Weakest Preconditions for High-Level Programs

Annegret Habel[1], Karl-Heinz Pennemann[1], and Arend Rensink[2]

[1] University of Oldenburg, Germany**
{habel,pennemann}@informatik.uni-oldenburg.de
[2] University of Twente, Enschede, The Netherlands
rensink@cs.utwente.nl

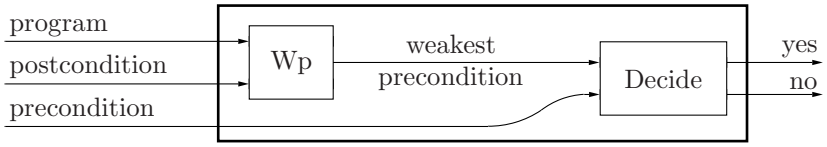
Abstract. In proof theory, a standard method for showing the correctness of a program w.r.t. given pre- and postconditions is to construct a weakest precondition and to show that the precondition implies the weakest precondition. In this paper, graph programs in the sense of Habel and Plump 2001 are extended to programs over high-level rules with application conditions, a formal definition of weakest preconditions for high-level programs in the sense of Dijkstra 1975 is given, and a construction of weakest preconditions is presented.

1 Introduction

Graphs and related structures are associated with an accessible graphical representation. Transformation rules exploit this advantage, as they describe local change by relating a left- and a right-hand side. Nondeterministic choice, sequential composition and iteration give rise to rule-based programs [19].

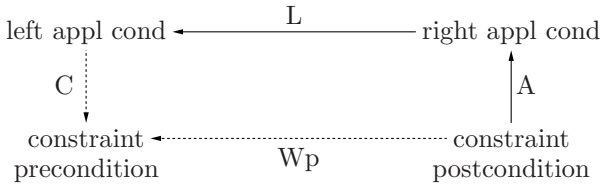
Formal methods like verification with respect to a formal specification are important for the development of trustworthy systems. We use a graphical notion of conditions to specify valid objects as well as morphisms, e.g. matches for transformation rules. We distinguish the use of conditions by speaking of constraints in the first case, and application conditions for rules in the latter. Conditions seem to be adequate for describing requirements as well as for reasoning about the behavior of a system.

A well-known method for showing the correctness of a program with respect to a pre- and a postcondition (see e.g. [7, 8]) is to construct a weakest precondition of the program relative to the postcondition and to prove that the precondition implies the weakest precondition.



** This work is supported by the German Research Foundation (DFG), grants GRK 1076/1 (Graduate School on Trustworthy Software Systems) and HA 2936/2 (Development of Correct Graph Transformation Systems).

In this paper, we use the framework of weak adhesive HLR categories to construct weakest preconditions for high-level rules and programs, using two known transformations from constraints to right application conditions, and from right to left application conditions, and additionally, a new transformation from application conditions to constraints.



The paper is organized as follows. In Section 2, high-level conditions, rules and programs are defined and an access control for computer systems is introduced as a running example. In Section 3, two basic transformations of [16] are reviewed and, additionally, an essential transformation from application conditions into constraints is presented. In Section 4, weakest preconditions for high-level programs are formally defined and a transformation of programs and postconditions into weakest preconditions is given. In Section 5, related concepts and results are discussed. A conclusion including further work is given in Section 6. A long version of this paper including properties of weakest preconditions and detailed proofs of some results is available as a technical report [18].

2 Conditions and programs

In this section, we will review the definitions of conditions, rules, and programs for high-level structures, e.g. graphs. We use the framework of weak adhesive HLR categories introduced as combination of HLR systems and adhesive categories. A detail introduction introduction can be found in [14, 15]. As a running example, we consider a simple graph transformation system consisting of rules and programs. We demonstrate that programs are necessary extensions of rules for certain tasks and conditions can be used to describe a wide range of system properties, e.g. security properties.

Assumption. We assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with a category $\mathcal{C}$, a class $\mathcal{M}$ of monomorphisms, a $\mathcal{M}$-*initial* object, i.e. an object I in $\mathcal{C}$ such that there exists a unique morphism $I \to G$ in $\mathcal{M}$ for every object G in $\mathcal{C}$; binary coproducts and *epi-$\mathcal{M}$-factorization*, i.e. for every morphism there is an epi-mono-factorization with monomorphism in $\mathcal{M}$.

For illustration, we consider the category **Graph** of all directed, labeled graphs, which together with the class $\mathcal{M}$ of all injective graph morphisms constitutes a weak adhesive HLR category with binary coproducts and epi-$\mathcal{M}$-factorization and the empty graph $\emptyset$ as the $\mathcal{M}$-initial object.

Example 1 (access control graphs). In the following, we introduce state graphs of a simple access control for computer systems, which abstracts authentication and models user and session management in a simple way. We use this example solely for illustrative purposes. A more elaborated, role-based access control model is considered in [22]. The basic items of our model are users, sessions, logs, computer systems, and directed edges between those items. An edge between a user and a system represents that the user has the right to access the system, i.e. establish a session with the system. Every user node is connected with one log, while an edge from a log to the system represents a failed (logged) login attempt. Every session is connected to a user and a system. The direction of the latter edge differentiates between sessions that have been proposed (an outgoing edge from a session node to a system) and sessions that have been established (an incoming edge to a session node from a system). Self-loops may occur in graphs during the execution of programs to select certain elements, but not beyond. An example of an access control graph is given in Figure 1.

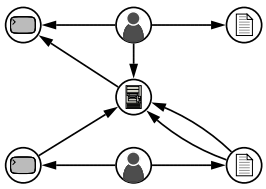


Fig. 1. A state graph of the access control system

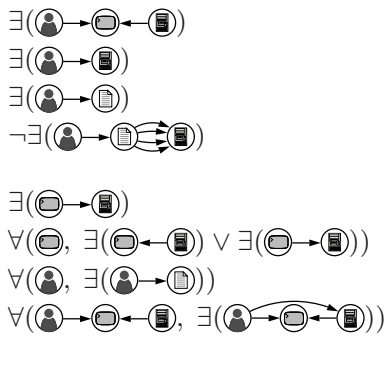
Conditions are nested constraints and application conditions in the sense of [16] generalizing the corresponding notions in [13] along the lines of [30].

Definition 1 (conditions). A *condition* over an object P is of the form $\exists a$ or $\exists(a, c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions [over P] are conditions [over P]. Additionally, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$. A morphism $p\colon P \to G$ *satisfies* a condition $\exists a$ $[\exists(a, c)]$ over P if there exists a morphism $q\colon C \to G$ in $\mathcal{M}$ with $q \circ a = p$ [satisfying c]. An object G *satisfies* a condition $\exists a$ $[\exists(a, c)]$ if all morphisms $p\colon P \to G$ in $\mathcal{M}$ satisfy the condition. The satisfaction of conditions [over P] by objects [by morphisms with domain P] is extended onto Boolean conditions [over P] in the usual way. We write $p \models c$ $[G \models c]$ to denote that morphism p [object G] satisfies c. Two conditions c and c' over P are *equivalent* on objects, denoted by $c \equiv c'$, if, for all objects G, $G \models c$ if and only if $G \models c'$.

We allow infinite conjunctions and disjunctions of conditions. In the context of objects, conditions are also called *constraints*, in the context of rules, they are called *application conditions*. As the required morphisms of the semantics are to be in $\mathcal{M}$, we sometimes speak of $\mathcal{M}$-satisfiability as opposed to $\mathcal{A}$-satisfiability, where $\mathcal{A}$ is the class of all morphisms (see [17]).

Notation. For a morphism $a\colon P \to C$ in a condition, we just depict C, if P can be unambiguously inferred, i.e. for conditions over some left- or right-hand side and for constraints over the $\mathcal{M}$-initial object I. Note, that for every constraint over P, there is an equivalent constraint over I, i.e. $d \equiv \forall(I \to P, d)$, for $d = \exists a$ or $\exists(a, c)$ (see [18]).

Example 2 (access control conditions). Consider the access control graphs introduced in Example 1. Conditions allow to formulate statements on the graphs of the access control and can be combined to form more complex statements. The following conditions are over the empty graph:



$\exists(\ )$	A user is logged into a system.
$\exists(\ )$	A user has an access right to a system.
$\exists(\ )$	A user is connected with a log.
$\neg\exists(\ )$	There are not more than three failed, logged login attempts for any system and any user.
$\exists(\ )$	A session is proposed.
$\forall(\ ,\ \exists(\ ) \vee \exists(\ ))$	Every session is either established or proposed.
$\forall(\ ,\ \exists(\ ))$	Every user is connected with a log.
$\forall(\ ,\ \exists(\ ))$	Every user that is logged into a system, has an access right.

Fig. 2. Conditions on access control graphs

We consider rules with application conditions [13, 16]. Examples and pointers to the literature can be found in [11, 6].

Definition 2 (rules). A *plain rule* $p = \langle L \leftarrow K \rightarrow R \rangle$ consists of two morphisms in $\mathcal{M}$ with a common domain K. L is called the left-hand side, R the right-hand side, and K the interface. An *application condition* $\mathrm{ac} = \langle \mathrm{ac}_L, \mathrm{ac}_R \rangle$ for p consists of two application conditions over L and R, respectively. A *rule* $\hat{p} = \langle p, \mathrm{ac} \rangle$ consists of a plain rule p and an application condition ac for p.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle m^*} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Given a plain rule p and a morphism $K \to D$, a *direct derivation* consists of two pushouts (1) and (2). We write $G \Rightarrow_{p,m,m^*} H$, $G \Rightarrow_p H$, or short $G \Rightarrow H$ and say that m is the *match* and m^* is the *comatch* of p in H. Given a rule $\hat{p} = \langle p, \mathrm{ac} \rangle$ and a morphism $K \to D$, there is a *direct derivation* $G \Rightarrow_{\hat{p},m,m^*} H$ if $G \Rightarrow_{p,m,m^*} H$, $m \models \mathrm{ac}_L$, and $m^* \models \mathrm{ac}_R$. Let $\mathcal{A}$ be the class of all morphisms in $\mathcal{C}$. We distinguish between $\mathcal{A}$*-matching*, i.e. the general case, and $\mathcal{M}$*-matching*, i.e. if the match and the comatch are required to be in $\mathcal{M}$.

Notation. For the category **Graph**, we write $\langle L \Rightarrow R \rangle$ to abbreviate the rule $\langle L \leftarrow K \rightarrow R \rangle$, where K consists of all nodes common to L and R.

Example 3 (access control rules). Consider the access control graphs introduced in Example 1. The rules in Figure 3 are used to formalize the dynamic behavior of the access control system, i.e. are the basis of the access control programs.

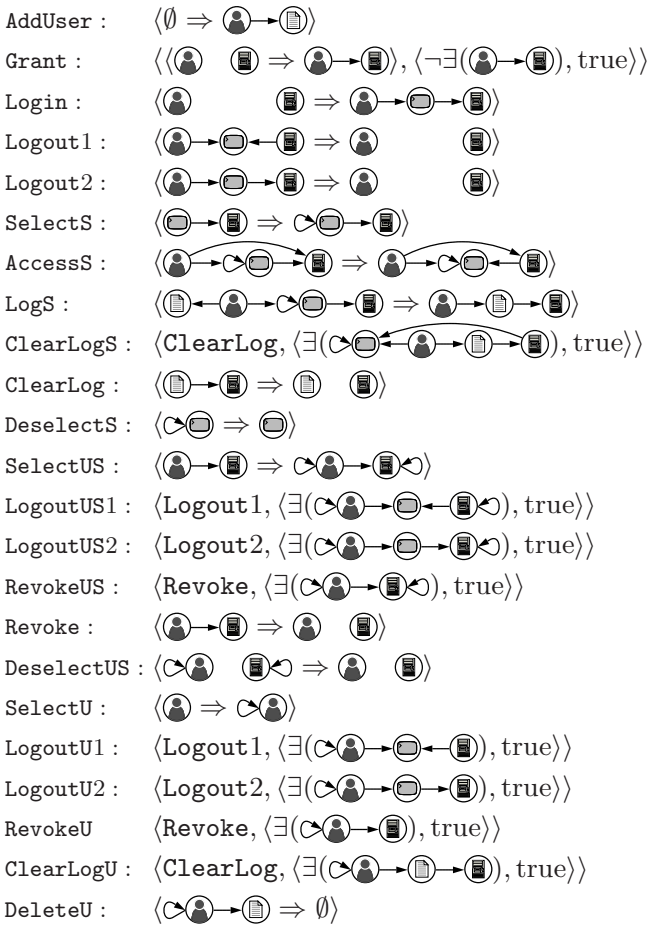


Fig. 3. Rules of the access control system

Note, for every rule, every match is in $\mathcal{M}$. `AddUser` is a plain rule to introduce a user (and the associated log) to the system. `Grant` is a rule with application conditions: It grants a user the right to access a system, unless the user already has access. `Login` models a user proposing a session to a system, while `Logout1` and `Logout2` cancel an established or a proposed session, respectively. Rules with suffix `S`, `US` and `U` concern selected sessions (`S`), user and systems (`US`) and

user (U) and are combined to programs in Figure 4. A description of each rule is given in [18].

We generalize the notions of programs on linear structures [7, 8] and graph programs [19, 27]) to high-level programs on rules.

Definition 3 (programs). *(High-level) Programs* are inductively defined:

(1) Skip and every rule p are programs.
(2) Every finite set $\mathcal{S}$ of programs is a program.
(3) Given programs P and Q, then $(P;Q)$, P^* and $P\downarrow$ are programs.

The *semantics* of a program P is a binary relation $[\![P]\!] \subseteq \mathcal{C} \times \mathcal{C}$ on objects which is inductively defined as follows:

(1) $[\![\texttt{Skip}]\!] = \{\langle G, G\rangle \mid G \in \mathcal{C}\}$ and for every rule p, $[\![p]\!] = \{\langle G, H\rangle \mid G \Rightarrow_p H\}$.
(2) For a finite set $\mathcal{S}$ of programs, $[\![\mathcal{S}]\!] = \cup_{P\in\mathcal{S}}[\![P]\!]$.
(3) For programs P and Q, $[\![(P;Q)]\!] = [\![Q]\!] \circ [\![P]\!]$, $[\![P^*]\!] = [\![P]\!]^*$ and

$$[\![P\downarrow]\!] = \{\langle G, H\rangle \mid \langle G, H\rangle \in [\![P]\!]^* \text{ and } \neg\exists M.\ \langle H, M\rangle \in [\![P]\!]\}.$$

Programs according to (1) are *elementary*; a program according (2) describes the *nondeterministic choice* of a program; a program $(P;Q)$ is the *sequential composition* of P and Q, P^* is the *reflexive, transitive closure* of P, and $P\downarrow$ is the *iteration* of P as long as possible.

Example 4 (access control programs). Consider the access control graphs in Example 1. The dynamic part of the control system $\texttt{Control}^*$ is the reflexive, transitive closure of the programs $\texttt{Control} = \{\texttt{AddUser}, \texttt{Grant}, \texttt{Login}, \texttt{Logout}, \texttt{ProcessLogin}, \texttt{Revoke}, \texttt{DeleteUser}\}$, depicted in Figure 3 and Figure 4, respectively. Logout cancels a session (established or proposed). ProcessLogin models

$$\begin{array}{ll}
\texttt{Logout} & = \{\texttt{Logout1}, \texttt{Logout2}\} \\
\texttt{ProcessLogin} & = \texttt{SelectS};\ \texttt{AccessS}\downarrow;\ \texttt{LogS}\downarrow;\ \texttt{ClearLogS}\downarrow;\ \texttt{DeselectS}\downarrow \\
\texttt{Revoke} & = \texttt{SelectUS};\ \texttt{LogoutUS}\downarrow;\ \texttt{RevokeUS};\ \texttt{DeselectUS} \\
\texttt{LogoutU} & = \{\texttt{LogoutU1}, \texttt{LogoutU2}\} \\
\texttt{DeleteUser} & = \texttt{SelectU};\ \texttt{LogoutU}\downarrow;\ \texttt{RevokeU}\downarrow;\ \texttt{ClearLogU}\downarrow;\ \texttt{DeleteU}
\end{array}$$

Fig. 4. Programs of the access control system

the reaction of a system towards a session proposal, which, dependent on the user's right, leads to an established session and the clearing of the user's log of failed attempts, or the denial and removal of that session and the logging of the failed attempt. Revoke removes a user's right to access a system, but not before closing the user's sessions to that system. Finally, DeleteUser is a program to delete a user and his/her associated log by canceling the user's sessions, by removing the user's access rights and by clearing the user's log.

Note, that there is no way to model certain actions like `DeleteUser` by a single rule, as a user, in principle, may have an arbitrary number of sessions or log entries. However, user deletion should be a transaction always applicable for every user.

Definition 4 (termination). A program P applied to an input object G *terminates* properly, if $\mathrm{PDer}(P,G)$ is finite, i.e. $\exists k \in \mathbb{N}.\ |\mathrm{PDer}(P,G)| \leq k$, where $\mathrm{PDer}(P,G)$ denotes the set of all partial derivations within the execution of a program P, starting with G (see [18]).

Remark 1. Execution of high-level programs requires backtracking, therefore the above definition of termination is more suitable than the classical one, i.e. the nonexistence of infinite derivations. This may be seen as follows: An infinite derivation implies infinitely many partial derivations. The other direction holds only if the number of matches is finite. By the uniqueness of pushouts, $\mathrm{PDer}(p,G)$ is finite and there cannot be infinitely many derivations of finite length for any program P.

3 Basic Transformations of Conditions

In the following, we recall two known transformations from constraints to application conditions and from right- to left application conditions [21, 13, 16] and present a new transformation from application conditions to constraints. Combining these basic transformations, we obtain a transformation from a post-condition over the rule to a precondition. First, there is a transformation from constraints to application conditions such that a morphism satisfies the application condition if and only if the codomain satisfies the constraint.

Theorem 1 (transformation of constraints into application conditions). *There is a transformation* A *such that, for every constraint c and every rule $p = \langle L \leftarrow K \rightarrow R \rangle$, and all morphisms $m^*\colon R \rightarrow H$, $m^* \models \mathrm{A}(p,c) \Leftrightarrow H \models c$.*

Second, there is a transformation from right to left application conditions such that a comatch satisfies an application condition if and only if the match satisfies the transformed application condition.

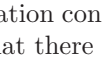
Theorem 2 (transformation of application conditions). *There is a transformation* L *such that, for every rule p, every right application condition* ac *for p, and all direct derivations $G \Rightarrow_{p,m,m^*} H$, $m \models \mathrm{L}(p,\mathrm{ac}) \Leftrightarrow m^* \models \mathrm{ac}$.*

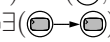
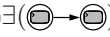
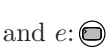
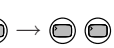
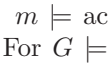
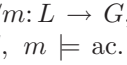
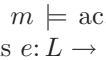
We consider a transformation of application conditions to constraints, which correspond to the universal closure of application conditions. For $\mathcal{A}$-matching however, the closure is over arbitrary morphisms and does not fit to the notion of $\mathcal{M}$-satisfiability. This is why a part of the application condition has to be transformed accordingly.

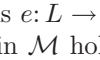
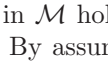
Theorem 3 (transformation of application conditions into constraints). *For weak adhesive HLR categories with $\mathcal{M}$-initial object, there is a transformation* C *such that, for every application condition* ac *over* L *and for all objects* G*,*

$$G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow \forall m\colon L \to G.\ m \models \mathrm{ac}$$

Construction. Define $\mathrm{C}(\mathrm{ac}) = \bigwedge_{e \in E}\ \forall(e \circ i,\ \mathrm{C}_e(\mathrm{ac}))$ where the junction ranges over all epimorphisms $e\colon L \to L'$ and $i\colon I \to L$ is the unique morphism from the $\mathcal{M}$-initial object to L. The transformation C_e is defined inductively on the structure of the conditions: $\mathrm{C}_e(\exists a) = \exists a'$ and $\mathrm{C}_e(\exists(a, c)) = \exists(a', c)$ if $a = a' \circ e$ is some epi-$\mathcal{M}$-factorization of a and $\mathrm{C}_e(\exists a) = \mathrm{C}_e(\exists(a, c)) = \mathrm{false}$ if there is no epi-$\mathcal{M}$-factorization of a with epimorphism e. For Boolean conditions, the transformation C_e is extended in the usual way.

Example 5. The application condition $\mathrm{ac} = \neg\exists($  $) \wedge \neg\exists($  $) \wedge \neg\exists($  $)$ over expresses that there is no edge between two given session nodes.

$$\begin{aligned}
\mathrm{C}(\mathrm{ac}) = \quad & \forall(\text{  },\ \mathrm{C}_{\mathrm{id}}(\mathrm{ac})) \wedge \forall(\text{},\ \mathrm{C}_e(\mathrm{ac})) \\
= \quad & \forall(\text{  },\ \neg\mathrm{C}_{\mathrm{id}}(\exists(\text{  })) \wedge \neg\mathrm{C}_{\mathrm{id}}(\exists(\text{  })) \wedge \neg\mathrm{C}_{\mathrm{id}}(\exists(\text{  }))) \\
& \wedge \forall(\text{  },\ \neg\mathrm{C}_e(\exists(\text{  })) \wedge \neg\mathrm{C}_e(\exists(\text{})) \wedge \neg\mathrm{C}_e(\exists(\text{  }))) \\
= \quad & \forall(\text{  },\ \mathrm{ac}) \wedge \forall(\text{},\ \neg\mathrm{false} \wedge \neg\mathrm{false} \wedge \neg\exists(\text{  })) \\
\equiv \quad & \forall(\text{},\ \neg\exists(\text{  }) \wedge \neg\exists(\text{  }) \wedge \mathrm{true}) \wedge \forall(\text{},\ \mathrm{true} \wedge \neg\exists(\text{})) \\
\equiv \quad & \forall(\text{},\ \neg\exists(\text{}) \wedge \neg\exists(\text{  })) \wedge \forall(\text{  },\ \neg\exists(\text{  }))
\end{aligned}$$

with $\mathrm{id}\colon$  $\to$ and $e\colon$ $\to$  .

Proof. In [17] is shown: For all $m'\colon L' \to G$ in $\mathcal{M}$ and all epimorphisms $e\colon L \to L'$,

$$m' \models \mathrm{C}_e(\mathrm{ac}') \quad \Leftrightarrow \quad m' \circ e \models \mathrm{ac}' \qquad (*)$$

We show: $\forall m\colon L \to G,\ m \models \mathrm{ac}$ if and only if $G \models \mathrm{C}(\mathrm{ac})$. "Only if". Assume $\forall m\colon L \to G,\ m \models \mathrm{ac}$. For $G \models \mathrm{C}(\mathrm{ac})$ to hold, G has to satisfy $\mathrm{C}_e(\mathrm{ac})$ for all epimorphisms $e\colon L \to L'$, i.e. for all epimorphisms $e\colon L \to L'$ and all morphisms $m'\colon L' \to G$ in $\mathcal{M}$ holds $m' \models \mathrm{C}_e(\mathrm{ac})$. Given such morphisms e and m', define $m = m' \circ e$. By assumption, $m \models \mathrm{ac}$, and by $(*)$ we have $m' \models \mathrm{C}_e(\mathrm{ac})$, hence $G \models \mathrm{C}(\mathrm{ac})$. "If". Assume $G \models \mathrm{C}(\mathrm{ac})$, i.e. G satisfies $\mathrm{C}_e(\mathrm{ac})$ for all epimorphisms $e\colon L \to L'$, i.e. for all epimorphisms $e\colon L \to L'$ and all morphisms $m'\colon L' \to G$ in $\mathcal{M}$ holds $m' \models \mathrm{C}_e(\mathrm{ac})$. Given an arbitrary morphism $m\colon L \to G$, consider the epi-$\mathcal{M}$-factorization $m' \circ e$. By assumption, $m' \models \mathrm{C}_e(\mathrm{ac})$, and by $(*)$ we have $m \models \mathrm{ac}$.

Remark 2. The uniqueness of epi-$\mathcal{M}$-factorizations (up to isomorphism) follows immediately from the uniqueness of epi-mono-factorizations, as every $\mathcal{M}$-morphism is a monomorphism.

Remark 3. For weak adhesive HLR categories with $\mathcal{M}$-initial object and $\mathcal{M}$-matching, there is a simplified transformation C such that, for every application condition ac over L and for all objects G, $G \models \mathrm{C}(\mathrm{ac}) \Leftrightarrow \forall m\colon L \to G \in \mathcal{M}.\ m \models \mathrm{ac}$.

For an application condition ac over L and $i\colon I \to L$, let $\mathrm{C}(\mathrm{ac}) = \forall(i, \mathrm{ac})$. For all $\mathcal{M}$-morphisms $m\colon L \to G$, $m \models \mathrm{ac}$ iff there exists an $\mathcal{M}$-morphism $p\colon I \to G$ such that for all $\mathcal{M}$-morphisms $m\colon L \to G$ holds $m \models \mathrm{ac}$ iff there exists an $\mathcal{M}$-morphism $p\colon I \to G$ such that for all $\mathcal{M}$-morphisms $m\colon L \to G$ with $p = m \circ i$ holds $m \models \mathrm{ac}$ iff $G \models \forall(i, \mathrm{ac})$ (Def.1).

Finally, the applicability of a rule can be expressed by a left application condition for the matching morphism.

Theorem 4 (applicability of a rule). *There is a transformation* Def *from rules into application conditions such that, for every rule p and every morphism $m\colon L \to G$,*

$$m \models \mathrm{Def}(p) \Leftrightarrow \exists H.G \Rightarrow_{p,m,m^*} H.$$

Construction. For a rule $p = \langle q, \mathrm{ac}\rangle$, let $\mathrm{Def}(p) = \mathrm{Appl}(q) \wedge \mathrm{ac}_L \wedge \mathrm{L}(p, \mathrm{ac}_R)$, where, for a rule $q = \langle L \xleftarrow{l} K \xrightarrow{r} R\rangle$, $\mathrm{Appl}(q) = \wedge_{a\in A}\neg\exists a$ and the index set A ranges over all morphisms $a\colon L \to L'$ such that the pair $\langle l, a\rangle$ has no pushout complement and there is no decomposition $a = a'' \circ a'$ of a with proper morphism a'' in $\mathcal{M}$ (a'' not an isomorphism) such that $\langle l, a'\rangle$ has no pushout complement.

Example 6. An example of Appl is given below for $\texttt{DeleteSys} = \langle$  $\leftarrow \emptyset \rightarrow \emptyset\rangle$. Intuitively, the application of `DeleteSys` requires the absence of additional edges adjacent to the system node. Therefore, `DeleteSys` may only be the last step in program deleting a system node. Appl(`DeleteSys`) is a condition over  .

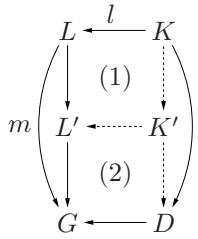
$$\mathrm{Appl}(\texttt{DeleteSys}) = \neg\exists($$$$) \wedge \neg\exists($$$$) \wedge \neg\exists($$$$) \wedge \neg\exists($$$$)$$
$$\wedge \neg\exists($$$$) \wedge \neg\exists($$$$) \wedge \neg\exists($$$$) \wedge \neg\exists($$$$)$$
$$\wedge \neg\exists($$$$)$$

Proof. For plain rules, we show that, for every morphism $m\colon L \to G$,

$$m \models \mathrm{Appl}(q) \Leftrightarrow \exists H.G \Rightarrow_{q,m,m^*} H.$$

"Only if" Let $m \models \mathrm{Appl}(q)$. Assume there is no direct derivation $G \Rightarrow_{q,m,m^*} H$. Then the pair $\langle l, m\rangle$ has no pushout complement and there is a morphism $a\colon L \to L'$ such that $\langle l, a\rangle$ has no pushout complement and $m \models \exists a$. Then $m \not\models \mathrm{Appl}(q)$. A contradiction. Consequently, there is a direct derivation $G \Rightarrow_{q,m,m^*} H$.

"If" Let $G \Rightarrow_{q,m,m^*} H$. Then, for every morphism $a\colon L \to L'$, $m \models \exists a$ iff there is some $m'\colon L' \to G$ in $\mathcal{M}$ such that $m' \circ a = m$. By the pushout-pullback decomposition, the pushout has a decomposition into two pushouts (1) and (2) and, in particular, $\langle l, a\rangle$ has a pushout complement. Consequently, for every morphism $a \in A$, $m \models \neg\exists a$, i.e. $m \models \mathrm{Appl}(q)$.



By the definition of Def and $\models$, Theorem 4, the statement above, and the definition of $\Rightarrow$, for every morphism $m\colon L \to G$, $m \models \mathrm{Def}(p)$ iff $m \models \mathrm{Appl}(q) \wedge m \models \mathrm{ac}_L \wedge m \models \mathrm{L}(p, \mathrm{ac}_R)$ iff $\exists H.G \Rightarrow_{q,m,m^*} H \wedge m \models \mathrm{ac}_L \wedge m^* \models \mathrm{ac}_R$ iff $\exists H.G \Rightarrow_{p,m,m^*} H$. This completes the proof.

4 Weakest Preconditions

In the following, we define weakest preconditions for high-level programs similar to the ones for Dijkstra's guarded commands in [7, 8], show how to construct weakest preconditions for high-level programs and demonstrate the use of weakest preconditions to reduce problems on programs, e.g. the invariance of conditions, onto tautology problems of conditions.

Definition 5 (weakest preconditions). For a program P relative to a condition d we define: A condition c is a *precondition*, if for all objects G satisfying c, (1) $\langle G, H\rangle \in [\![P]\!]$ implies $H \models d$ for all H, (2) $\langle G, H\rangle \in [\![P]\!]$ for some H, and (3) P terminates for G. A precondition c is a *weakest precondition*, denoted by $\mathrm{wp}(P, d)$, if for all other preconditions c' of P relative to d, c' implies c. A condition c is a *liberal precondition*, if for all objects $G \models c$ at least (1) is satisfied, and a *weakest liberal precondition*, denoted by $\mathrm{wlp}(P, d)$, if all other liberal preconditions c' of P relative to d imply c. A condition c is a *termination precondition*, if for all objects $G \models c$ properties (1) and (3) are satisfied, and a *weakest termination precondition*, denoted by $\mathrm{wtp}(P, d)$, if all other termination preconditions c' of P relative to d imply c.

The following fact points out a simple proof scheme for weakest preconditions.

Fact 1 (weakest preconditions). A condition c is a weakest precondition if, for all objects G, $G \models c$ if and only if properties (1)-(3) are satisfied.

For the construction of weakest preconditions, we make use of the fact that $\mathrm{wp}(P, d)$ is a conjunction of three properties and treat properties (1) and (3), and property (2) separately. We observe property (2) is equivalent to the negation of property (1) for $d = \neg\mathrm{true}$, hence we state:

Fact 2 (existence of results). $G \models \neg\mathrm{wlp}(P, \mathrm{false}) \Leftrightarrow$ property (2) is satisfied.

Assumption. We assume that $\langle \mathcal{C}, \mathcal{M}\rangle$ is a weak adhesive HLR category with finite number of matches, i.e. for every morphism $l\colon K \to L$ and every object G, there exist only a finite number of morphisms $m\colon L \to G$ s.t. $\langle l, m\rangle$ has a pushout complement.

Theorem 5 (weakest preconditions). *For weak adhesive HLR categories with finite number of matches, there are transformations* Wlp, Wtp *and* Wp *such that for every program P and every condition d,* $\mathrm{Wlp}(P, d)$ *is a weakest liberal precondition,* $\mathrm{Wtp}(P, d)$ *is a weakest termination precondition and* $\mathrm{Wp}(P, d)$ *is a weakest precondition of P relative to d.*

Construction. The transformations are defined inductively over the structure of programs. For every rule p, let

$$\mathrm{Wlp}(p, d) = \mathrm{Wtp}(p, d) = \mathrm{C}(\mathrm{Def}(p) \Rightarrow \mathrm{L}(p, \mathrm{A}(p, d))).$$

For any program P, $\mathrm{Wp}(P,d) = \mathrm{Wtp}(P,d) \wedge \neg\mathrm{Wlp}(P,\mathrm{false})$.
For any set $\mathcal{S}$ of programs and programs P, Q,

$$\begin{array}{ll}
\mathrm{Wlp}(\texttt{Skip},d) & = d \\
\mathrm{Wlp}(\mathcal{S},d) & = \bigwedge_{P\in\mathcal{S}} \mathrm{Wlp}(P,d) \\
\mathrm{Wlp}((P;Q),d) & = \mathrm{Wlp}(P,\mathrm{Wlp}(Q,d)) \\
\mathrm{Wlp}(P^*,d) & = \bigwedge_{i=0}^{\infty} \mathrm{Wlp}(P^i,d) \\
\mathrm{Wlp}(P{\downarrow},d) & = \mathrm{Wlp}(P^*,\mathrm{Wlp}(P,\mathrm{false}) \Rightarrow d)
\end{array}$$

$$\begin{array}{ll}
\mathrm{Wtp}(\texttt{Skip},d) & = d \\
\mathrm{Wtp}(\mathcal{S},d) & = \bigwedge_{P\in\mathcal{S}} \mathrm{Wtp}(P,d) \\
\mathrm{Wtp}((P;Q),d) & = \mathrm{Wtp}(P,\mathrm{Wtp}(Q,d)) \\
\mathrm{Wtp}(P^*,d) & = \bigwedge_{i=0}^{\infty} \mathrm{Wlp}(P^i, d \wedge \mathrm{Wtp}(P,\mathrm{true})) \wedge \bigvee_{k=0}^{\infty} \mathrm{Wlp}(P^{k+1},\mathrm{false}) \\
\mathrm{Wtp}(P{\downarrow},d) & = \mathrm{Wtp}(P^*,\mathrm{Wlp}(P,\mathrm{false}) \Rightarrow d)
\end{array}$$

where for $i \geq 0$, P^i is inductively defined by $\texttt{Skip}$ for $i = 0$ and by $P^{i+1} = (P^i; P)$.

Proof. We show $\mathrm{Wlp}(P,d) \equiv \mathrm{wlp}(P,d)$, $\mathrm{Wtp}(P,d) \equiv \mathrm{wtp}(P,d)$, and $\mathrm{Wp}(P,d) \equiv \mathrm{wp}(P,d)$. The first two proofs are done by induction over the structure of programs. First we consider elementary programs consisting of a single rule p. For all objects G, we have:

$$\begin{array}{lr}
\quad G \models \mathrm{Wlp}(p,d) & \\
\Leftrightarrow G \models \mathrm{C}(\mathrm{Def}(p) \Rightarrow \mathrm{L}(p,\mathrm{A}(p,d))) & \text{(Def. Wlp)} \\
\Leftrightarrow \forall L \overset{m}{\to} G.\ m \models (\mathrm{Def}(p) \Rightarrow \mathrm{L}(p,\mathrm{A}(p,d))) & \text{(Thm. 3)} \\
\Leftrightarrow \forall L \overset{m}{\to} G.\ m \models \mathrm{Def}(p) \Rightarrow m \models \mathrm{L}(p,\mathrm{A}(p,d)) & \text{(Def. } \models\text{)} \\
\Leftrightarrow \forall L \overset{m}{\to} G, R \overset{m^*}{\to} H.\ m \models \mathrm{Def}(p) \Rightarrow m^* \models \mathrm{A}(p,d) & \text{(Thm. 2)} \\
\Leftrightarrow \forall L \overset{m}{\to} G, R \overset{m^*}{\to} H.\ (G \Rightarrow_{p,m,m^*} H) \Rightarrow H \models d & \text{(Thms. 4 \& 1)} \\
\Leftrightarrow \forall H.\ \langle G,H\rangle \in [\![p]\!] \Rightarrow H \models d & \text{(Def. } [\![p]\!]\text{)} \\
\Leftrightarrow G \models \mathrm{wlp}(p,d) & \text{(Def. wlp)}
\end{array}$$

Thus, $\mathrm{Wlp}(p,d)$ is a weakest liberal precondition of p relative to d. Furthermore, $G \models \mathrm{Wtp}(p,d)$ if and only if $G \models \mathrm{Wlp}(p,d)$, as every rule application terminates by the finiteness assumption and wtp reduces to wlp for single rules p. For composed programs, the statement follows by structural induction (see [18].)

For Wp, we now show for every program P, $\mathrm{Wp}(P,d) \equiv \mathrm{wp}(P,d)$: $\mathrm{Wp}(P,d)$ is defined as $\neg\mathrm{Wlp}(P,\mathrm{false}) \wedge \mathrm{Wtp}(P,d)$, which is, by the first two equations, equivalent to $\neg\mathrm{wlp}(P,\mathrm{false}) \wedge \mathrm{wtp}(P,d)$, which is equivalent to $\mathrm{wp}(P,d)$ (see [18].)

Example 7 (access control system). Consider the access control for computer systems, presented in Examples 1-4. For the system, one might want to ensure the validity of certain properties, e.g.:

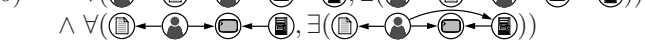
(1) Always, every user logged into a system, has an access right to the system: *secure* implies $\mathrm{wlp}(\texttt{Control}, \textit{secure})$, where

$$\textit{secure} = \forall(\text{}, \exists(\text{  })).$$

(2) Every user can always be deleted: $\exists($$)$ implies wp(`DeleteUser`, true)
(3) Every user can always have his access right to a system revoked:
$\exists($  $)$ implies wp(`Revoke`, true)

By calculating weakest [liberal] preconditions, the problem to decide these properties can be reduced onto the tautology problem for conditions. The meaning of *secure* implies wlp(`Control`, *secure*) can be seen as follows: The constraint *secure* is an invariant, i.e. given a state satisfying *secure*, every next state will also satisfy *secure*. For a proof, we have to show *secure* implies Wlp(P, *secure*) for every program $P \in$ `Control`.

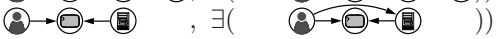
We give explicit proof of property (1) for the programs `AddUser` and `Grant`. For the program `AddUser`, *secure* implies Wlp(`AddUser`, *secure*), which can be proved as follows:

$\mathrm{A}($`AddUser`$, \mathit{secure}) =$ $\forall($  $)$
$\wedge\ \forall($  $)$

$\mathrm{L}($`AddUser`$, \mathrm{A}($`AddUser`$, \mathit{secure})) =$ $\forall($  $) = \mathit{secure}$

$$\begin{aligned}\mathrm{Wlp}(\texttt{AddUser}, \mathit{secure}) &= \mathrm{C}(\mathrm{Def}(\texttt{AddUser}) \Rightarrow \mathrm{L}(\texttt{AddUser}, \mathrm{A}(\texttt{AddUser}, \mathit{secure})))\\ &= \mathrm{C}((\mathrm{Appl}(\texttt{AddUser}) \wedge \mathrm{true} \wedge \mathrm{true}) \Rightarrow \mathit{secure})\\ &\equiv \mathrm{C}(\mathrm{true} \Rightarrow \mathit{secure}) \equiv \mathrm{C}(\mathit{secure})\\ &= \forall(\emptyset, \mathit{secure})\\ &\equiv \mathit{secure}\end{aligned}$$

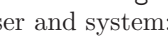
This is no surprise as we could also have argued that a newly added user cannot have an established session with a system, hence every application of `AddUser` preserves the satisfaction of *secure*. For the program `Grant`, *secure* implies Wlp(`Grant`, *secure*), even without the additional application condition.

$\mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure}))$
$=$ $\forall($  $)$
$\wedge\ \forall($  $)$
$\wedge\ \forall($  $)$
$\wedge\ \forall($  $)$

$\mathrm{Wlp}($`Grant`$, \mathit{secure})$
$= \mathrm{C}(\mathrm{Def}($`Grant`$) \Rightarrow \mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure})))$
$= \mathrm{C}((\mathrm{Appl}($`Grant`$) \wedge \neg\exists($  $) \wedge \mathrm{true}) \Rightarrow \mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure})))$
$\equiv \mathrm{C}(\neg\exists($$) \Rightarrow \mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure})))$
if $\mathrm{C}(\mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure})))$
if $\mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure}))$

Note, *secure* implies $\mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure}))$ and thus Wlp(`Grant`, *secure*). We also have *secure* implies Wlp(`Login`, *secure*), the proof of which is similar to *secure* implies Wlp(`Grant`, *secure*), as $\mathrm{L}($`Grant`$, \mathrm{A}($`Grant`$, \mathit{secure})) \equiv \mathrm{L}($`Login`$, \mathrm{A}($`Login`$, \mathit{secure}))$.

For `Logout`, `ProcessLogin`, `Revoke` and `DeleteUser`, we shall only sketch the proofs. It is easy to see that `Logout1` as well as `Logout2` preserve the satisfaction of *secure*, hence we can assume *secure* implies Wlp(`Logout`, *secure*). Concerning `ProcessLogin`, one can prove the invariance of *secure* for every used rule, i.e. `SelectS`, `AccessS`, `LogS`, `ClearLogS` and `DeselectS`, and moreover, for every subprogram of `ProcessLogin`. Intuitively the only interesting part is the proof of *secure* implies Wlp(`AccessS`, *secure*), while the validity of this claim is quite obvious. Concerning `Revoke`, one can show that `LogoutUS`$\downarrow$ leaves no sessions for any selected user and system (see property (4)). As a consequence, `RevokeUS` will preserve the satisfaction of *secure*, as do all other parts of `Revoke`, hence *secure* implies Wlp(`Revoke`, *secure*). The proof of `DeleteUser` is similar.

(4) After execution of `LogoutUS`$\downarrow$, there is no established session left for any selected user and system: wlp(`LogoutUS`$\downarrow$, $\neg\exists($  $))\equiv$ true.
(5) C(Appl(`Logout1`)) $\wedge$ C(Appl(`Logout2`)) is an invariant for all programs P in `Control`, and all subprograms and rules of `Revoke` and `DeleteUser`.

One can show property (4) by using (5) as one observes Wlp(`LogoutUS`$\downarrow$, *secure*) $\equiv$ Wlp(`LogoutU`*, true) $\equiv$ true. Property (5) expresses that certain edges adjacent to a session node do not exist, while others have a multiplicity of at most 1. Proving property (5) for all rules used in `Control` is tedious, but nonetheless straightforward, as every subcondition may handled separately. Intuitively only subprograms and rules have to be considered that contain a session node, and moreover, that create or delete edges adjacent to session nodes.

5 Related concepts

In this section we briefly review other work on using graph transformation for verification. Before we do so, however, we wish to point out one important global difference between this related work and the approach of this paper.

- The approach of this paper is based on the principle of *assertional reasoning*, and inherits both the advantage and the disadvantage of that principle. The advantage is that the approach is general where it can be made to apply, meaning that it provides a method to verify finite-state and infinite-state systems alike. The disadvantage is that finding invariants is hard and cannot be automated in general.
- Existing approaches are typically based on the principle of *model checking*, which essentially involves exhaustive exploration, either of the concrete states (which are often too numerous to cover completely) or on some level of abstraction (in which case the results become either unsound or incomplete). On the positive side, model checking is a push-button approach, meaning that it requires no human intervention.

In other words, there is a dividing line between the work in this paper and the related work reported below, which is parallel to the division between theorem

proving and model checking in "mainstream" verification (see [20] for an early discussion). Since current wisdom holds that these approaches can actually be combined to join strengths (e.g., [5, 25]), we expect that the same will turn out to hold in the context of graph transformation.

The first paper in which it was put forward that graph transformation systems can serve as a suitable specification formalism on the basis of which model checking can be performed was Varró [32]; this was followed up by [33] which describes a tool chain by which graph transformation systems are translated to Promela, and then model checked by SPIN. We pursued a similar approach independently in [28, 29], though relying on dedicated (graph transformation-based) state space generation rather than an existing tool. The two strands were compared in [31]. Again independently, Dotti et al. [10, 9] also describe a translation from a graph transformation-based specification formalism (which they call object-based graph grammars) to Promela.

Another model checking-related approach, based on the idea of McMillan unfoldings for Petri Nets (see [24]), has been pursued by Baldan, König et al. in, e.g., [2, 1], and in combination with abstraction in [3, 23]. The latter avoids the generation of complete (concrete) state spaces, at the price of being approximative, in other words, admitting either false positives (unsoundness) or false negatives (incompleteness) in the analysis. The (pure) model checking and abstraction-based techniques were briefly compared in [4].

Termination. In addition to the general verification methods discussed above, a lot of research has been carried out on more specific properties of graph grammars. Especially relevant in our context is the work on *termination* of graph grammars. This is known to be undecidable in general (see [26]), but under special circumstances may be shown to hold; for instance, Ehrig et al. discuss such a special case for *model transformation* in [12].

6 Conclusion

This paper extends graph programs to programs over high-level rules with application conditions, and defines weakest preconditions over high-level programs similar to the ones for Dijkstra's guarded commands in [7, 8]. It presents transformations from application conditions to constraints, which, combined with two known transformations over constraints and application conditions, can be used to construct weakest preconditions for high-level rules as well as programs.

A known proof technique for showing the correctness of a program with respect to a pre- and a postcondition is to construct a weakest precondition and to show that the precondition implies the weakest precondition. We demonstrate the applicability of this method on our access control for computer systems.

Further topics could be the followings.
(1) Consideration of strongest postconditions.
(2) Comparison of notions: A comparison of conditions – as considered in this paper – and first-order formulas on graphs and high-level structures.

(3) Generalization of notions: The generalization of conditions to capture monadic second order properties.
(4) An investigation of the tautology problem for conditions with the aim to find a suitable class of conditions, for which the problem is decidable.
(5) Implementation: A system for computing/approximating weakest preconditions and for deciding/semideciding correctness of program specifications.

References

1. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars. In *Concurrency Theory*, volume 3170 of *LNCS*, pages 83–98. Springer, 2004.
2. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Graph Transformations (ICGT'02)*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
3. P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In *Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 255–272. Springer, 2003.
4. P. Baldan, B. König, and A. Rensink. Graph grammar verification through abstraction. In B. König, U. Montanari, and P. Gardner, editors, *Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*, number 04241 in Dagstuhl Seminar Proceedings, 2005.
5. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
6. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation. In *Handbook of Graph Grammars and Computing by Graph Trans.*, volume 1, pages 163–245. World Scientific, 1997.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
8. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1989.
9. O. M. dos Santos, F. L. Dotti, and L. Ribeiro. Verifying object-based graph grammars. *ENTCS*, 109:125–136, 2004.
10. F. L. Dotti, L. Foss, L. Ribeiro, and O. M. dos Santos. Verification of distributed object-based systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 2884 of *LNCS*, pages 261–275. Springer, 2003.
11. H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1979.
12. H. Ehrig, K. Ehrig, J. De Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 214–228. Springer, 2005.
13. H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 72, 2006.
14. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer-Verlag, Berlin, 2006.
15. H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundamenta Informaticae*, 72, 2006.

16. A. Habel and K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and System Modeling*, volume 3393 of *LNCS*, pages 293–308. Springer, 2005.
17. A. Habel and K.-H. Pennemann. Satisfiability of high-level conditions. In *Graph Transformations (ICGT'06)*, this volume of LNCS. Springer, 2006.
18. A. Habel, K.-H. Pennemann, and A. Rensink. Weakest preconditions for high-level programs: Long version. Technical Report 8/06, University of Oldenburg, 2006.
19. A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures*, volume 2030 of *LNCS*, pages 230–245. Springer, 2001.
20. J. Y. Halpern and M. Y. Vardi. Model checking vs. theorem proving: A manifesto. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proc. International Conference on Principles of Knowledge Representation and Reasoning*, pages 325–334. Morgan Kaufmann Publishers, 1991.
21. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars. In *SEGRAGRA'95*, volume 2 of *ENTCS*, pages 95–104, 1995.
22. M. Koch, L. V. Mancini, and F. Parisi-Presicce. Graph-based specification of access control policies. *Journal of Computer and System Sciences (JCSS)*, 71:1–33, 2005.
23. B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 197–211. Springer, 2006.
24. K. L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Fourth Workshop on Computer-Aided Verification (CAV)*, volume 663 of *LNCS*, pages 164–174. Springer, 1992.
25. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.
26. D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998.
27. D. Plump and S. Steinert. Towards graph programs for graph algorithms. In *Graph Transformations (ICGT'04)*, volume 3256 of *LNCS*, pages 128–143. Springer, 2004.
28. A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE-TR-2003-2, pages 150–160. University of Southhampton, 2003.
29. A. Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, page 485. Springer, 2004.
30. A. Rensink. Representing first-order logic by graphs. In *Graph Transformations (ICGT'04)*, volume 3256 of *LNCS*, pages 319–335. Springer, 2004.
31. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Graph Transformations (ICGT'04)*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.
32. D. Varró. Towards symbolic analysis of visual modeling languages. *ENTCS*, 72(3), 2003.
33. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 3(2):85–113, 2004.