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

Claudia Ermel\textsuperscript{1}, Karsten Ehrig\textsuperscript{2},

\textit{Institut für Softwaretechnik und Theoretische Informatik}\n\textit{Technische Universität Berlin, Germany}

\textbf{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 \textit{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.

\textit{Key words:} graph transformation, Petri nets, animation

\section{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.

\textsuperscript{1} Email: lieske@cs.tu-berlin.de\n\textsuperscript{2} Email: karstene@cs.tu-berlin.de

\textit{This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science}
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 [21] 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 [20,1], GENGE [2,12] or DiaGen [15] 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 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-

![Fig. 1. Different views on formal models](image-url)
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 [11] 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 views. 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

In this section, we review the definition of AHL nets and their behavior, and present our running example, the specification of the well-known Dining Philosophers as AHL net.

The version of AHL-nets defined in this section corresponds to [7]. Places are typed, that is, the data elements on these places and the terms occurring in the inscriptions of the attached arcs are required to be of a specified sort. This typing reduces the marking graph considerably.

Definition 2.1 (Algebraic High-Level Net)

An algebraic high-level net (AHL-net) \( N = (\text{SPEC}, P, T, \text{pre, post, cond, type, } A) \) consists of an algebraic specification \( \text{SPEC} = (S, OP, E; X) \), sets \( P \) and \( T \) of places and transitions respectively, pre- and post-domain functions \( \text{pre, post} : T \rightarrow (T_{OP}(X) \otimes P)^\oplus \) assigning to each transition \( t \in T \) the pre- and post-domains \( \text{pre}(t) \) and \( \text{post}(t) \) (see first Remark), respectively, a firing condition function \( \text{cond} : T \rightarrow \mathcal{P}_{\text{fin}}(\text{EQNS}(S, OP, X)) \) assigning to each transition \( t \in T \) a finite set \( \text{cond}(t) \) of equations over the signature \( (S, OP) \) with variables \( X \), a type function \( \text{type} : P \rightarrow S \) assigning to each place \( p \in P \) a sort \( \text{type}(p) \in S \), and an \( (S, OP, E) \)-algebra \( A \) (see [8]).

Remarks

(i) Denoting by \( T_{OP}(X) \), or more precisely \( T_{(S,OP)}(X) \) the set of terms with variables \( X \) over the signature \( (S, OP) \) (see [8]), and by \( M^\oplus \) the free
commutative monoid over a set $M$, the set of all type-consistent arc inscriptions $T_{OP}(X) \otimes P$ is defined by $T_{OP}(X) \otimes P = \{(\text{term}, p)|\text{term} \in T_{OP}(X)_{\text{type}(p)}, p \in P\}$.

Thus, $\text{pre}(t)$ (and similar $\text{post}(t)$) is of the form $\text{pre}(t) = \sum_{i=1}^{n}(\text{term}_i, p_i)$ ($n \geq 0$) with $p_i \in P, \text{term}_i \in T_{OP}(X)_{\text{type}(p_i)}$. This means, $\{p_1, \ldots, p_n\}$ is the pre-domain of $t$ with arc-inscription $\text{term}_i$ for the arc from $p_i$ to $t$ if the $p_1, \ldots, p_n$ are pairwise distinct (unary case) and arc-inscription $\text{term}_{i_1} \oplus \ldots \oplus \text{term}_{i_k}$ for $p_{i_1} = \ldots = p_{i_k}$ (multi case). In our sample AHL net (see Example 2.3) we have the multi case. (ii) As places are typed, a marking $m$ is an element $m \in (A \otimes P)^\oplus$ with $A \otimes P = \{(a, p)|a \in A_{\text{type}(p)}, p \in P\}$.

Enabling and firing of transitions are defined as follows.

**Definition 2.2 (Firing Behavior of AHL Nets)**

Given an AHL-net as above and a transition $t \in T$, $\text{Var}(t)$ denotes the set of variables occurring in $\text{pre}(t), \text{post}(t)$, and $\text{cond}(t)$. An assignment $\text{asg}_A : \text{Var}(t) \to A$ is called consistent if the equations $\text{cond}(t)$ are satisfied in $A$ under $\text{asg}_A$.

The marking $\text{pre}_A(t, \text{asg}_A) – \text{similarly } \text{post}_A(t, \text{asg}_A)$ is defined for $\text{pre}(t) = \sum_{i=1}^{n}(\text{term}_i, p_i)$ by $\text{pre}_A(t, \text{asg}_A) = \sum_{i=1}^{n}(\text{asg}_A(\text{term}_i), p_i)$, where $\text{asg}_A : T_{OP}(\text{Var}(t)) \to A$ is the extension of the assignment $\text{asg}_A$ to an evaluation of terms (see [8]).

A transition $t \in T$ is enabled under a consistent assignment $\text{asg}_A : \text{Var}(t) \to A$ and marking $m \in (A \otimes P)^\oplus$, if $\text{pre}_A(t, \text{asg}_A) \leq m$. In this case, the successor marking $m'$ is defined by $m' = m \ominus \text{pre}_A(t, \text{asg}_A) \oplus \text{post}_A(t, \text{asg}_A)$ and gives raise to a firing step $m[t, \text{asg}_A]m'$.

**Example 2.3 (The Dining Philosophers as AHL Net)**

As example we show the AHL net for The Dining Philosophers in Fig. 2 (see [19,16] 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 consisting of sorts $\text{Nat}$ for natural numbers, $\text{String}$ for place and transition names, and $\text{Bool}$ for firing conditions. All places have the type $\text{Nat}$, i.e. all tokens are elements of a $\text{Nat}$-algebra, The arcs are inscribed each by one or more variables or terms from
Fig. 2. The Dining Philosophers (a) modeled as AHL Net (b)

$T_{op}(X)$ denoting computation operations to be executed on token values if the transition fires.

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 VL alphabet ($TG, Csp$) and a VL syntax grammar. The alphabet on the one hand specifies the abstract syntax (the symbols and links used in the VL) by means of a type graph, formally given as attributed graph structure signature. On the other hand, the concrete syntax (the layout of the graphical representations of symbols and links) is given by extending the abstract syntax type graph by graphical operation symbols and links (cf. [3]) yielding the complete type graph $TG$, and by a graphic Constraint Satisfaction Problem $Csp$ on positions and sizes of the used graphics. VL sentences (diagrams according to the VL alphabet) are graphs typed over $TG$, that satisfy the equations in the $Csp$ and that can be derived by applying the VL syntax grammar rules.

In the following, we present attributed graph structures as defined in [9] and define the visual language for AHL nets.

**Definition 3.1 (Attributed Graph Structure Signatures)** A graph structure signature $GSIG = (S_G, OP_G)$ is an algebraic signature with unary operations $op : s \rightarrow s'$ 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 $OP_D \cap OP_G = \emptyset$.

$ASSIG$ is called well-structured if for each $op : s \rightarrow s'$ in $OP_G$ we have $s \notin S_D$.

$ASSIG$-algebras and $ASSIG$-homomorphisms build a category [9] which
is denoted by $\text{ASSIG-Alg}$. In the following, we call $\text{ASSIG}$-algebras attributed graphs and $\text{ASSIG}$-homomorphisms attributed graph morphisms.

Now we can define the attributed graph structure signature $\text{ASSIG}_{\text{AHL}}$ for AHL nets. AHL nets are considered as $\text{ASSIG}_{\text{AHL}}$-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 $\text{ASSIG}_{\text{AHL}} = (\text{GSIG}_{\text{AHL}}, \text{DSIG}_{\text{AHL}})$. In Fig. 3, the sorts of $\text{GSIG}_{\text{AHL}}$ are represented as nodes. The operations are the arcs between the sort nodes (the $\text{op}$-links between graph sorts), from sort nodes to data nodes, (the $\text{attr}$-links between graph sorts and attribute sorts) and the arcs connecting the abstract syntax sort nodes and the concrete syntax sort nodes (the $\text{graphic}$-links). The $\text{DSIG}$ part (data signature) consists of the attribute value sorts of the basic specification, i.e. Nat, Bool and String, and their usual operations. The attribute values are used for the arc inscriptions, tokens and transition firing conditions.

Graphical constraints are indicated by dotted arrows at the concrete syntax level of Fig. 3 and define for example that the token number is drawn inside 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.

As an example for a layout condition which should hold for all AHL nets (i.e. sentences of our AHL net language) we consider the formalization of the condition “Token numbers are written inside the ellipse representing their corresponding place,” as graphical constraint of the Csp. We use the variables $a$ and $b$ for arbitrary graphical objects, $lt$ indicates the left top corner point of an object, and $w$ and $h$ its width and height. Point coordinates $x$ and $y$ of point $P$ are written $P.x$ and $P.y$. This means that e.g. $a.lt.x$ denotes the $x$-coordinate of the left top corner of object $a$. The constraint is a set

![Fig. 3. Visual Alphabet for AHL Nets](image-url)
of inequations over these variables and expresses that object \( b \) is completely inside object \( a \):

\[
\begin{align*}
\text{inside}(\text{Object } a, \text{Object } b) & \{ \\
& a.lt.x < b.lt.x; \\
& a.lt.x + a.w > b.lt.x + b.w \\
& a.lt.y < b.lt.y; \\
& a.lt.y + a.h > b.lt.y + b.h; \\
& \}
\end{align*}
\]

Fig. 4. Layout condition inside as graphical constraint of the Csp

The other layout conditions are formalized by more complex constraints but with the same underlying principle: the scope of possible variable bindings is restricted by equations/inequations over constraint variables denoting the position or size of graphical objects. The set of constraints defined by the language designer together with some initial constraints (e.g. all graphical objects are positioned within the editor panel and have a default size) comprise the CSP which has to be satisfied by all visual sentences of the language.

In general, a VL syntax grammar \((S, P)\) consists of a start graph \( S \) and a set \( P \) of language defining rules. These rules are defined at the abstract syntax level and restrict the set of visual sentences of the VL to the meaningful ones. The start graph of the VL syntax grammar for AHL nets consists of a single Net node only. Fig. 5 shows the VL syntax rules for the VL of AHL nets, realising the insertion of places, transitions, arcs and tokens. A negative application condition (NAC) specifies a situation which must not be present in the graph the rule is to be applied to. Thus, it is possible to forbid e.g. that the same place name is taken for two different places (see rule \( \text{insPlace}(\text{PlName}) \)).

Valid visual sentences (AHL nets) are derived by one or more syntax rule applications at the abstract syntax level. The concrete syntax of a sentence is computed after its derivation in a way that the resulting layout satisfies the CSP of the AHL net alphabet. The VL syntax grammar together with the corresponding VL alphabet define the visual language \( VL = \{ VLS \mid S \xrightarrow{\ast} P \}^{\text{VLS}} \), 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 \( P \).
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 \textit{ASSIG-Alg}.

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

\textbf{Proposition 3.3 (Pushouts of \textit{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 \( f_{DSIG} = \text{id}_{DSIG} \) for \( f \) in \textit{ASSIG-Alg}. Given \( f : A \rightarrow B \in M \) and \( a : A \rightarrow C \) then there exists their pushout in \textit{ASSIG-Alg}.

\textbf{Proof:} See [9].

\textit{Category \textit{ASSIG-Alg} and class } \( M \) \textit{are fixed throughout this section.}

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

(i) a rule \( p = (L \xleftarrow{\upsilon} I \xrightarrow{r} R) \) of \textit{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 \( \upsilon, r \in M \), i.e. \( \upsilon \) and \( r \) are injective and identities on the data type \( T_{DSIG}(X) \),
(ii) a direct transformation $G \xrightarrow{p,m} H$ 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 $G_0 \xrightarrow{S} G_n$.

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

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 $(TG, Csp)$. 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: $ML = \{ D \mid S \xrightarrow{S} D \}$ where the layout of each VL sentence $D$ satisfies the layout constraints in $Csp$.

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 [13], Parisi-Presicce et al. [18] 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 an AHL 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 datatype 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 \( \text{ASSIG-Alg} \) 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 \((1, \ldots, 5)\) on place thinking, all chopsticks \((1, \ldots, 5)\) 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. 6). 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 \( p \).

![Behavior rules for the AHL net model Dining Philosophers](image)

**Fig. 6.** 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. Analogously to the VL syntax rules (see Fig. 5), the behavior rules are defined at the abstract syntax level, only. The concrete syntax of derived AHL nets (denoting system states) is again computed after the derivation according to the Csp of the VL alphabet for AHL nets.

**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 $\text{ASSIG}_{\text{AHL}}$-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 $a : p_i \rightarrow t$ [$a : t \rightarrow p_i$]

for each arc inscription term $tm \in \text{inscr}(a)$
genenerate a token node of type Token attributed by a copy $tk$ of $tm$;
connect the token node by an arc of type $\text{EdgeTk}$ 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 an animation view as a special case of a view.

4.1 Views for Behavior Models

Fig. 7 shows some aspects of the characterization of views in UML [21]. 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:

- The basic system model is a VL model (see Def. 3.5), i.e. a typed graph transformation system $M = (TG, S, P)$. 

![Fig. 7. Relation of Model and Views](image-url)
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. 4.2) 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 \( M \), according to the definition of subrule embedding (Def. 4.1).

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 (Subrule Embedding)**

Given a rule \( p = (L \xleftarrow{l} I \xrightarrow{e} R) \), a rule \( s = ((L_s \xleftarrow{l_s} I_s \xrightarrow{e} R_s) \) is called subrule of \( p \) if there are injective graph morphisms \( e : L_s \rightarrow L \), \( f : I_s \rightarrow I \) and \( g : R_s \rightarrow R \) such that the diagrams to the right are pullbacks.

The triple \( t = (e, f, g) \) from \( s \) to \( p \) (short \( t : s \rightarrow p \)) is called subrule embedding. In this context, \( p \) is called extending rule.

**Definition 4.2 (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 view embedding \( v : V \rightarrow M \) is called view of \( M \) or restriction of \( M \) to \( TG_V \), written \( V = M|_{TG_V} \).

A view embedding \( v \) is a tuple \( v = (tv : TG_V \rightarrow TG, s : S_V \rightarrow S, f_p : P_V \rightarrow P, \{se(p_V), p_V \in P_V\}) \) where \( tv : TG_V \rightarrow TG \) is the type graph inclusion and \( s : S_V \rightarrow S \) is graph restriction to \( TG_V \), written \( S_V = S|_{TG_V} \), where the diagram to the right is a pullback.

For each rule \( p = (L \xleftarrow{l} I \xrightarrow{e} R) \in P \), the rule \( p_V = (L_V \xleftarrow{l_V} I_V \xrightarrow{r_V} R_V) \) with \( f_p(p_V) = p \) is obtained as follows: \( L_V \xleftarrow{e} L \), as well as \( I_V \xrightarrow{f} I \) and \( R_V \xrightarrow{g} R \) are graph restrictions to \( TG_V \) (constructed as pullbacks, similar to the construction of \( s : S_V \rightarrow S \)). The morphisms \( L_V \xleftarrow{l_V} I_V \) and \( I_V \xrightarrow{r_V} R_V \) are the unique morphisms due to pullback construction of \( f : I_V \rightarrow I \).

Due to pullback composition, we have the double pullback construction shown in the diagram to the right. Thus, \( se(p_V) = (e, f, g) \) from \( p_V \) to \( p \) is a subrule embedding according to Def. 4.1.
We now define constructions for the interaction of two views and for the integration of different views into one view.

**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 \(I = (TG_I, S_I, P_I)\) where \(TG_I\) is given by the pullback to the right (called interaction pullback), and \((I, I \xrightarrow{i_1} V_1)\) and \((I, I \xrightarrow{i_1} V_2)\) are the views \(V_1|_{TG_I}\) and \(V_2|_{TG_I}\).

**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). Analogously, \(S_U\) is constructed as pushout object of \(S_1\) and \(S_2\) over \(S_I\). 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 (represented by an alphabet of the animation domain \((TG_A, \text{Csp}_A))\). Formally, to define an animation view \(AV\) for a formal VL model \(F = (TG_F, S_F, P_F)\), the basic alphabets for the VL model and for the animation domain have to be united to an integrated view alphabet (see Def. 4.5). A set of view transformation rules over the integrated view alphabet describes how the symbols and links of \(F\) are extended coherently by animation symbols and links (Def. 4.6). The translation of the initial graph \(S_F\) to the initial animation graph \(S\) and of the set of behavior rules \(P_F\) to animation rules \(P\) is realized by applying the view transformation rules and yields the integrated view model \(M = (TG_F \cup TG_A, S, P)\) (see Def. 4.8). We show that the \(F\) is a view of \(M\), i.e. \(F = M|_{TG_F}\). At last, we define the animation view \(AV\) for \(F\) to be the restriction \(AV = M|_{TG_A}\). Note that both the formal model \(F\) and its animation view \(AV\) are different views of the same integrated view model \(M\).

**Definition 4.5 (Integrated View Alphabet)**
Let \((TG_A, \text{Csp}_A)\) be the alphabet of an animation domain, and \((TG_F, \text{Csp}_F)\) the alphabet of a formal specification language. Then the integrated view alphabet \(VA = (TG_F \cup TG_A, \text{Csp}_F \cup \text{Csp}_A)\) consists of the type graph \(TG_F \cup TG_A\), constructed as pushout in ASSIG-Alg, and of the constraint satisfaction problem \(\text{Csp}_F \cup \text{Csp}_A\), obtained by union of the two sets of
equations $C_{sp_F}$ and $C_{sp_A}$.

Obviously, a VL model $F = (TG_F, S_F, P_F)$ over the alphabet $(TG_F, C_{sp_F})$ is also a VL model over the integrated view alphabet $VA$.

**Definition 4.6 (View Transformation Rules)**

Let $VA = (TG = TG_F \cup TG_A, C_{sp} = C_{sp_F} \cup C_{sp_A})$ be an integrated view alphabet. A view transformation rule $p_v = (L_v \xrightarrow{id} L_v \xrightarrow{r_v} R_v)$ is a rule typed over $TG$ with $L_v|_{TG_F} = R_v|_{TG_F}$. $VTR$ is a set of view transformation rules.

**Remarks:**

View transformation rules are non-deleting, thus $L_v = I_v$. Moreover, they only add objects that are typed over $TG_A$. All objects typed over $TG_F$ are preserved by view transformation rules, hence $L_v|_{TG_F} = R_v|_{TG_F}$.

In order to be able to apply view transformation rules to behavior rules, we need a general construction defining how to apply rules to rules:

**Definition 4.7 (Rewriting Rules by Rules)**

Let $q = (L_q \xleftarrow{l_q} I_q \xrightarrow{r_q} R_q)$ be a rule to be applied to another rule $p_1 = (L_1 \xleftarrow{l_1} I_1 \xrightarrow{r_1} R_1)$. Then a derivation step $p_1 = q \Rightarrow p_2$ with $p_2 = (L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ is defined depending on the existence of matches:

- **Case (1)** there exists a match $L_q \xrightarrow{h} I_1$:
  Then $q$ is applied via the matching $h : L_q \rightarrow I_1$ as in the following diagram:

  $L_q \quad I_q \quad R_q$
  \[
  \xrightarrow{h} \quad \xrightarrow{} \quad \xrightarrow{}
  \]
  $I_1 \quad I' \quad I_2$
  \[
  \xleftarrow{} \quad \xleftarrow{} \quad \xrightarrow{}
  \]
  $L' \quad L \quad L_2$
  \[
  \xleftarrow{} \quad \xleftarrow{} \quad \xrightarrow{}
  \]
  $R_1 \quad R' \quad R_2$
  \[
  \xrightarrow{} \quad \xrightarrow{} \quad \xrightarrow{}
  \]

  The new rule $p_2 = (L_2 \xleftarrow{l_2} I_2 \xrightarrow{r_2} R_2)$ is obtained by applying $q$ to $I_1$ via the matching morphism $h : L_q \rightarrow I_1$, to $L_1$ via the matching morphism $L_q \rightarrow I_1 \rightarrow L_1$ and to $R_1$ via the matching morphism $L_q \rightarrow I_1 \rightarrow R_1$.

- **Case (2)** there exists no match $L_q \rightarrow I_1$ but a match $L_q \xrightarrow{h} R_1$. In this case, $q$ is applied to $R_1$ only. The result of the application of $q$ to $R_1$ is illustrated in the following diagram:
\[ L_q \rightarrow I_q \rightarrow R_q \]
\[ h \]
\[ R_1 \rightarrow R' \rightarrow R_2 \]
\[ I_1 \rightarrow I_2 \]
\[ L_1 = L_2 \]

where \( R_2 \) is the object resulting from the direct derivation via \( q \) of \( R_1 \), \( I_2 \) is the common part (pullback) of \( R' \) (left unchanged by the application of \( q \) to \( R_1 \)) and \( I_1 \) (left unchanged by any application of \( p_1 \)), and \( L_2 \) is just the unchanged left-hand side of \( p_1 \).

• Case (3) there exists no match \( L_q \rightarrow I_1 \) but a match \( L_q \rightarrow^h L_1 \). In this case, \( q \) is applied to \( L_1 \) only, as illustrated in the following diagram:

\[ L_q \rightarrow I_q \rightarrow R_q \]
\[ h \]
\[ L_1 \rightarrow L' \rightarrow L_2 \]
\[ I_1 \rightarrow I_2 \]
\[ R_1 = R_2 \]

The resulting rule \( p_2 \) is obtained from the application of \( q \) to the left hand side \( L_1 \) of \( p_1 \). \( I_2 \) is again constructed as pullback, and \( R_2 \) is the unchanged right-hand side of \( p_1 \).

Remarks:
• In case (1), the rule morphisms of the resulting rule \( p_2 = (L_2 \leftarrow I_2 \rightarrow R_2) \) contain mappings between objects that are preserved by \( q \), and mappings between objects that are added to all three graphs by \( q \). For this case, it has been shown in [17] that the rewriting of rule \( p_1 \) to rule \( p_2 \) via rule \( q \) is reflected in the correspondence between the objects produced by \( p_1 \) and those produced by \( p_2 \):

\[
\begin{align*}
q : p_1 \Rightarrow p_2 & \text{ via } h : L_q \rightarrow I_1, \\
g_1 : L_1 \rightarrow G_1, q : G_1 \Rightarrow G_2 & \text{ via } f = g_1 \circ l_1 \circ h : L_q \rightarrow G_1, \\
p_2 : G_2 \Rightarrow H_2 & \text{ via } g_2 : L_2 \rightarrow G_2, \\
\end{align*}
\]

as indicated in the diagram to the right.
• In case (2), \( q \) only modifies the items that are produced by \( p_1 \). The new
rule \( p_2 \) coincides with \( p_1 \) in the left-hand side \( L_1 \), but has a different right-hand side. A modification of \( R_1 \) induces (in general) a modification of the interface part \( I_1 \).

- In case (3), \( q \) only modifies the items that are removed by \( p_1 \). The new rule \( p_2 \) coincides with \( p_1 \) in the right-hand side \( R_1 \), but has a different left-hand side.

**Definition 4.8 (Integrated View Model)**

Let \( F = (TG_F, S_F, P_F) \) be a VL model, and \( TG = TG_F \cup TG_A \) the type graph of the integrated view alphabet. Let \( VTR \) be a sequence of view transformation rules, typed over \( TG \). Then \( M = (TG, S, P) \) is the integrated view model for \( F \), which is constructed as follows: the start graph \( S \) of \( M \) is derived from \( S_F \) by applying the rules of \( VTR \) to \( S_F \) as long as possible. The rules in \( P \) are called integrated animation rules. Each integrated animation rule \( p = (L \leftarrow I \rightarrow R) \in P \) is derived from a corresponding behavior rule \( p_F = (L_F \leftarrow I_F \rightarrow R_F) \in P_F \), by applying the rules of \( VTR \) to \( p_F \) as long as possible, where each derivation is constructed according to Def. 4.7. The construction of the integrated view model is called view transformation.

**Proposition 4.9 (\( F \) is View of the Integrated View Model)**

Let \( F = (TG_F, S_F, P_F) \) be a VL model and \( M = (TG, S, P) \) its integrated view model constructed by view transformation with \( VTR \) according to Def. 4.8. Then, \( F \) is a view of \( M : F = M \mid TG_F \).

**Proof Sketch:**

We have to show that there exists a view embedding \( v = (TG_F \xrightarrow{t} TG, S_F \xrightarrow{s} S, P_F \xrightarrow{f_P} P) \) such that for each \( p_F \in P_F \) there is a subrule embedding \( p_F \xrightarrow{(e,f,g)} p \).

- By construction of \( TG \), we know that \( TG_F \xrightarrow{t} TG \) is type graph inclusion.
- \( S_F \xrightarrow{s} S \) is graph restriction to \( TG_F \) because the diagram \( S_F \xrightarrow{s} S \rightarrow TG = S_F \rightarrow TG_F \xrightarrow{t} TG \) is a pullback, which can be shown by adequate pullback compositions starting from the definition of view transformation rules and their restrictions to \( TG_F \).
- The rule mapping \( f_P \) maps each rule \( p_F \) to the rule \( p \) resulting from the view transformation step of applying the rules in \( VTR \) to \( p_F \). As view transformation rules only add new objects but do not delete any, the morphisms \( e : L_F \rightarrow L, f : I_F \rightarrow I \) and \( g : R_F \rightarrow R \) are injective. Moreover, the diagrams \( I_F \rightarrow L_F \rightarrow L = I_F \rightarrow I \rightarrow L \) and \( I_F \rightarrow R_F \rightarrow R = I_F \rightarrow I \rightarrow R \) are pushouts in all three cases of rewriting rule \( p_F \) in Def. 4.7 and due to the fact that \( L_q = I_q \) in all view transformation rules. This yields a valid subrule embedding \( (e,f,g) : p_F \rightarrow p \) for each rule transformation.

**Definition 4.10 (Animation View of a VL Model)**

Let \( (TG_A, Csp_A) \) be the alphabet of an animation domain. Let \( F = (TG_F, S_F, P_F) \) be a VL model over the alphabet \( (TG_F, Csp_F) \). Let \( VTR \) be a set of view
transformation rules typed over $TG = TG_F \cup TG_A$. Let $M = (TG, S, P)$ be the integrated view model constructed from $F$ by applying the rules in $VTR$. Then, the corresponding animation view of $M$ is defined as view $AV = M|_{TG_A}$.

4.3 View Transformation for Petri Nets

We suggest the following guidelines for the definition of the integrated view alphabet and the view transformation rules for the case that the formal specification language is the VL of Petri nets:

- In the integrated view alphabet, the Net symbol from the Petri net alphabet is linked to the top-level symbol of the animation domain alphabet. This top-level symbol represents the animation context, i.e. the part of the view which is not changed by animation and where all animated symbols should be linked to.

- The animated symbols that are changed (moved) during animation directly correspond to the tokens of the Petri net. The layout and position of these animated symbols depends on the place where the token is lying in the active state. Therefore, each view transformation rule inserts an animated symbol (or a set of symbols) for each token, depending on their places (e.g. a chopstick symbol is inserted for each token on the place table) and links it to the animation context (e.g. the table symbol).

Example 4.11 Animation View for The Dining Philosophers

Fig. 8 shows the integrated view alphabet ($TG, CSP$) for the Dining Philosophers where the AHL net alphabet ($TG_F, CSP_F$) (lower part of Fig. 8) is united with an animation domain alphabet ($TG_A, CSP_A$) (upper part of Fig. 8). The animation context consists of a round table with numbered plates on top. The animated symbols are the thinking and eating philosophers positioned around the table, and the chopsticks besides the plates. In addition to the graphical constraints for AHL nets ($CSP_F$), the integrated view alphabet now contains constraints for positioning the philosopher symbols and the chopsticks at the table in relation to their plates ($CSP_A$, not depicted in Fig. 8).

Fig. 9 shows the view transformation rules which are typed over the integrated view alphabet in Fig. 8. Note that, again, for rules we only have to define the abstract syntax, as the concrete syntax of the derived graph is always computed according to the CSP of the alphabet.

One initial rule generates the animation context and links it to the abstract syntax of the Net symbol. Token rules then generate an animated symbol for each token depending on its place and link the animated symbol to the Table symbol. After applying all view transformation rules as long as possible, first to the start graph $S_F$ of the VL model (i.e. the initially marked AHL net) and then to the behavior rules $p_F \in P_F$, the formal VL model $F$ is transformed into an integrated view model $M = (TG, S, P)$, which now also contains symbols from the animation domain alphabet in the start graph $S$ and in the rules $P$.
Fig. 8. Integrated view alphabet for *The Dining Philosophers*

Fig. 9. View transformation rules for *The Dining Philosophers* (now called integrated animation rules).

Fig. 10 illustrates an integrated animation rule derived by applying the view transformation rules in Fig. 9 to the behavior rule *take* in Fig. 6. To both sides of the behavior rule *take*, the view transformation rule *init* added the animation context, the symbol *table*. To the left-hand side of the behavior rule, the view transformation rule *table* added two animated symbols of type *Chopstick* for the two tokens on place *table*, and the view transformation rule *thinking* added one animated symbol of type *ThinkingPhilo* for the token on place *thinking*. To the right-hand side of the behavior rule, the view transformation
The derived animation rule for *take* added one animated symbol of type *EatingPhilo*.

![Animation Rule for Take](image)

Fig. 10. Derived animation rule for *The Dining Philosophers*

The second animation rule for *put* is constructed analogously and equals the reversed rule for *take*. The integrated animation rules in *P* now model the behavior of the *Dining Philosophers* according to the AHL net model *F*, but visualized also in the animation domain. In the animation view (the restriction of *M* to *TG_A*), only the symbols from the animation domain are visualized, but the behavior still corresponds to the definition of the formal VL model behavior. Fig. 11 shows the result of the application of animation rule *take* to a system state (the graph *S_A*) where one philosopher is eating (holding two chopsticks) and the others are thinking. The state change now is visualized in the animation view, i.e. only the concrete syntax of the animation symbols typed over *TG_A* is shown.

![Animation Rule for Put](image)

Fig. 11. Behavior of *The Dining Philosophers* in the animation view

The nature of our animation differs in two respects from the notion of *simulation*. *Simulation* usually 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). Up to now, we model animation steps by graph rules (animation rules) in the layout of the animation view, which leads again to discrete state changes. In order to allow continuous state change visualization, we can enhance these animation rules by operations for continuous changes.
of objects such as motions or changes of size or color. Therefore, using the GENGED environment, it is possible e.g. to enhance the animation rule take (see Fig. 10) by animation operations for the smooth movement of the chopsticks from the table towards the philosopher who is going to eat.

5 Implementation of Animation Views in GenGED

Fig. 12 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. 12. The GENGED environment extended by features for animation

We explain Fig. 12 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 specify 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 Merge Alphabet 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
Ermel and Ehrig

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 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 [10] 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. Fig. 13 shows a screenshot of the animation editor with an animation rule from the Dining Philosophers example. Here, move-operations for the two chopstick icons are defined. In general, more than one animation operation can be defined for one rule and the starting time and duration for each animation operation can be specified conveniently using the time bar at the bottom of the window.

![Animation editor of the GenGED environment](image.png)

Fig. 13. Animation editor of the GenGED environment

(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 [22]) and can be viewed by any external SVG viewer tool or SVG-capable browser. 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. [14]) 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


URL http://www.cs.tu-berlin.de/%7Elieske/public/IDPT02.ps.gz


URL http://www.cs.tu-berlin.de/%7Elieske/public/gEB04.pdf


URL http://www.w3.org/TR/svg11/