Project proposal, NWO-GBE Open Competition 2006

1.a. **Project Title**: Graphs for Abstract Interpretation of Languages
1.b. **Project Acronym**: GRAIL
1.c. **Principal Investigator**: Dr. A. Rensink, University of Twente

2 Summary

As more and more systems in our everyday environment contain major software parts, and we are depending on such systems more and more (*we are counting on them*), the importance of the dependability of the embedded software is increasing. Unfortunately, there are still very few generally applicable methods for *software verification*, i.e., the ensurance of its correct functioning under all circumstances. Reasons for this are, one the one hand, the degree of expertise necessary for existing verification methods, and on the other, their poor embedding in the average software development trajectory. An important practical objection is, moreover, that current verification methods typically assume the existence of a sufficiently detailed and precise *model of system behaviour*. In practice such models hardly ever exist, and the time and expertise to construct them is missing. Examples of methods that *are* being used widely in practice are therefore typing and testing, neither of which necessarily depends on the existence of models.

In this procect we investigate a new way of *automatically* verifying software on the basis of *code*, without assuming a predefined model. The technique used is *static analysis*, a general principle that encompasses typing; the new aspect is the use of *graph transformations* to capture the effect of the software. Graphs offer a natural model for the behaviour of dynamic software systems, and at the same time offer the basis for a generic form of static analysis, which can be driven by the properties to be verified.

3 Description of the Proposed Research

3.1 Problem Description

As computing is becoming ubiquitous, so is software. Until recently this was not perceived to be a problem by the manufacturers of the devices into which the computers are disappearing. Software could not pose any problem of greater complexity than the ones that had previously been faced, and solved, for the design and production of the devices themselves — or so the ubiquitous, and indeed quite understandable, assumption has been among the hardware engineers.

Reality is now known to be different. The combinatorial explosion due to the exponential growth of the *state space* size as a function of the *code* size leads to a complexity that the human mind cannot truly grasp, but that it continually fools itself into thinking that it *can* grasp. As a result, possible combinations of cases or states are very easily overlooked, leading to a lack of reliability and security, and hence a lack of dependability — the software typically cannot be justifiably trusted to function correctly under all circumstances. This is aggravated in no small degree by the phenomena of architectural erosion, legacy systems or plain bad design, which can lead to uncontrolled (in some cases exponential) growth in the size of the code base itself.

At the same time, in seeming contradiction to the lack of dependability of the systems embedded into our everyday environment, society is in fact depending on them more and more, increasing the potential consequences of failure.

Part of the solution to this problem lies in the application of automatic verification instead of (or in addition to) the current state-of-practice in software engineering — which more often than not amounts to testing only, with little if any notion of functionality coverage. What is more, due the architectural problems listed above, more often than not there is no good — i.e., precise — specification or model of the expected functionality in the first place. To be realistically applicable, a method for automatic verification should take this into account as well.

To meet these aims, in this project we propose to use *abstract interpretation* (also known as *static analysis*) of programs, where the basis of the analysis is in fact the code itself, and no model is required. That is not to say that we propose to do without models altogether; instead, in a very real sense, models are extracted from the code. This is in fact one aspect in which our approach deviates from existing methods for abstract interpretation, which are very syntax directed: essentially we represent program states as graphs, build a virtual machine for the programming language based on graph transformations, and using this systematically generate and analyse the state space as a state/transition model. However, rather than actually building the complete graphs, which are far too large and too many to go through, at this point the abstraction comes in: by cutting off irrelevant parts of the graph and collapsing other parts that are similar enough, the model can shrink sufficiently for analysis to be feasible — at the price of some precision, since the abstract model will only approximate the concrete one.

We will show some small example methods in C, that illustrate typical errors made by novice and experienced programmers alike, and which can be quite hard to detect.

```
int      *pi;
size_t   size;

void setValues1(int *newPtr, size_t nsz){
        pi= newPtr;
        size= nsz;
};

void setValues2(int *newPtr, size_t nsz){
        if(pi!=NULL)
                free(pi);
        pi= newPtr;
        size= nsz;
};

void setValues3(int *newPtr, size_t nsz){
        if(pi!=NULL)
                free(pi);
        pi= (int*)calloc(nsz, sizeof(int));
        size= nsz;
        mcpy(pi, newPtr, size);
};
```

Example 1 *Let us point out the sort of problems that can arise with the kind of code in the above listing. Figure 1 depicts sample states of the program as graphs, using a straightforward encoding.*

1. *If the memory pointed to by* `pi` *is not pointed to by any other pointer variable (*absence of sharing*), every call to* `setValues1` *(except for the first one) will cause this memory to lose*

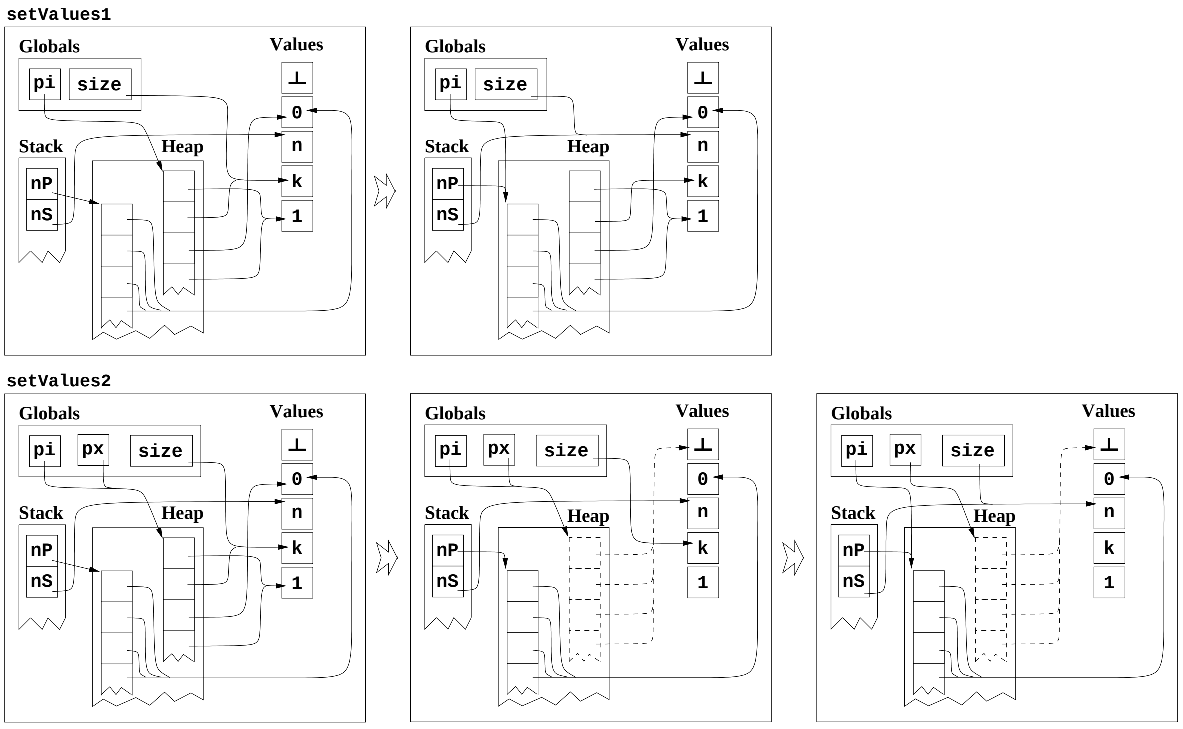

Figure 1: **Effects of executing** `setValues1` **and** `setValues2`

any chance to be freed — a phenomenon known as memory leakage*. This is detrimental for the system performance as a whole and ultimately a cause of instability and even failure.*

The two upper squares of figure 1 represent the state of the system before the first statement of `setValues1` *and after its last one. In a full example, there would be intermediate stages between these two graphs. Nodes in this hypothetical case represent either variables or values. The variables allocated in each of the three spaces (Global, Heap or Runtime Stack) that the memory of a system is divided into are enclosed in rectangles. Integer-valued variables are linked to their assigned value through arcs. Pointer variables are linked to the variable they point to also through arcs. The chosen initial state is precisely of the sort that can cause trouble (no sharing). As we can see in the graph to the right, the memory previously pointed to by* `pi` *becomes unreachable from any pointer variable in any of the three memory spaces. This can be detected in this representation by observing that the edge removed was the last one pointing to the node representing this piece of memory.*

2. *If on the other hand, we* are *sharing the memory pointed to by* `pi`*, a call to* `setValues2` *while "*`pi!=NULL`*" holds, will cause all pointers to this memory to become invalid. They are* dead references *or* dangling pointers*. It is a common mistake that such a situation goes unnoticed and dangling pointers end up being used somewhere else in the program. Usage of a dangling pointer causes problems ranging from data corruption to system failure.*

   In the lower part of Figure 1, we see the encoded state before the first statement, just after the conditional statement and after the last one. The middle stage represents the memory pointed to by `pi` *being freed (after the call to* `free`*): the variables point now to invalid values ($\perp$) (the dashed boxes are merely another way to show this same fact). In the last stage of the call,* `pi` *points now to the new piece of memory, but the variable(s) that shared its old value point to* *memory holding invalid values, allowing further (erroneous) uses of this memory to be caught.*

3. *In* `setValues3`*, the same problems may arise as in the previous case, but in the context of a function with different side effects. In the previous case, the called function afterwards holds a reference to the data passed to it, whereas here, such data is only read. In either case, the caller has to be aware of what happens with it: in the first case, it could have to respect some sharing policy on the resource; in the second, if this memory is allocated on the heap, it will have to be freed, since no reference to it has been kept.*

3.2 Approach

We propose to marry two areas of research that are both quite venerable in the context of computer science: Abstract Interpretation and Graph Transformation.

Abstract interpretation is a method for statically analysing the properties of programs — in other words, it is a method designed to work on the code level. It is usually reckoned to have started with the seminal paper by Cousot and Cousot [11], and has its roots in lattice and fixpoint theory. Analysing a program by abstract interpretation basically amounts to:

- Defining a lattice (or semi-lattice) that will serve as a universe of abstractions of the possible states a program can be in. These abstractions only partially represent the actual program states, but if done well, the part that they represent is sufficient to predict the properties of interest with a high decree of accuracy.
- Giving semantics to the programming language, in terms of operations on the elements of the lattice discussed above. That is, a statement transforms one abstract state into another. The meaning of a whole program is the composition of the individual abstract state transformers.
- Computing the final states of the program on this abstract level. The computation is defined as the fixpoint of the sequence of transformations.
- Determining whether all the (abstract) final states satisfy the properties of intereest. Since the abstraction typically loses some information, this step is not precise and may result in so-called *false negatives*, meaning that errors detected on the abstract level are to be taken as warnings rather than real, concrete errors.

It follows that abstract interpretation is parameterised by the choice of representation for the concrete and or abstract states. For example, a survey on the choices can be found in [12]. It is here where this project proposes a direction that is new with respect to pre-existing work. We propose to represent states of a system with graphs, and to interpret program statements through graph transformations [10, 9]. Since they have only little imposed structure, graphs are well suited to represent complex referential structures, as evidenced in fact by Figure 1, which essentially contains graphs already. This makes it easy for non-specialised users to learn the formalism. (This fact should not be dismissed as trivial, since — as argued above — the widespread acceptance of formal methods is held back by the learning curve involved in their application.)

So far, this merely fixes our concrete level of modelling; the more important choice is actually the abstract level. States and state spaces in all but the most trivial systems are huge; in fact the state space can easily be infinite. Our answer to this is graph *shapes*, which are essentially graphs quotiented through some similarity relation on the nodes. The similarity relation should be chosen so as to preserve and reflect graph properties according to the needs of the analysis being carried out. To illustrate this, let us continue with the above example.

Example 2 *Figure 2 shows the same sequence of example 1, only the states are now abstracted. Integer values are all collapsed into one cell (our fictitious analysis needs not pay any attention to them), and only one graph node is used to represent whole ranges of memory. Named variables are all represented as before. Would there be more, they could be culled (*sliced*, in model checking terms) away if they are irrelevant for the analysis.*

In terms of the shapes discussed above, the similarity relation is chosen such that all data values are "similar": only the undefined value is distinguished. Likewise, all memory cells that point to similar data values are similar, meaning that the abstraction essentially recognises two (kinds of) heap nodes only.

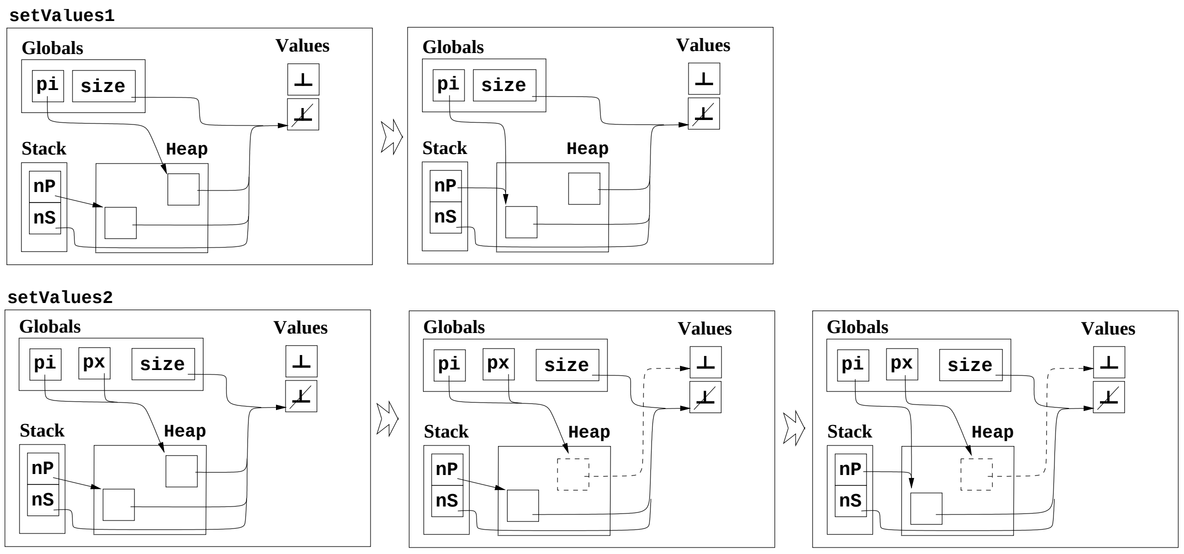

Figure 2: **Abstract execution of** `setValues1` **and** `setValues2`

To briefly address the feasibility of this method, let us point out that we already did some prior work; first of all, on explicit state spaces on the basis of graphs as states and graph transformation for operational semantics [13, 2, 20, 21], but also to develop a proof-of-concept implementation [19, 22, 16]. A further strand of work is the use of graph transformation for the definition of operational programming language semantics [3].

Summarizing, the advantages of this approach are:

- We have an intuitive and understandable concrete modelling formalism (graphs), which allows us to define operational semantics of the programming language in question without too much effort (see, e.g., [3] where we have done this for an experimental language with all essential object-oriented features);
- Due to the choice of graphs as concrete states, we have at our disposal a powerful and intuitive notion of abstraction that naturally uses structural similarities in the states and can be geared to property-driven abstractions;
- The (infinite-state) concrete and (finite-state) abstract model are extracted directly and automatically from the code and do not have to be constructed beforehand. These models can actually be used for other purposes besides verification, such as (for instance) test generation;

- The approach results in a completely automatic software verification method — where it should be stressed that, due to the abstraction and the implicit loss of information, errors found have the status of *warnings* only. (This is in fact true for all static analysis methods.)

In addition to working out the theory, we see it as imperative to *implement* the techniques developed, hand in hand with their theoretical developement. As a core engine we propose to use the GROOVE tool [16], which is designed to do concrete graph state generation.

3.3 Project Objectives

1. Define property-driven abstractions of graphs, based on the notion of node similarity and graph quotienting;
2. Extend existing algorithms for carring out the verification on the abstract model, to cope with the chosen shape-based representation;
3. Develop graph transformation-based operational semantics of the (often unsafe and poorly defined) concrete programming languages used in (embedded) systems programming;
4. Hand in hand with the activities above, implement the techniques and algorithms as an extension to GROOVE;
5. Show the viability of the approach using (at least) medium-sized case studies, ideally consisting of real production code fragments.

3.4 Other Related Work

Heap analysis. One of the larger bodies of work, and a source of inspiration for our research, is formed by the results on heap analysis by Sagiv and others [24]. This is a very powerful method, since it includes the ability to direct the abstraction using so-called *instrumentation predicates* — essentially encoding the properties that one is interested in. On the downside, they currently do not have an automatic method, since the updates of the instrumentation predicates have to be written by hand, shich is a far from trivial task.

Pointer analysis. Most of the work done in this area, most notably Evan's tool [15] (LCLint) is geared towards compile-time assertion of correctness in a restricted sense. A program is given a rather simplistic semantic of a partially ordered set of "variable updates", variables are given a "meta-state" (through program annotations) that has to be preserved at every returning point of the procedure. Cycles are *not* taken into account. This makes the results obtained with this tool inherently incomplete. The same kind of argument holds for several approaches in a similar direction.

Separation logic. Using ideas from Separation Logic [23], O'Hearn and others have recently achieved some successes in the static analysis of C programs, including liveness properties, which are traditionally not covered by this method: see [14]. Although comparable in spirit, their method is currently geared towards single-linked lists only.

Graph transformation-based abstractions. Within the theory of graph transformation, there are two alternative approaches worth mentioning. First of all, re-using the principles of McMillan-unfoldings [18] from Petri Nets, in [6, 7, 17] König et al. have proposed *Petri graph unfoldings*, which give rise to abstractions if one “cuts off” the unfolding early (before the analogous McMillan criterion is fulfilled). More recently, Bauer and Wilhelm in [8] have proposed a *multi-tiered* or *hierarchical* abstraction for graphs, sharing many of our basic ideas.

3.5 Embedding within own existing activities

This research is embedded in the research institute CTIT (Centre for Telematics and Information Technology) and NIRICT (Netherlands Institute for Research on ICT). In addition, the results of the GRAIL project will find its way in the CeDICT (3TU Centre for Dependable ICT Systems), where there is a need for an industrially applicable approach to software verification.

The GRAIL project perfectly complements the current NWO projects GROOVE (Graphs for Object-Oriented Verification, NWO 612.000.314), on model checking explicit state spaces on the basis of graphs, and GRASLAND (Graphs for Software Language Definition, NWO 612.063.408), on the use of graph transformation for the definition of operational programming language semantics.

4 Project planning

Phasing. The numbers in the following table refer to the activities defined in Section 3.3.

Tasks	1	2	3	4	5
Year 1	X		X	X	
Year 2		X	X	X	
Year 3		X		X	X
Year 4	— Consolidation and PhD thesis —				

Educational aspects. The candidate will enroll in the course program offered by IPA (see Section **??**). If needed, the candidate will take some courses from the Computer Science or Applied Mathematics curriculum offered by the faculty EWI. This will have to be decided on the basis of the candidate’s background.

5 Expected Use of Instrumentation

Not applicable.

6 Literature

Key publications of the research team

[1] BRINKSMA, E. Verification is experimentation! *Software Tools for Technology Transfer 3*, 2 (2001), 107–111.

[2] DISTEFANO, D., KATOEN, J.-P., AND RENSINK, A. Safety and liveness in concurrent pointer programs. In *Fourth International Symposium on Formal Methods for Components an d Objects*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To be published.

[3] KASTENBERG, H., KLEPPE, A., AND RENSINK, A. Defining object-oriented execution semantics using graph transformations. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)* (2006), R. Gorrieri and H. Wehrheim, Eds., vol. 4037 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 186–201.

[4] RENSINK, A. Canonical graph shapes. In *Programming Languages and Systems — European Symposium on Programming (ESOP)* (2004), D. A. Schmidt, Ed., vol. 2986 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 401–415.

[5] RUYS, T. C., AND BRINKSMA, E. Managing the verification trajectory. *Software Tools for Technology Transfer 4*, 2 (2003), 246–259.

Other references

[6] BALDAN, P., CORRADINI, A., AND KÖNIG, B. A static analysis technique for graph transformation systems. In *Concurrency Theory (CONCUR)* (2001), K. Larsen and M. Nielsen, Eds., vol. 2154 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 381–395.

[7] BALDAN, P., KÖNIG, B., AND KÖNIG, B. A logic for analyzing abstractions of graph transformation systems. In *Static Analysis* (2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 255–272.

[8] BAUER, J., AND WILHELM, R. Abstract interpretation of graph transformation. In *Simulation and Verification of Dynamic Systems* (2006), D. M. Nicol, C. Priami, H. R. Nielson, and A. M. Uhrmacher, Eds., no. 06161 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[9] CORRADINI, A., MONTANARI, U., ROSSI, F., EHRIG, H., HECKEL, R., AND LÖWE, M. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. I: Foundations. World Scientific, Singapore, 1997, ch. 3, pp. 163–246. Technical report version: Dipartimento di Informatica, TR–96–17.

[10] COURCELLE, B. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*. 1990, pp. 193–242.

[11] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1977), 238–252.

[12] DEUTSCH, A. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *PEPM '95: Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 1995), ACM Press, pp. 226–229.

[13] DISTEFANO, D., KATOEN, J.-P., AND RENSINK, A. Who is pointing when to whom? on the automated verification of linked list structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)* (2004), K. Lodaya and M. Mahajan, Eds., vol. 3328 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 250–262.

[14] DISTEFANO, D., O'HEARN, P., AND YANG, H. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2006), H. Hermanns and J. Palsberg, Eds., vol. 3920 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 287–302.

[15] EVANS, D. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (New York, NY, USA, 1996), ACM Press, pp. 44–53.

[16] KASTENBERG, H., AND RENSINK, A. Model checking dynamic states in GROOVE. In *Model Checking Software (SPIN)* (2006), A. Valmari, Ed., vol. 3925 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 299–305.

[17] KÖNIG, B., AND KOZIOURA, V. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2006), H. Hermanns and J. Palsberg, Eds., vol. 3920 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 197–211.

[18] MCMILLAN, K. L. A technique of state space search based on unfolding. *Form. Methods Syst. Des. 6*, 1 (1995), 45–65.

[19] RENSINK, A. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)* (2004), J. Pfalz, M. Nagl, and B. Böhlen, Eds., vol. 3062 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 479–485.

[20] RENSINK, A. Representing first-order logic using graphs. In *International Conference on Graph Transformations (ICGT)* (2004), H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., vol. 3256 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 319–335.

[21] RENSINK, A. Nested quantification in graph transformation rules. In *Graph Transformations (ICGT)* (2006), A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds., vol. 4178 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 1–13.

[22] RENSINK, A., SCHMIDT, Á., AND VARRÓ, D. Model checking graph transformations: A comparison of two approaches. In *International Conference on Graph Transformations (ICGT)* (2004), H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., vol. 3256 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 226–241.

[23] REYNOLDS, J. Separation logic: a logic for shared mutable data structures. *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on* (2002), 55–74.

[24] SAGIV, S., REPS, T. W., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst. 24*, 3 (2002), 217–298.