ENFORCe: A System for Ensuring Formal Correctness of High-level Programs

Karl Azab, Annegret Habel, Karl-Heinz Pennemann and Christian Zuckschwerdt*
*Carl v. Ossietzky Universität Oldenburg, Germany
{azab,habel,pennemann,zuckschwerdt}@informatik.uni-oldenburg.de

Abstract. *Graph programs allow a visual description of programs on graphs and graph-like structures. The correctness of a graph program with respect to a pre- and a postcondition can be shown in a classical way by constructing a weakest precondition of the program relative to the postcondition and checking whether the precondition implies the weakest precondition. ENFORCe is a currently developed system for ensuring formal correctness of graph programs and, more general, high-level programs by computing weakest preconditions of these programs. In this paper, we outline the features of the system and present its software framework.*

Keywords: high-level programs, correctness, formal verification, weakest preconditions, weak adhesive HLR categories.

1 Introduction

Graph transformation has many application areas in computer science, such as software engineering or the design of concurrent and distributed systems. It is a visual modeling technique and plays a decisive role in the development of growingly larger and complex systems. However, the use of visual modeling techniques alone does not guarantee the correctness of a design. In context of rising standards for trustworthy systems, there is a growing need for the verification of graph transformation systems. Therefore, tools supporting formal verification of graph transformations will increase the attractiveness of this modeling technique and are in this sense important for its practical application.

There exist several tools specifically concerned with graph transformation: Engines for plain transformation, e.g., [Bus04, GBG$^+$06, MP06], general purpose tools with visual editors and debuggers for transformation systems like [Tae04, SWZ99, BGN$^+$04], and tools concerned with model checking or analysis of transformation systems properties, e.g., [Tae04, KK06, SV03, KR06, BBG$^+$06].

Until now, most of these tools focus on transformation systems, instead of rule-based programs. Programs featuring at least sequential composition and iteration are Turing-complete and necessary to model transactions when dealing with an arbitrary number of elements. Moreover, most tools are specifically concerned with a distinct kind of structure, let it be simple labeled, (typed) attributed graphs or hyper-

graphs. From a theoretical point of view, weak adhesive HLR categories [EEPT06] are an important effort to build a unified theory for transformation systems covering several kinds of structures, e.g., various kinds of (hyper-)graphs, place-transition nets and algebraic specifications. Unfortunately, there do not exist tools designed to follow that idea, i.e., whose algorithms will work for more than just a specific kind of structure.

In this paper, we will present the main ideas of ENFORCe, a suite of tools for ensuring the correctness of high-level programs. It is designed for weak adhesive HLR categories, exploiting the fact that necessary high-level algorithms can be based on a small set of structure-specific methods. Structurally, ENFORCe consists of *Applications* (e.g., user interface), *Correctness Tools* and *Transformations* (e.g., for proving the correctness of a program), *Engines* (i.e., specific data structures and methods) and a *Core* containing general high-level notions and methods, connecting these components. We plan to reuse existing engines, like GRAJ. Our efforts aim for a tool supplementary to existing tools such as [Tae04, KK06, KR06, BBG+06], i.e., in terms of structures or functionality (see related systems).

The paper is organized as follows. In Section 2, we introduce programs for high-level structures like graphs and algebraic specifications and present a method for showing correctness for high-level programs. In Sections 3 and 4, we present the system requirements and the system design. In Section 5, we give an overview on related systems. A conclusion including further work is given in Section 6.

2 Correctness of Programs

In this section, we give an informal introduction to the main concepts of the paper, in particular into correctness of high-level programs based on all kinds of high-level structures such as graphs, place-transition nets, and algebraic specifications. The concepts are illustrated by a running example in the category of graphs. For more details refer to [EEPT06, HPR06].

Assumption. We assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with a decidable set $\mathcal{M}$, binary coproducts, epi-$\mathcal{M}$-factorization, an $\mathcal{M}$-*initial* object, i.e., there is an object I such that, for every object G in $\mathcal{C}$, there exists a unique morphism from I to G in $\mathcal{M}$, and a finite number of matches for each object, i.e., for every morphism $l\colon K \to L$ in $\mathcal{M}$ and every object G, there exist only a finite number of morphisms $m\colon L \to G$ such that $\langle l, m \rangle$ has a pushout complement.

Example 1 (access control graphs). For illustration, we consider the weak adhesive HLR category of all directed labeled graphs. We consider a simple access control for computer systems, which abstracts authentication and models user and session management in a simple way. The basic items are users, sessions, and computer systems with directed edges between them. 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 session is connected to a user and a system. The direction of the latter edge differentiates between proposed and established sessions, i.e., an edge from a session node to a system in the first case and a reversed edge in the latter. 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. The complete example is published in [HPR06].

We use a graphical notion of conditions to specify valid system and program states, as well as morphism.

Figure 1: A state graph of the access control system

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.

In the context of objects, conditions are also called *constraints*, in the context of rules, they are called *application conditions*.

Example 2 (access control conditions). The condition $nosession = \neg\exists(\emptyset \to$  $)$ over the empty graph expresses that a selected user shall not have an established session, and the condition $nouser = \neg\exists(\emptyset \to$  $)$ means that no user is selected.

Transformation rules form the elementary steps of our computing model.

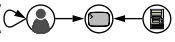
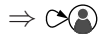
Definition 2 (rules). A *rule* consists of a *plain rule* $p = \langle L \leftarrow K \rightarrow R\rangle$, shortly denoted by $\langle L \Rightarrow R\rangle$, and a pair $\langle \mathrm{ac}_L, \mathrm{ac}_R\rangle$ of conditions over L and R, respectively. L is called the left-hand side, R the right-hand side, and K the interface. The conditions ac_L,ac_R are called the *left* and *right application condition* of p.

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

A *direct derivation* through a plain rule p 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. A *direct derivation* $G \Rightarrow_{\hat{p},m,m^*} H$ through a rule is a direct derivation $G \Rightarrow_{p,m,m^*} H$ through the underlying plain rule such that the match m satisfies the left application condition ac_L and the comatch m^* satisfies the right application condition ac_R.

Example 3 (access control rules). The rule `SelectU` selects a user and the rule `LogoutU1` cancels an established session of a selected user.

`SelectU`: $\langle$  $\Rightarrow$  $\rangle$

`LogoutU1`: $\langle$  $\Rightarrow$  $\rangle$

Sequential composition and iteration give rise to rule-based programs.

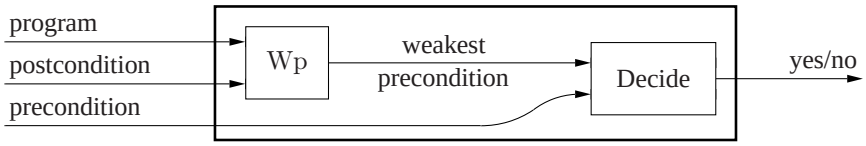
Definition 3 (programs). *Programs* are inductively defined: $\texttt{Skip}$ and every rule p are programs. Every finite set $\mathcal{S}$ of programs is a program. Given programs P and Q, then the sequential composition $(P;Q)$, the reflexive, transitive closure P^* and the as long as possible iteration $P\downarrow$ are programs. The *semantics* of a program P is a binary relation on $\mathcal{C}$. Programs of the form $(P;(Q;R))$ and $((P;Q);R)$ are considered as equal; by convention, both can be written as $P;Q;R$.

Example 4 (access control program). The program $\texttt{Logout} = \texttt{SelectU};\texttt{LogoutU1}\downarrow$ selects a user and closes all of his established sessions.

Definition 4 (correctness). A program P with respect to a pre- and a postcondition is *correct* if, for all objects G satisfying the precondition holds: H satisfies the postcondition for every pair $\langle G, H\rangle$ in the semantics of P, there is some pair $\langle G, H\rangle$ in the semantics of P, and the program P terminates for G.

Concerning correctness, we are considering the following strategies:

Correctness by proof. A well-known method for showing the total correctness of a program with respect to a pre- and a postcondition is to construct a weakest precondition (Wp) of the program relative to the postcondition and to prove that the precondition implies the weakest precondition. For partial correctness, it suffices to consider weakest liberal preconditions (Wlp).



In [HPR06], we consider weakest preconditions for high-level programs similar to the ones for Dijkstra's guarded commands and show how to construct weakest preconditions for programs on weak adhesive HLR categories with a finite number of matches. In case of rules, the construction of a weakest precondition makes use of two known transformations [HW95, EEHP06, HP05] from constraints to right application conditions, and from right to left application conditions, and additionally, a new transformation from application conditions to constraints [HPR06].

However, this method requires an algorithm for the implication problem for conditions, which may be able to decide the problem for a suitable class of conditions, and approximate the decision in the general case. Moreover, the construction of weakest preconditions for programs with iteration relies on invariants, which in the general case requires an approximation or user intervention.

Example 5 (correctness by proof). Consider the program $\texttt{LogoutUser}$ of Example 4 and the conditions in Example 2. One might verify the partial correctness of $\texttt{LogoutUser}$ with respect to the precondition $nouser$ and the postcondition $nosession$. According to [HPR06], we construct the weakest liberal precondition $\mathrm{Wlp}(\texttt{LogoutUser},\ nosession) = \mathrm{Wlp}((\texttt{SelectU};\texttt{LogoutU1}\downarrow),\ nosession) = \mathrm{Wlp}(\texttt{SelectU}, \mathrm{Wlp}(\texttt{LogoutU1}\downarrow,\ nosession))$. One has to show that $\mathrm{Wlp}(\texttt{LogoutU1}\downarrow,\ nosession) = \mathrm{Wlp}(\texttt{LogoutU1}^*, \mathrm{Wlp}(\texttt{LogoutU1}, \mathrm{false}) \Rightarrow nosession) = \mathrm{Wlp}(\texttt{LogoutU1}^*,\ \forall(\emptyset \rightarrow$ ),

$\neg\mathrm{Appl}(\texttt{LogoutU1})) \Rightarrow \mathit{nosession})$ if $\mathrm{Wlp}(\texttt{LogoutU1}^*, \neg\exists(\emptyset \to$ $) \Rightarrow \mathit{nosession})$ equivalent to true, hence $\mathrm{Wlp}(\texttt{LogoutU}, \mathit{nosession})$ equivalent to true. Obviously nouser implies true, hence `LogoutUser` is correct with respect to the given conditions. For more examples, we refer to the long version of [HPR06].

Correctness by transformation. Given a program with pre- and postcondition, a correct program is derived from the input program by minimal semantical restrictions. The main idea is to insert assertions in form of applications conditions into rules within iterations of the program to enforce the invariance of postconditions. The construction is based on the integration of constraints into application conditions of rules. It makes use of the two known transformations from constraints to right application conditions (A), and from right to left application conditions (L) [HW95, HP05]. The idea of this approach is summarized below.

Example 6 (Correctness by transformation). Considering the postcondition $\mathit{nosession}$, the program $\texttt{LogoutUser} = \texttt{SelectU}; \texttt{LogoutU1}\downarrow$ is transformed into a partial correct program $\texttt{LogoutUser}' = \texttt{Assert}(c); \texttt{SelectU}; \langle \texttt{LogoutU1}, \langle \mathrm{ac}, \mathrm{true}\rangle\rangle^*$, where constraint $c = \mathrm{Wlp}(\texttt{SelectU}, (\mathrm{Wlp}(P, \mathrm{false}) \Rightarrow \mathit{nosession}))$, application condition $\mathrm{ac} = \mathrm{L}(\texttt{LogoutU1}, \mathrm{A}(\texttt{LogoutU1}, (\mathrm{Wlp}(P, \mathrm{false}) \Rightarrow \mathit{nosession})))$, and $\texttt{Assert}(c) = \langle\langle I \Rightarrow I\rangle, \langle c, \mathrm{true}\rangle\rangle$ for any condition c over the $\mathcal{M}$-initial object I. As observed in Example 5, $((\mathrm{Wlp}(P, \mathrm{false}) \Rightarrow \mathit{nosession})$ is equivalent to true. A subsequent optimization step may be able to eliminate some superfluous application conditions.

The strategies for ensuring correctness base on certain high-level transformations (see Table 1) such as the transformations from constraints to right application conditions and from right to left application conditions. In a concrete weak adhesive HLR category, high-level transformations may be applied by

Symbol	Description	Reference
A	From constraints to application conditions	[HW95, HP05]
L	From right to left application conditions	[HW95, HP05]
C	From application conditions to constraints	[HPR06]
	⋮	

Table 1: High-level transformations

using a small set of elementary, structure-specific operations (see Table 2) such as the constructions of pushouts and pushout complements, the set of all epimorphisms with a given domain G, the composition of two morphisms, and the $\mathcal{M}$-test for morphisms.

Symbol	Description
PO	Construct a pushout along $\mathcal{M}$-morphisms
POC	Construct a pushout complement of two morphisms, if possible
$=$	Check commutativity of two morphisms
$\circ$	Construct the composition of two morphisms
initial	Construct morphism from initial object to input object
matches	Find all $\mathcal{M}$-matchings of one object in another
epi$\mathcal{M}$	Construct an epi-$\mathcal{M}$-factorization of a morphism
epimorphisms	Construct all epimorphisms with a given domain (up to iso.)
is$\mathcal{M}$? [isEpi?]	Is the given morphism an $\mathcal{M}$-morphism [epimorphism]?
	$\vdots$

Table 2: Structure-specific operations

3 System Requirements

The software framework should work on high-level programs, i.e., programs on high-level structures like graphs, place-transition nets, and algebraic specifications. For program specifications, i.e., programs with pre- and postconditions, there should be tools for correctness by proof and correctness by transformation. For the correctness strategies we identify a chain of algorithmic dependencies, see Figure 2. In the figure, we exclude standard tools, e.g., checking whether a given object satisfies a given condition. The dependencies are organized in three layers; the correctness strategies (correctness tools) depending on high-level transformations of conditions that in turn depend on elementary structure-specific operations. For one transformation system working on graphs and for another on Petri-nets, the structure-specific operations differ but the algorithms for transformation of conditions and the correctness tools remain the same. From a software engineering point of view, the components modeling correctness algorithms and weak adhesive HLR categories should therefore be loosely coupled and have as few dependencies on each other as possible. This ensures that the system can be easily extended with new weak adhesive HLR categories and high-level algorithms.

4 System Design

This section describes the basic software components of the ENFORCe framework. Basically, the system consists of five components: *Engines* represents specific weak adhesive HLR categories, the *Core* evaluates conditions and connects Engines with the third component, *Transformations*, that contain algorithms transforming conditions, and the *Application* uses the four previous components to calculate the correctness results its user has requested. The components and their static dependencies are illustrated in Figure 3 (a).

Engines. An Engine is the combination of the structural implementation of a weak adhesive HLR category with a category specific implementation of the operations listed in Table 2. E.g. `GraphEngine`,

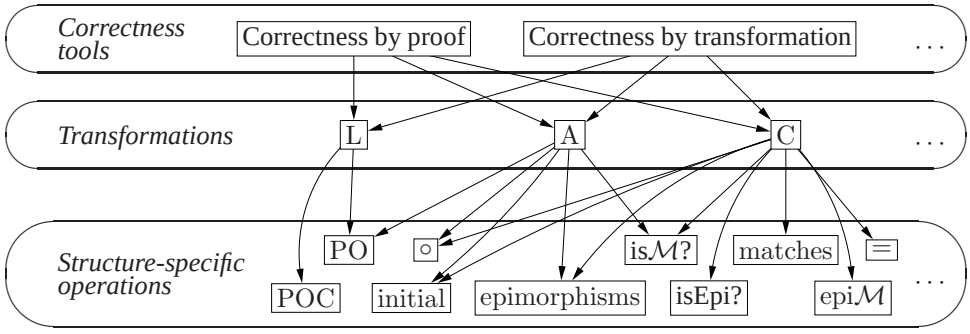


Figure 2: Levels in ENFORCe

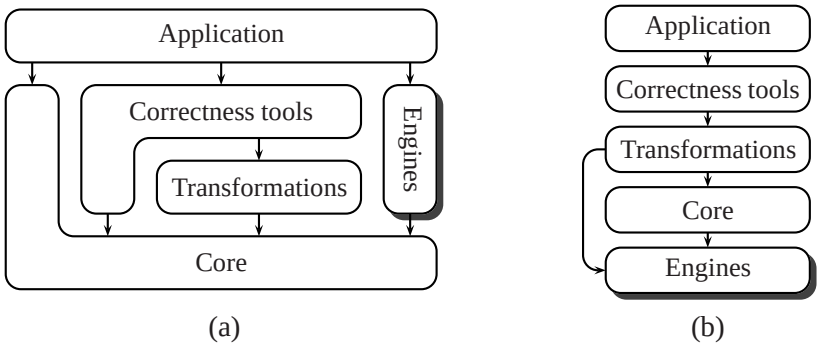


Figure 3: (a) Static dependencies and (b) runtime data flow

contains the data structures for directed labeled graphs and graph morphisms as well as the algorithms working exclusively on these structures. As ENFORCe may have several Engines the Engine component is shown with a shadow. The Core and Transformations can use different (and new) Engines without having to be modified or updated.

Correctness tools and Transformations. These two components contain algorithms operating exclusively on weak adhesive HLR categories and can therefore be abstracted from the actual category in question. An example of algorithms working at this level is the chain of transformations from constraints to right- to left application conditions to weakest preconditions. Pseudo code for the transformation from constraints to application conditions is shown in Figure 4. Correctness tools and Transformations works on conditions, explaining their static dependency on the Core.

Core. The Core consists of two important parts: One contains data structures for programs and conditions. It also evaluates conditions with the help of operations in Engines. The other part channels and controls the communication between Transformations and individual Engines at runtime. To facilitate communication, the Core provides an interface for Engine plug-ins and works as a dependency injector, explaining the runtime connection between Transformations and Engines.

Although of secondary concern, the Core can execute high-level programs. Most necessary parts are already implemented for other functionality: Data structures modeling programs and conditions, evaluation of conditions, and the matching and pushout operation in the Engines.

```
Data: Rule r, Condition c
Result: the transformed result in c

R := r.rightHandSide
P := c.morphism.domain
A := createTupleSet (initial (R), initial (P), false)
if c is Universal then
    j := new Disjunction
    foreach (s, p) in A do
        j += new Existential(s, subroutine (p, c))
    end
    c := j
else if c is Existential then
    j := new Conjunction
    foreach (s, p) in A do
        j += new Universal(s, subroutine (p, c))
    end
    c := j
else
    foreach c1 in c.children do
        subroutine (p, c1)
    end
end
```

Algorithm 1: transformation A

```
Data: Morphism p, Morphism x, Boolean check_u
Result: the set of morphism tuples to a common codomain, in A

A := new Set()
t, q := pushout (p, x)
E := epimorphisms (t.codomain)
foreach e in E do
    r := compose (q, e) // e o q
    if r in M then
        u := compose (t, e) // e o t
        if not check_u or u in M then
            A += (u, r)
        end
    end
end
```

Algorithm 2: createTupleSet

```
Data: Morphism p, Condition c
Result: the transformed result in c

if c is Existential or c is Universal then
    A := createTupleSet (p, c.morphism, true)
    if c is Existential then
        j := new Disjunction
        foreach (u, r) in A do
            if c is basic then
                j += new Existential(u)
            else
                j += new Existential(u, subroutine (r, c.child))
            end
        end
    else // c is Universal
        j := new Conjunction
        foreach (u, r) in A do
            j += new Universal(u, subroutine (r, c.child))
        end
    end
    c := j
else // c is a boolean constraint
    foreach c1 in c.children do
        subroutine (p, c1)
    end
end
```

Algorithm 3: subroutine

Figure 4: Pseudo code for the transformation A from constraint to right application condition

Application. This is the action initiating component of the system. The runtime data flow between the components is shown in Figure 3 (b). The Application contacts the Core with orders to connect the system with an Engine and then uses one of the Correctness tools. The Application provides the graphical user interface (GUI) and manages input/output for creation, saving and loading of data structures, e.g., rules, structures and morphisms. To create structures usable by an Engine, the Application must know the specifics of the data structures of the Engine. This static dependency is illustrated in Figure 3 (a).

Our Current Status

ENFORCe is a work in progress. A Java based implementation of the Core component is running. We have a working `GraphEngine` based on software from GRAJ [Bus04] and implementations of the Transformations from constraints to right- to left application conditions. Our plans include an Application with a GUI allowing users to experiment with the functionality promised by ENFORCe.

5 Related Systems

There are several related systems that may be distinguished functionally and methodically: E.g., one may distinguish between (e) transformation engines and (s) tools supporting model checking, verification or analysis (termination, confluence).

Tool	Abbreviation/Synopsis		Reference
AGG	Attributed Graph Grammar system	s	[Tae04]
AUGUR 2	analysis of hypergraph transformation system	s	[KK06]
CHECKVML	CHECK Visual Modelling Languages	s	[SV03]
FUJABA	From UML to JAva and BAck	s	[BGN+04]
GROOVE	GRaph based Object-Oriented VErification	s	[KR06]
PROGRES	PROgramming with Graph REwriting System	s	[SWZ99]
GRAJ	GRAph programs in Java	e	[Bus04]
GRGEN	Graph Rewrite GENerator	e	[GBG+06]
YAM	York Abstract Machine	e	[MP06]

Table 3: A selection of related systems

AGG [Tae04] is a general development environment for attributed graph transformation systems written in Java. It consists of a SPO-based transformation engine, graphical editors, a visual interpreter/debugger and a set of validation tools. AGG supports graph parsing, a transformation of (basic) constraints into equivalent left application conditions [HW95] and critical pair analysis, i.e., a test for confluency.

AUGUR 2 [KK06] is a tool for analyzing node-preserving hyperedge transformation systems by abstraction to so-called Petri graphs: A node in a Petri graph represents multiple hypergraph nodes, while token represent hyperedges. The system consists of approximating algorithms for the abstraction of hypergraph transformation system, a coverability as well as a planned reachability algorithm for deciding Petri graph properties, and abstraction refinement algorithms in the case of a counterexample.

CHECKVML [SV03] is a tool for model checking dynamic consistency properties of arbitrary visual modeling languages (e.g., UML, Petri nets) by generating a model-level specification. Such high-level specifications are translated into a tool independent abstract representation of transition systems defined by a corresponding meta-model. This intermediate representation is automatically translated to the input language of the back-end model checker tool SPIN.

The FUJABA TOOL SUITE [BGN+04] is primarily an UML CASE Tool. Implemented as a plugin

within this framework is an approximative invariant checker [BBG+06] for conjunctions of negative existential graph conditions for transformation system with basic negative application condition and priorities (SPO with gluing condition). Apart from the priorities, the method corresponds to the construction of a weakest precondition and the decision of the implication problem while ignoring the application conditions and the implicit gluing conditions of the rules (both correct approximations).

GRAJ [Bus04] is a tool for executing graph programs. The system consists of a virtual machine, a compiler translating rules into GRAJ machine code and a recently developed graphical user interface. The virtual machine provides primitives for manipulating graphs and storing the execution history of a program needed for implementing the non-deterministic behavior of programs. The `GraphEngine` of ENFORCe will make use of GRAJ.

GRGEN [GBG+06] is a generative programming system for graph rewriting. It consists of a compiler for SPO rules specified in a declarative language, a transformation engine called libGr written in C and a shell-like frontend for the transformation engine called GrShell. GRGEN is aimed at attributed typed directed multigraphs, supporting various matching conditions and featuring attribute computation, relabeling and regular graph rewrite sequences comparable to graph programs.

GROOVE [KR06] is a set of (planned) tools for software model checking of object-oriented systems. It aims at directed edge-labeled graphs without parallel edges, a structure suitable for representing binary predicate logic. The GROOVE Simulator, consisting of a user interface and a SPO-based transformation engine, may be used for state space generation of (finite) transformation systems. The state space is translated to a Kripke structure for standard CTL model checking.

PROGRES [SWZ99] is a set of tools as well as a hybrid visual language for attributed graph transformation. The environment consists of graphical and textual editors supporting syntax-directed editing of graphical specifications and incremental parsing of textual language elements, an interpreter/debugger with built-in constraint checking facilities for transformation specifications, and a compiler backend that translates graph transformations into C-code and generates a tcl/tk-based user interface for calling graph transformations and displaying manipulated graphs.

YAM [MP06] defines a stack-based abstract machine language for graph transformation, comparable to postscript for graphics. This includes low-level instructions as get node, get node/edge label, get source/target, add/delete/relabel node edge, to which graph transformations rules get translated to. The YAM interpreter is written in C, while a compiler for translating graph rules and programs to YAM code is still under development.

ENFORCe focuses on correctness of high-level programs with application conditions. Its functionality will distinguish it from most tools presented here, e.g., from AGG which is primarily concerned with confluency. Tools concerned with correctness include AUGUR 2, CHECKVML, GROOVE and the FUJABA invariant checker. Due to its approximation technique, AUGUR 2 is restricted to node-preserving hypergraph replacement system, while it will be able to check a certain fragment of monadic second order properties for hypergraphs (see [BCKK04] for details). GROOVE is a model checker tool and will be able to handle arbitrary edge-labeled graph transformation systems with application conditions once abstraction is added to its features, while the type of checkable properties depends on the used abstraction. The FUJABA invariant checker is concerned with story patterns (= graph transformation rules with basic negative application conditions) and considers a small, decidable fragment of first-order logic. ENFORCe aims for full first-order properties.

6 Conclusion

ENFORCe is a suite of tools for ensuring the correctness of high-level programs. It is designed for weak adhesive HLR categories, exploiting the fact that necessary high-level algorithms can be based on a small set of structure-specific methods. Structurally, ENFORCe consists of Applications (e.g., user interface), Correctness tools (e.g., for correctness by construction), Engines (i.e., specific data structures and methods for a weak adhesive HLR category) and a Core containing general high-level notions and methods, connecting Engines with the rest of the system. This separation allows us to include new categories with a minimum of effort and to develop new Correctness tools and Transformation which instantly work with any Engine. While developing more efficient algorithms for a category, the ability to quickly exchange Engines could be useful for comparing the performance. Further topics could be the following:

(1) Engines for other weak adhesive HLR categories, like the categories of place-transition nets, hypergraphs, or typed attributed graphs.

(2) Adapters for other existing transformation engines like YAM or GRGEN. Adapters provide an interface and complete functionality, if necessary.

(3) Further Correctness tools and Transformations like semantic converters of conditions and rules, for switching the satisfiability and matching notions from arbitrary morphisms to $\mathcal{M}$-morphisms and vice versa [HP06], or a tool for proving the conflictfreeness of specifications.

(4) The construction of a correct program from a specification in form of a pre- and postcondition, e.g., see [LEHS06]).

Acknowledgment. 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).

References

[BBG+06] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proc. of the 28th int. conference on Software engineering (ICSE'06)*, pages 72–81. ACM Press, 2006.

[BCKK04] Paolo Baldan, Andrea Corradini, Barbara König, and Bernhard König. Verifying a behavioural logic for graph transformation systems. In *Proc. of COMETA '03*, volume 104 of *ENTCS*, pages 5–24. Elsevier, 2004.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, 2004.

[Bus04] Giorgio Busatto. GraJ: A system for executing graph programs in Java. Technical Report 3/04, University of Oldenburg, 2004. Available at [Uni].

[EEHP06] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundamenta Informaticae*, 2006. To appear. Available at [Uni].

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer-Verlag, Berlin, 2006.

[GBG+06] Rubino Geiß, Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 383–397. Springer, 2006.

[HP05] Annegret Habel and Karl-Heinz 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.

[HP06] Annegret Habel and Karl-Heinz Pennemann. Satisfiability of high-level conditions. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 430–444. Springer, 2006.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 445–460. Springer, 2006. A long version is available as technical report at [Uni].

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph grammars — a constructive approach. In *SEGRAGRA'95*, volume 2 of *ENTCS*, pages 95–104, 1995.

[KK06] Barbara König and Vitali Kozioura. Augur 2 — a new version of a tool for the analysis of graph transformation systems. In *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'06)*, ENTCS. Elsevier, 2006. To appear.

[KR06] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In *Model Checking Software (SPIN)*, volume 3925 of *LNCS*, pages 299–305. Springer, 2006.

[LEHS06] Marc Lohmann, Gregor Engels, Reiko Heckel, and Stefan Sauer. Model-driven monitoring: An application of graph transformation for design by contract. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*. Springer, 2006.

[MP06] Greg Manning and Detlef Plump. The York abstract machine. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT'06)*, ENTCS. Elsevier, 2006. To appear.

[SV03] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003: 6th International Conference on Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95. Springer, 2003.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In *Handbook of Graph Grammars and Computing by Graph Trans.*, volume 2, pages 487–550. World Scientific, 1999.

[Tae04] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446–453. Springer, 2004.

[Uni] `http://formale-sprachen.informatik.uni-oldenburg.de/pub/eindex.html`.