}$, if $pre_A(t, asg_A) \leq m$.

  The marking $pre_A(t, asg_A)$ – analogously $post_A(t, asg_A)$ – is defined for $pre(t) = \sum_{i=1}^{n}(term_i, p_i)$ by $pre_A(t, asg_A) = \sum_{i=1}^{n}(\overline{asg}_A(term_i), p_i)$, where $\overline{asg}_A : T_{OP}(Var(t)) \to A$ is the extended evaluation of terms under assignment asg_A. The successor marking m' is defined in the case of t being enabled by $m' = m \ominus pre_A(t, asg_A) \oplus post_A(t, asg_A)$ and gives raise to a *firing step* $m[t, asg_A\rangle m'$.

Example 2.3 (The Dining Philosophers as AHL Net)
As example we show the AHL net for *The Dining Philosophers* in Fig. 2 (see [17,15] for the corresponding Place/Transition net). We identify five philosophers as well as their chopsticks by numbers. Fig. 2 (a) shows the initial situation where all philosophers are thinking and all chopsticks are lying on the table. Fig. 2 (b) shows the AHL net with corresponding initial marking (all philosopher numbers are on place thinking, all chopstick numbers on place table). In Fig. 2 (b), the transition take is enabled as a thinking philosopher and his left and right hand side chopsticks are available. The firing of transition take with the variable binding $p = 2$, for example, results in the AHL net with the same net structure as in Fig. 2 (b), but with a different marking: token 2 is removed from place thinking and added to place eating, and tokens 2 and 3 are removed from place table, as the chopstick computing operation (p mod 5) +1 is evaluated to 3.

As datatype specification we have the specification NAT for natural numbers. The tokens on all places are elements of a corresponding NAT-algebra [7], i.e. natural numbers. The arcs are inscribed each by one or more variables or terms from $T_{OP}(X)$ denoting computation operations to be executed on token values if the transition fires.

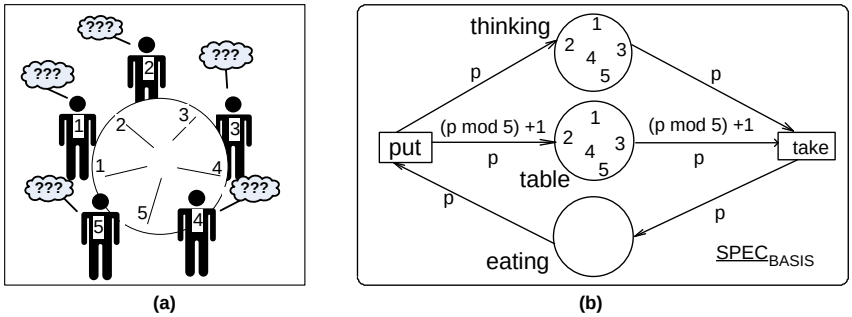


Fig. 2. The *Dining Philosophers* (a) modeled as AHL Net (b)

3 AHL Nets as Graph Transformation Systems

In this section, we review some basic concepts of the graph-transformation based approach for the generic description of visual languages (VLs) and show how the *Dining Philosophers* example can be presented in this approach.

3.1 The Visual Language of AHL Nets

The generic description of a VL using graph transformation results in a VL specification consisting of a visual alphabet (a type graph) and a visual syntax grammar. VL sentences are graphs typed over the type graph and can be derived by applying the syntax grammar rules. Two levels of descriptions are considered, called *abstract syntax* and *concrete syntax*. The abstract syntax level describes the symbols and links used in the VL, whereas the concrete syntax level describes their layout. The abstract syntax of a visual alphabet is defined by an attributed graph structure signature, whereas the concrete syntax extends this signature according to the layout of symbols, specified by graphical operation symbols (cf. [3]) and by a graphic constraint statisfaction problem on positions and sizes of the used graphics.

In the following, we present attributed graph structures as defined in [8].

Definition 3.1 (Attributed Graph Structure Signatures) A graph structure signature $GSIG = (S_G, OP_G)$ is an algebraic signature with unary operations $op : s \to s'$ in OP_G only. An attributed graph structure signature $ASSIG = (GSIG, DSIG)$ consists of a graph structure signature $GSIG$ and a data signature $DSIG = (S_D, OP_D)$ with attribute value sorts $S'_D \subseteq S_D$ such that $S'_D = S_D \cap S_G$ and $OP_D \cap OP_G = \varnothing$.
$ASSIG$ is called *well-structured* if for each $op : s \to s'$ in OP_G we have $s \notin S_D$.

$ASSIG$-algebras and $ASSIG$-homomorphisms build a category [8] which is denoted by **ASSIG-Alg**. In the following, we call $ASSIG$-algebras *attributed graphs* and $ASSIG$-homomorphisms *attributed graph morphisms*.

Now we can define the attributed graph structure signature ASSIG_{AHL} for AHL nets. AHL nets are considered as ASSIG_{AHL}-algebras.

Definition 3.2 (Visual Alphabet for AHL Nets)
The visual alphabet for AHL nets (shown visually in Fig. 3) is given by the attributed graph structure signature $\mathsf{ASSIG}_{AHL} = (GSIG_{AHL}, DSIG_{AHL})$. In Fig. 3, the sorts of $GSIG_{AHL}$ are represented as nodes. The operations are the arcs between the sort nodes (the *op*-links between graph sorts), from sort nodes to data nodes, (the *attr*-links between graph sorts and attribute sorts) and the arcs connecting the abstract syntax sort nodes and the concrete syntax sort nodes (the *graphic*-links). The $DSIG$ part (data signature) consists of the attribute value sorts of the basic specification, i.e. *Nat* and *Bool* and their usual operations. The attribute values are used for the arc inscriptions, tokens and transition firing conditions.

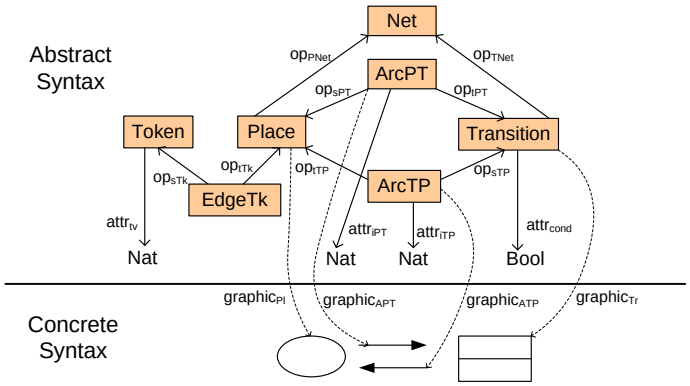


Fig. 3. Type Graph visualizing the ASSIG for AHL Nets

Visual sentences of a specific VL are **ASSIG**-algebras (attributed graph structures) where the layout constraints of the corresponding Csp are satisfied. For the AHL net alphabet, the constraints define for example that the token number is drawn within the place figure, that an arc inscription is positioned near the center of the corresponding arc, and that a firing condition is written in the lower part of the transition rectangle.

In general, a *VL syntax grammar* (S, P) consisting of an empty start graph S and a set P of language defining rules, defines the allowed editing operations and restricts the set of visual sentences to the meaningful ones. The VL syntax grammar together with the corresponding VL alphabet define the visual language $VL = \{VLS \mid S \overset{*}{\Longrightarrow}_P VLS\}$, where VLS is the set of all VL sentences derivable from the start sentence S of the VL syntax grammar with the VL syntax rules R.

3.2 Modeling the Behavior of AHL Nets by Graph Rules

In this section we focus on simulating the dynamic behavior of visual models based on a visual language (VL models). Formally, this is done by defining a suitable graph transformation system in **ASSIG-Alg**.

Therefore, we first define the double-pushout approach to graph transformation on the basis of category **ASSIG-Alg**.

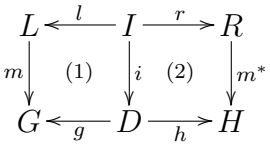
Proposition 3.3 (Pushouts of $ASSIG$-Homomorphisms) *Let M be a distinguished class of all homomorphisms f which is defined by $f \in M$ if f_{GSIG} is injective and $f_{DSIG} = id_{DSIG}$ for f in* **ASSIG-Alg**. *Given $f : A \to B \in M$ and $a : A \to C$ then there exists their pushout in* **ASSIG-Alg**.

Proof: See [8].
Category **ASSIG-Alg** and class M are fixed throughout this section.

Definition 3.4 (Typed Attributed Graph Transformation System) A typed attributed graph transformation system $GTS = (S, P)$ based on (***ASSIG-Alg***, M) consists of an $ASSIG$-algebra S, called start graph and a set P of rules, where

(i) a rule $p = (L \xleftarrow{l} I \xrightarrow{r} R)$ of $ASSIG$-algebras L, I and R attributed over the term algebra $T_{DSIG}(X)$ with variable set X of variables $(X_s)_{s \in S_{DSIG}}$, called left-hand side L, interface I and right-hand side R, and homomorphisms $l, r \in M$, i.e. l and r are injective and identities on the data type $T_{DSIG}(X)$,

(ii) a *direct transformation* $G \overset{p,m}{\Longrightarrow} H$ via a rule p and a homomorphism $L \xrightarrow{m} G$, called *match*, is given by the diagram to the right, called *double-pushout diagram*, where (1) and (2) are pushouts in ***ASSIG-Alg***,



(iii) a typed attributed graph transformation, short transformation, is a sequence $G_0 \Rightarrow G_1 \Rightarrow ... \Rightarrow G_n$ of direct transformations, written $G_0 \overset{*}{\Rightarrow} G_n$,

(iv) the language $L(GTS)$ is defined by $L(GTS) = \{G \mid S \overset{*}{\Rightarrow} G\}$.

This leads to the following definition of a VL model:

Definition 3.5 (VL Model) Let VL be a visual modeling language used for the formal specification of behavior models, given by the VL alphabet TG. Then, a *VL model* is a subclass of VL sentences modeling all possible states of one specific behavior model, given by the typed, attributed graph transformation system $M = (TG, S, P)$, where S is the initial state (a VL sentence) and P is a set of graph rules, called behavior rules. The VL model states are given by the language $ML \subseteq VL$ defined by the VL model: $ML = \{D | S \overset{*}{\Longrightarrow} D\}$.

For each VL behavior rule $L \xrightarrow{r} R$, L contains the subpart of the state relevant for the state transition to be considered, and R models the update of this subpart. Thus, a VL behavior rule represents the change caused by a state transition. For example, a certain Petri net is a VL model according to Def. 3.5 with respect to the visual Petri net language: The VL model is the set of all sentences over the Petri net language with the same net structure i.e. one fixed net but different markings. The markings are given by an initial

marking and all reachable markings in the given net, which is expressed by the behavior rules for Petri nets.

This "classical" approach to translate Petri nets to graph transformation systems has its roots in the works of Kreowski [12], Parisi-Presicce et al. [16] and Corradini et al. [6]. In the case of high-level Petri nets, multiple and individual tokens can be represented by using attributed graph grammars where tokens in high-level nets can be data of arbitrary algebraic data types.

In the following, we show that the token game of a Petri net can be given in terms of the behavior of a VL model according to Def. 3.5. We construct behavior rules which correspond to firing the transitions of the net. More precisely, we have to ensure that a transition in the net is enabled if and only if the corresponding rule is applicable to the visual sentence corresponding to the net and that firing a transition in the net corresponds to a derivation step in the grammar and vice versa. The token game then can be simulated by applying the behavior rules to a VL sentence modeling a marked Petri net (a VL model state). For each Petri net behavior rule $L \xrightarrow{r} R$, L defines the predomain of a certain transition, and R defines corresponds to its postdomain. Thus, r removes the marking from the transition's predomain and adds the required marking to the places in its postdomain. This approach to model Petri net behavior can be applied to various types of low-level and high-level Petri nets.

For AHL nets, one behavior rule is defined for each transition in the net. Variables and operations from the algebraic specification are used in the rules. The formal relationship between AHL nets and attributed graph grammars is presented in [4]. There we give a proof of the semantical compatibility of AHL nets and their representation as graph transformation system based on the formal semantics of AHL net behavior (as given in Def. 2.2) and the construction of graph derivations as pushouts in the category **ASSIG-Alg** of attributed graphs and graph morphisms.

Example 3.6 (VL Model for the *Dining Philosophers*)
Based on our VL for AHL nets, the VL model for the *Dining Philosophers* comprises all those VL sentences containing the places thinking, table and eating as well as the transitions take and put and the arcs with term inscriptions as depicted in Fig. 2 (b). As initial marking we assume all philosopher data elements (1, ..., 5) on place thinking, all chopsticks (1, ..., 5) on place table and no tokens on place eating. As our net contains two transitions, we have two behavior rules for the transitions put and take realizing the transformations of an eating philosopher to a thinking philosopher and back (see Fig. 4). The effect of rule put is that the philosopher puts his two chopsticks down onto the table. Rule take is the reversed rule of put. We use a variable for the philosopher token (p). The data values for the two chopsticks p and (p mod 5) + 1 are computed by matching p to a number and by computing the value of (p mod 5) + 1 according to the current binding of the variable p.

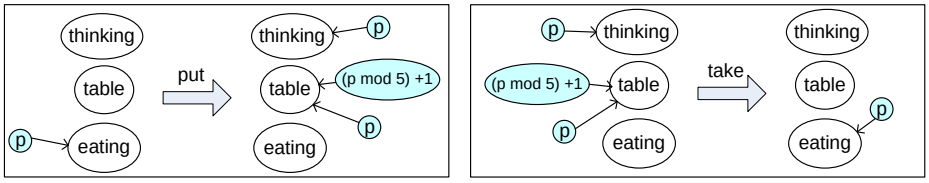


Fig. 4. Behavior rules for the AHL net model *Dining Philosophers*

Note that it is possible to generate behavior rules for arbitrary AHL nets automatically according to the general definition of firing transitions in AHL nets. The algorithm for generating behavior rules is given in Def. 3.7.

Definition 3.7 (Translation of AHL Net Transitions to Graph Rules) Each transition $t \in T$ is translated to an attributed graph rule $r_t : L_t \to R_t$. The attributed graphs in L_t and R_t of such a rule are $\mathbf{ASSIG}_{AHL}$-algebras. Both contain Place nodes for all places in the pre- and postdomain of t. In L_t [R_t], the places p_i in the predomain [postdomain] are marked according to the following algorithm:

for each arc $a : p_i \to t$ [$a : t \to p_i$]
  for each arc inscription term $tm \in inscr(a)$
    generate a token node of type Token attributed by a copy tk of tm;
    connect the token node by an arc of type EdgeTk to place p_i;

It is shown formally in [4] that this translation preserves the semantics, i.e. that for each firing sequence in the AHL net there is a unique transformation in the translated graph transformation system such that the resulting graph corresponds to the marking of the AHL net.

The behavior rules are the basis for animation introduced in Section 4.

4 Animation Views for AHL Nets

To bridge the gap between the underlying descriptive specification of a process (e.g. as Petri net) and a natural dynamic visual representation of processes being simulated, we suggest the definition of an *animation view* for a VL model. On the one hand, this animation view must be easy to comprehend; people who are non-specialists in the underlying formal process modeling technique (e.g. Petri nets) should be able to observe (interesting parts of) the functional behavior of the model. On the other hand, the behavior shown in the animation view has to correspond to the behavior defined in the formal model. Hence, in this section we propose a graph transformation based view translation for a VL model from its formal specification to an animation view. Thus, at first we give some general definitions concerning views on VL models, and then define animation views as a special case of a view.

4.1 Views for Behavior Models

Fig. 5 shows some aspects of the characterization of *views* in UML [19]. Different stakeholders are to be seen who look at different (sets of) diagrams where each diagram contains information about a subset of elements from the same underlying model (depicted here as a set of model elements). From this informal characterization of models and views, we intend to reflect the following features in our formalization:

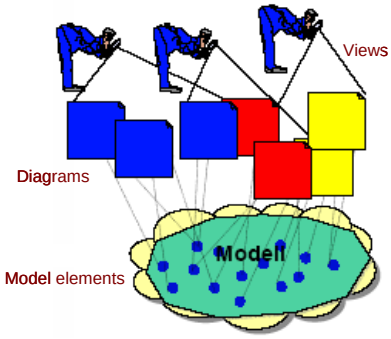


Fig. 5. Relation of Model and Views

- The basic system model is a VL model (see Def. 3.5), i.e. a typed graph transformation system $M = (TG, S, P)$.
- A *view* is an incomplete specification of a system, focusing on a particular aspect or subsystem. Hence, in our formalization, a view is a VL model which is a part of a larger VL model. This *part of* relation is captured in the formal definition of views (Def. 4.1) by a type graph morphism from the type graph of the view TG_V to the type graph of the larger VL model TG. We define views at the level of type graphs for visual languages to emphasize the fact that a view usually is presented using an adequate type of diagrams, i.e. a special VL. Note that the recursive way to define views allows us to have views of other views. The behavior of a view is given by the restricted graph transformation system M to the type graph of the view TG_V (where the rules of the view are subrules of the rules of M).
- Different views of the same VL model can be related to each other. This relation is expressed formally in the definition of *interaction* of views (Def. 4.3).
- Two different views of a VL model can be composed to one common view by gluing their common parts. This is called *integration* of views (Def. 4.4).

Definition 4.1 (View / Restriction)

Let $M = (TG, S, P)$ be a VL model (see Def. 3.5) in VL. Then the pair (V, v) with the VL model $V = (TG_V, S_V, P_V)$ and the embedding $v : V \to M$ is called *view of* M or *restriction of* M *to* TG_V, written $V = M|_{TG_V}$. The embedding $v = (t_V : TG_V \to TG, s : S_V \to S, f_P : P_V \to P)$ is called *view embedding*. $t_V : TG_V \to TG$ is the type graph inclusion and $s : S_V \to S$ is graph restriction to TG_V, written $S_V = S|_{TG_V}$, where the diagram to the right is a pullback.

$$\begin{array}{ccc} S_V & \longrightarrow & TG_V \\ {\scriptstyle g_1}\downarrow & (PB) & \downarrow{\scriptstyle t_1} \\ S & \xrightarrow[m]{} & TG \end{array}$$

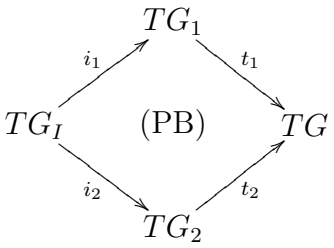
For each $p \in P$ there exists a subrule $p_V \in P_V$ such that $f_P(p_V) = p$ and $se_V : p_V \to p$ is the subrule embedding induced by t_V.

Obviously, view embeddings may be composed. This allows to regard a view (V_0, v_0) of another view (V_1, v_1) of M as an extended view $(V_0, v_0; v_1)$ of M.

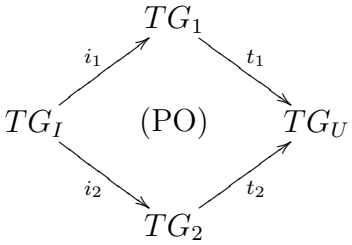
Definition 4.2 (Extension of a View)
Let $(V_0, v_0 : V_0 \to V_1)$ be a view of V_1 and $(V_1, v_1 : V_1 \to V_2)$ be a view of V_2. Then $(V_0, v_0; v1 : V_0 \to V_2)$ is also view of V_2 with $v_0; v_1$ being the composition of view embeddings constructed by componentwise inclusion and subrule embedding: $v_0; v_1 = (t_1 \circ t_0, f_{S_1} \circ f_{S_0}, f_{P_1} \circ f_{P_0})$. We call the view $(V_0, v_0; v_1 : V_0 \to V_2)$ *extension of view* $(V_0, v_0 : V_0 \to V_1)$.

Additionally, we define constructions for the interaction between two views and for the integration of different views into one view. Based on these constructions, we give a formalization for the consistency of views depending on their view interaction.

Definition 4.3 (Interaction of Views)
Let $(V_1, V_1 \xrightarrow{v_1} M)$ and $(V_2, V_2 \xrightarrow{v_2} M)$ be two different views of $M = (TG, S, P)$ with $V_1 = (TG_1, S_1, P_1)$ and $V_2 = (TG_2, S_2, P_2)$. The interaction of views is defined as VL model $I = (TG_I, S_I, P_I)$ given by the two common views $(I, I \xrightarrow{i1} V_1)$ and $(I, I \xrightarrow{i1} V_2)$ in the pullback to the right (called interaction pullback).



Definition 4.4 (Integration of Views)
Let $(V_1, V_1 \xrightarrow{v_1} M)$ and $(V_2, V_2 \xrightarrow{v_2} M)$ be two different views of $M = (TG, S, P)$ with interaction $I = (TG_I, S_I, P_I)$. The *integration* of V_1 and V_2 is the VL model $U = (TG_U, S_U, P_U)$ where the diagram to the right is a pushout (called integration pushout). The rule set P_U consists of the set of amalgamated rules $p_U = p_1 \oplus p_2$ over the subrule embeddings $p_1 \leftarrow p_I \rightarrow p_2$ induced by the interaction I.



4.2 Defining Animation Views for VL Models

In order to represent the behavior of a VL model directly in a domain-oriented layout, the system states are mapped onto graphical representations for real-world objects and values (modeled by the type graph of the *animation view*).

The nature of animation differs considerably from the notion of *simulation* as modeled so far. *Simulation* visualizes state changes within the means of the VL model itself. The simulator sees a Statechart or a Petri net, where simulation steps are carried out by switching to another marking (of a Petri net) or by highlighting another state (in a Statechart). Moreover, simulation relies on discrete steps and cannot depict continuous changes (e.g. there is no state between a marking of a Petri net and the successor marking after a transition has fired). Hence, we model animation actions by graph rules in the layout of the animation view, and enhance these animation rules by attributes for continuous changes of objects such as motions or changes of size or color.

Both the formal model $F = (TG, S, P)$ and its animation view are different

views of the same integrated VL model M. This integrated VL model M is constructed by adding the symbols and links needed for the animation view to the original type graph TG, to the start graph S and to the behavior rules in P. We specify this construction by graph rules called *view transformation rules*. On the basis of these view transformation rules it is possible to enforce coherence between the behavior rules of the original VL model and the animation rules of the animation view.

Definition 4.5 (Animation View of a VL Model)
Let TG_A be the alphabet of an animation domain. Let $F = (TG_F, S_F, P_F)$ be a VL model over a formal behavior specification language VL. Let $M = (TG, S, P)$ be a VL model with the integrated type graph $TG = TG_F \cup TG_A$, where TG is constructed as pushout in $\mathbf{ASSIG}_{AHL}$ over a common interface. Let VGT be a set of rules (called view transformation rules), typed over TG. The start graph S of M is derived from S_F by applying the rules of VGT to S_F in a given order, where each rule is applied as often as possible, before the next rule is applied. The rules in P are called *animation rules*. For each animation rule $L \stackrel{r}{\rightarrow} R \in P$, L $[R]$ is derived from L_F $[R_F]$ of the corresponding behavior rule $L_F \stackrel{r_F}{\rightarrow} R_F \in P_F$, by applying the rules of VGT (analogously to the derivation of S) to the graphs in the rule sides.

Then an *animation view* of M is defined as view $V = M|_{TG_A}$.

We suggest the following guidelines for the definition of view transformation rules to be applied to VL sentences over a Petri net alphabet:

- The *animation context* contains the part of the view which is not changed by animation and where all animated symbols should be linked to. This corresponds to the Net symbol of the Petri net alphabet. Therefore, the Net symbol should be linked to the top-level symbol of the animation context.
- The *animated part* of the animation view consists of icons which are changed (moved) during animation. These icons directly correspond to the tokens of the Petri net. The layout and position of the icons depends on the place where the token is lying in the active state. Therefore, the symbols for the animated part should be linked to tokens depending on their places.
- Transitions and their adjacent arcs denoting their pre- and postdomains correspond to animation rules applicable to specific states of the model in the animation view. They are not linked to symbols from TG_A.

Example 4.6 Animation View for *The Dining Philosophers*

Fig. 6 shows the integrated alphabet for the *Dining Philosophers* animation view where the AHL net alphabet (lower part of Fig. 6) is united with an animation domain alphabet (upper part of Fig. 6). The animation context consists of a round table with numbered plates on top. The animated part are the thinking and eating philosophers positioned around the table, and the chopsticks besides the plates. The position of a philosopher is defined in our

animation domain alphabet by a constraint relating the philosopher's icon's position to the position of her (numbered) plate icon. Constraints ensure as well that plates are ordered in a circle on the table.

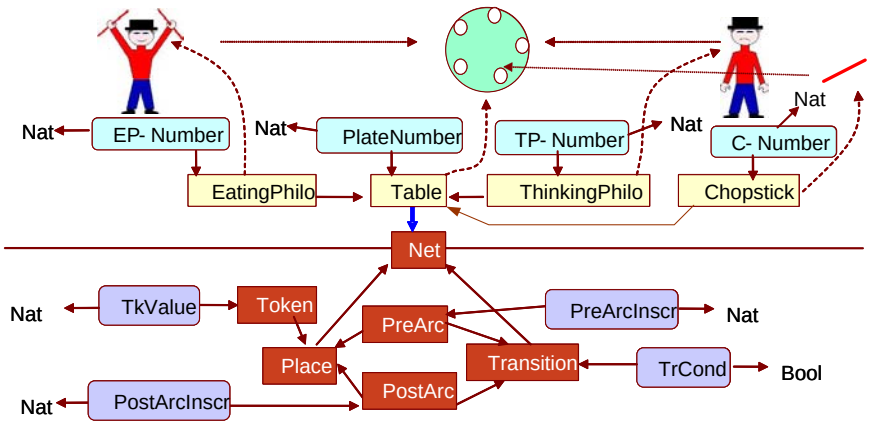


Fig. 6. Integrated alphabet for *The Dining Philosophers*

Fig. 7 shows the view transformation rules which are typed over the integrated alphabet in Fig. 6.

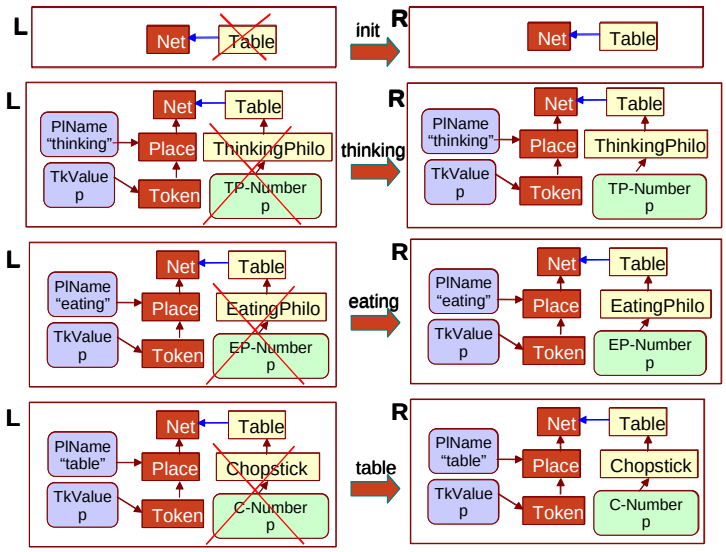


Fig. 7. View transformation rules for *The Dining Philosophers*

One initial rule generates the fixed animation context and links it to the abstract syntax of the Net symbol. Token rules then link each token depending on its place to its new animation symbol (a corresponding icon at a certain position within the fixed animation context). After suitable application of all these view transformation rules, a sentence of an AHL net VL model is transformed into a sentence of the integrated VL, which now also contains the animation view.

Fig. 8 illustrates an animation rule derived by applying the view transformation rules in Fig. 7 to the behavior rule take in Fig. 4.

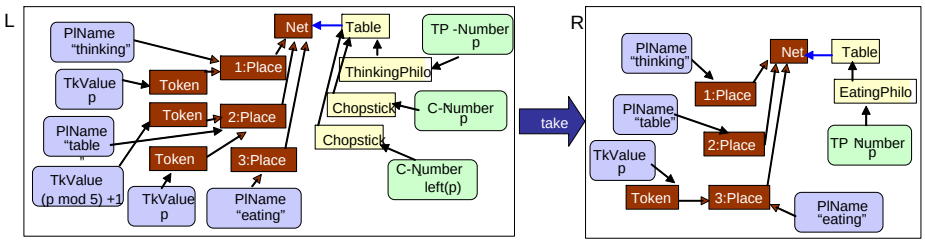


Fig. 8. Derived animation rule for *The Dining Philosophers*

The second animation rule for put is constructed analogously and equals the reversed rule for take. The animation rules now model the behavior of the *Dining Philosophers* according to the AHL net model, but visualized also in the animation view.

5 Implementation of Animation Views in GenGED

Fig. 9 presents the GenGED environment for generic visual language definition and model simulation, now extended by the methodology for animation view definition and scenario animation as proposed in this paper.

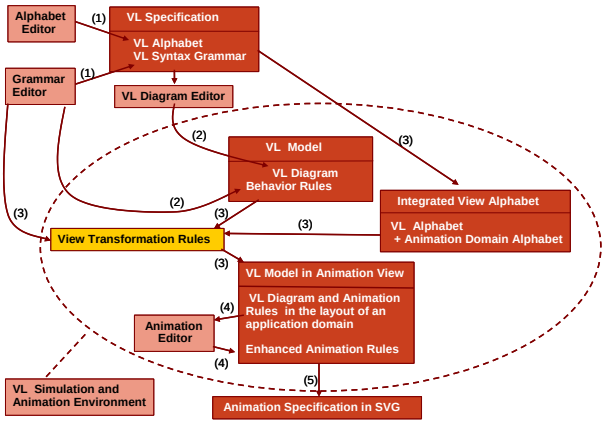


Fig. 9. The GenGED environment extended by features for animation

We explain Fig. 9 by adding to the workflow different roles for users of the GenGED environment (different roles need not necessarily be taken by different persons) and describing who is doing what:

(1) The *language designer* defines the VL Specification by using the Alphabet Editor to define the VL Alphabet and using the Grammar Editor to define the VL Syntax Grammar.

(2) The *model designer* uses the VL Specification to edit a VL Diagram and defines the Behavior Rules using the Grammar Editor. The VL Diagram together with the Behavior Rules specifiy the VL Model.

(3) The *view designer* specifies an Animation View by defining the alphabet for the animation domain and merging it with the VL Alphabet to an integrated View Alphabet. To this end, the Alphabet Editor has been extended

by a *MergeAlphabet* action allowing to integrate two different alphabets. The common symbols and links (identified by equal names) are glued and appear only once in the integrated alphabet. The information about their original alphabet(s) is stored for each symbol of the integrated alphabet. Moreover, the *view designer* defines the View Transformation Rules over the View Alphabet. Applying the View Transformation Rules to the VL Model, he generates a transformed VL Model in the Animation View. To this end, the Grammar Editor has been extended by an *ApplyMetaGrammar* action to apply a meta grammar (containing e.g. the view transformation rules) to a model grammar (e.g. the behavior rules plus start graph). Starting with the first meta rule, each meta rule is applied as often as possible to the LHS and RHS of each model rule. The *view designer* has to take care that the application of the meta rules terminates. In our view transformation rules, a NAC equal to the RHS ensures that each rule can be applied only once at each match.

(4) The *animation designer* uses the Animation Editor [9] to produce Enhanced Animation Rules extending the animation rules by *animation operations* realizing continuous changes of graphics such as moving, or changing the size or the color. The *animation designer* defines these animation operations visually, e.g. by drawing a required on-screen route interactively on the scenario background. Moreover, more than one animation operation can be defined for one rule and the starting time and duration for each animation operation can be specified.

(5) the *model validator* works in the VL Simulation and Animation Environment. He or she simulates (or animates) the behavior of a VL model by applying the behavior or animation rules to the current model state. Single animation steps can be viewed in the animation environment by applying an animation rule to a VL diagram. Animation sequences can be recorded by performing a sequence of animation rule applications. The complete animation then is stored in the XML-based SVG format (Scalable Vector Graphics [20]) and can be viewed by any external SVG viewer tool. In the VL Simulation and Animation Environment the *model validator* can switch between the different views for one model. Thus, the formal model can be shown in the layout of e.g. the AHL net alphabet, or the animation view is activated to show the model behavior in the layout of the application domain. The triggering of the simulation or animation steps (by selecting a rule) is visualized in all selected views at once.

6 Conclusion

In this paper we have extended the generic description of visual languages based on graph transformation systems by the notions VL model, views on a VL model and, especially, animation views of a VL model. A VL model is a visual presentation of the states of a behavior model, where VL is a

visual modeling language used for the formal specification of behavior models. In our running example, we have formally specified the VL model for *The Dining Philosophers* using AHL nets. This VL model can be animated in our approach by integrating the VL alphabet with a freely chosen domain-specific animation alphabet and transforming the VL model states to states typed over the integrated alphabet. This view-transformation based approach ensures that the behavior in the VL model is mapped consistently to the animation view.

On the practical side, the GenGED tool environment [2] has now been extended in order to be able to manage the combination of different views by allowing to merge their alphabets (view integration) in the alphabet editor. Moreover, in the generated environment it is now possible to select a view for the simulation or animation of a VL model.

Future work is planned to cover the animation of still more visual behavior specification languages, e.g. considering selected diagram types from UML. In more complex cases the VL models may lead to large graph transformation systems which are difficult to handle and to understand. Therefore, for practical use, structuring concepts for graph transformation (see e.g. [13]) should be incorporated in the presented approach, and also implemented in the tool AGG [1], which is the underlying graph transformation engine for GenGED. Work is in progress to implement type graphs with inheritance and multiplicities as underlying language model in AGG, which should make it easier to go the step from a meta model description (e.g. a UML class diagram) to the corresponding type graph for a UML based visual language.

References

[1] *AGG Homepage*, http://tfs.cs.tu-berlin.de/agg.

[2] Bardohl, R., *A Visual Environment for Visual Languages*, Science of Computer Programming (SCP) **44** (2002), pp. 181–203.

[3] Bardohl, R., C. Ermel and H. Ehrig, *Generic Description of Syntax, Behavior and Animation of Visual Models*, TR 2001/19, TU Berlin (2001). http://www.cs.tu-berlin.de/%7Elieske/public/TR-Anim01.ps.gz

[4] Bardohl, R., C. Ermel and J. Padberg, *Formal Relationship between Petri Nets and Graph Grammars as Basis for Animation Views in GenGED*, in: *Proc. IDPT 2002: Sixth World Conference on Integrated Design and Process Technology* (2002). http://www.cs.tu-berlin.de/%7Elieske/public/IDPT02.ps.gz

[5] Bardohl, R., G. Taentzer, M. Minas and A. Schürr, *Application of Graph Transformation to Visual Languages*, in: H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools* (1999).

[6] Corradini, A. and U. Montanari, *Specification of Concurrent Systems: From Petri Nets to Graph Grammars*, in: G. Hommel, editor, *Proc. Workshop on Quality of Communication-Based Systems, Berlin, Germany* (1995).

[7] Ehrig, H. and B. Mahr, *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics,* EATCS Monographs on TCS **6**, Springer, Berlin, 1985.

[8] Ehrig, H., U. Prange and G. Taentzer, *Fundamental theory for typed attributed graph transformation*, in: F. Parisi-Presicce, P. Bottoni and G. Engels, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, Springer LNCS 3256 (2004), pp. 161–177.

[9] Ehrig, K., *Konzeption und Implementierung eines Generators für Animationsumgebungen für visuelle Modellierungssprachen*, TR 2003-17, TU Berlin (2003).

[10] Ermel, C. and R. Bardohl, *Scenario Animation for Visual Behavior Models: A Generic Approach*, Software and System Modeling: Special Section on Graph Transformations and Visual Modeling Techniques **5** (2004).

[11] *GenGED Homepage*, http://tfs.cs.tu-berlin.de/genged.

[12] Kreowski, H.-J., *A Comparison between Petri-nets and Graph Grammars*, in: *LNCS 100* (1981), pp. 1–19.

[13] Kreowski, H.-J. and S. Kuske, *Graph transformation units with interleaving semantics*, Formal Aspects of Computing **11** (1999), pp. 690–723.

[14] Minas, M., *Diagram Editing with Hypergraph Parser Support*, in: *Proc. IEEE Symp. on Visual Languages*, Capri, Italy, 1997, pp. 226–233.

[15] Padberg, J., H. Ehrig and L. Ribeiro, *Algebraic high-level net transformation systems*, Mathematical Structures in Computer Science **5** (1995), pp. 217–256.

[16] Parisi-Presicce, F., H. Ehrig and U. Montanari, *Graph Rewriting with Unification and Composition*, in: *3rd Workshop on Graph Grammars and their Application to Computer Science, Springer LNCS 291* (1987), pp. 496–514.

[17] Reisig, W., *Petri Nets*, EATCS Monographs on Theoretical Computer Science **4**, Springer, 1985.

[18] Taentzer, G., *AGG: A Graph Transformation Environment for Modeling and Validation of Software*, in: *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, Springer LNCS 3062 (2004).

[19] *Unified Modeling Language – version 1.5,* (2004), available at http://www.omg.org/technology/documents/formal/uml.htm.

[20] WWW Consortium (W3C), *Scalable Vector Graphics (SVG) 1.0 Specification.* http://www.w3.org/TR/svg, (2000).