CTIT Technical Report TR–CTIT–03–35, University of Twente, 2003

A Logic of Local Graph Shapes

Arend Rensink
Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl

August 11, 2003

Abstract

Graphs are an intuitive model for states of a (software) system that involve dynamic resource allocation or pointer structures, such as, for instance, object-oriented programs. However, a straightforward encoding results in large individual states and large, or even unbounded, state spaces. As usual, some form of abstraction is necessary in order to arrive at a tractable model.

In this paper we propose a fragment of first-order graph logic that we call *local shape logic* (LSL) as a possible abstraction mechanism. LSL is inspired by previous work of Sagiv, Reps and Wilhelm. An LSL formula constrains the multiplicities of nodes and edges in state graphs; abstraction is achieved by reasoning not about individual, concrete state graphs but about their characteristic shape properties. We show that the expressiveness of LSL is equal to integer programming, and as a consequence, LSL is decidable. (This is a slight generalisation of the decidability of two-variable first order logic.)

We go on to define the concept of the *canonical shape* of a state graph, which is expressed in a *monomorphic* sub-fragment of LSL, for which we define a graphical representation. We show that the canonical shapes give rise to a finite abstraction of the state space of a software system, and we give an upper bound to the size of this abstract state space.

Contents

1 Introduction **2**

2 Graphs and type graphs **3**
2.1 Graphs 3
2.2 Graph morphisms, graph partitionings, and type graphs 4
2.3 Integer programs 6

3 Shape logic **7**
3.1 Multiplicities 9
3.2 Local shape logic 10
3.3 From integer programs to local shape logic 17

4 Shape graphs **19**
4.1 Monomorphic shapes 21
4.2 Canonical shapes 24

5 Conclusion **27**
5.1 Summary and related work . . . 27
5.2 Variations and extensions . . . 29
5.3 Future work . . . 30

References **30**

1 Introduction

This paper contributes to an investigation into the use of graphs as models of system states, for the (eventual) purpose of system verification, especially of *software* systems. The approach we follow in this investigation is based on the following choices:

- An individual system state (state snapshot) is modelled as an *edge-labelled graph*, in which the nodes roughly stand for the entities or resources (records, objects) present in the state, and the edges for properties or fields (attributes, variables) of those resources.
- The dynamic behaviour of a system is modelled as a *transition system* in which the states are graphs in the above sense, and the transitions are (possibly labelled) connections from each state to all potential next states reachable from that state, containing enough information to trace the identities of resources in the states.
- Such transition systems are generated from *graph grammars*, consisting of an initial graph that models the initial state of the system, and a set of production rules that model the computation steps of the system. Each production rule serves as an instruction on how to get from one state to a next.

The advantage of using graphs and graph grammars as a basis for the semantics is that they naturally arise on different levels of modelling, from the design level (UML models, see e.g. [6, 16, 15, 20]) to the code and even byte code level ([1, 21]).

As usual in the context of verification, the main problem is *state space explosion*; i.e., the effect that, even for small systems, the number of states to be analysed exceeds all reasonable bounds. The most promising solution technique to cope with this is *abstraction*, meaning that information is discarded from the model after which it can be represented more compactly — usually at the cost of either soundness or completeness of the verification.

In the context of graphs, we have previously studied abstraction techniques in [11, 12]. The idea there is to have one or more nodes whose cardinality is not *a priori* fixed but may grow unboundedly in the course of system execution. This idea can be found also in *shape graphs*, as defined by by Sagiv, Reps and Wilhelm in [28]. Here, too, some graph nodes (called *summary nodes*) stand for multiple instances; furthermore, shape graphs contain additional information about whether an edge is necessarily there for every instance of a given node and whether outgoing edges may be pointing to (i.e., sharing) the same node instance.

This paper is a consequence of our earlier efforts, inspired by the work on shape graphs. We define a theory of graph shape by putting additional information about the edges, as well as information about the multiplicity of nodes, in the form of a formula in what we call *local shape logic* (LSL). LSL is essentially a fragment of typed first-order logic, where the typing is controlled by a *type graph* of which all models are required to be instances. Besides the type graph, another parameter in the definition of LSL is a *multiplicity algebra*, which partially controls the expressiveness of the

logic. We show LSL to be decidable by reducing formulae to sets of integer programs. Since, vice versa, any integer program can be expressed as an LSL formula it follows that the two are equally expressive.

The combination of a type graph and a shape constraint gives rise to a (generalised) shape graph. Thus, each shape graph defines a set of state graphs, viz. those instances of the type graph that satisfy the shape constraint. Unfortunately, LSL formulae in general lack the appealing pictorial representation of graphs. However, as a last result we also show a graphical representation of a *monomorphic* fragment of LSL, and we show that any shape graph is equivalent to a set of monomorphic shape graphs. We show that the set of distinct monomorphic shape graphs over a given type graph is finite, and we give an upper bound for its size.

The paper is structured as follows: Section 2 contains basic definitions, in Section 3 we introduce local shape logic and show its decidability; and in Section 4 we discuss (monomorphic) shape graphs. Section 5 concludes and discusses related and future work.

2 Graphs and type graphs

We represent states as graphs. Nodes can be thought of as *locations* or *objects*, and edges are used to represent *variables*, in particular *references*. Computation steps, which are as usual equated with the state changes they inflict, are thus modelled by changes to graphs.

2.1 Graphs

We start by recalling the formal definition of graphs. We assume the existence of the following:

- A global set $\mathbf{N}$ of *nodes*, ranged over by u, v, w; subsets of $\mathbf{N}$ are denoted N, V, W. We use the symbol $\bot$ ($\notin \mathbf{N}$) to denote an *undefined* node; for an arbitrary set $N \subseteq \mathbf{N}$ we write $N_\bot$ to denote $N \cup \{\bot\}$.
- a global set $\mathbf{L}$ of *labels*, ranged over by a, b, c.

In this paper we use edge-labelled graphs, defined as follows:

Definition 2.1 (graph) *A* graph G *is a tuple* (N, E)*, where*

- $N \subseteq \mathbf{N}$ *is a finite set of* nodes*;*
- $E \subseteq N \times \mathbf{L} \times N_\bot$ *is a finite set of* edges*.*

It follows that a graph consists of binary edges (v, a, w) with $w \in \mathbf{N}$ but also *unary* edges $(v, a, \bot)$. We also refer to unary edges as *node properties* or *node predicates*. For most purposes unary and binary edges will be treated in the same way.

We use $\mathbf{G}$ to denote the set of all graphs, ranged over by G, H. Given a graph G, we denote the node and edge sets of G by N_G, E_G, respectively; moreover, $src_G\colon E_G \to N_G$, $tgt_G\colon E_G \to N_{G,\bot}$ and $\ell_G\colon E_G \to \mathbf{L}$ are functions extracting the source nodes, target nodes and labels from the edges of G. Note that, due to the finiteness of N and E, the set $L_G = \{\ell(e) \mid e \in E_G\}$ of labels occurring in G is finite as well. We feel free to drop the subscripts $_G$ if they are clear from the context.

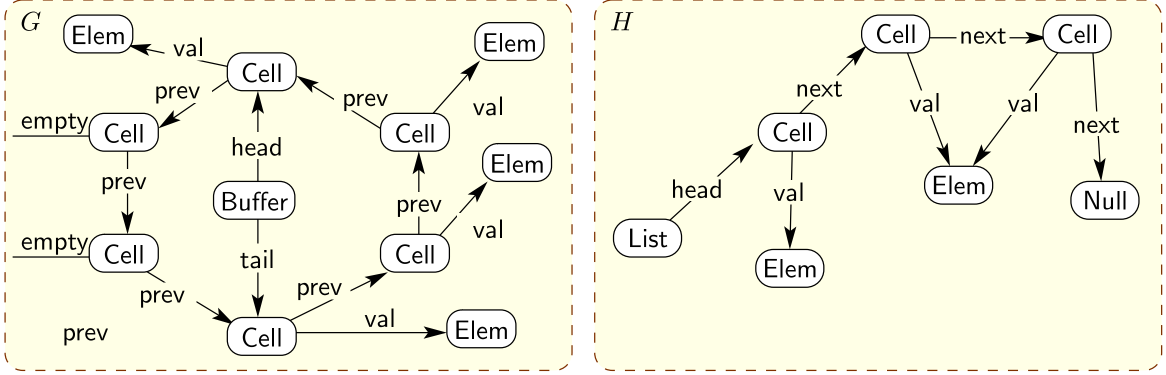


Figure 1: State snapshots of a six-slot circular buffer and a three-element linked list

Pictorial representation. As usual, we draw graphs by showing nodes as boxes and binary edges as arrows between them. Unary edges are drawn without an arrow head or target node; alternatively, we often represent unary edges by writing their labels inside the source nodes and omitting the edges. Also, we combine edges with the same source and target nodes by drawing a single edge decorated with the collection of labels. The following example gives an idea how such graphs may be used to represent states.

Example 2.2 *Figure 1 shows two state graphs. The graph G on the left hand side models a circular buffer, as a backwards-linked list of cells of which a number is filled with values (modelled by* val*-labelled edges to the nodes representing the respective values) and the others are empty (modelled by* empty*-labelled unary edges). One of the cells is designated* head *to indicate that this is the head of the buffer, whose value is to be retrieved next; in contrast, the* tail *cell contains the newest value.* Buffer*,* Cell *and* Elem *are node predicates used to reflect the types of the nodes. The right hand side graph H depicts a linked list, using a similar encoding.*

The effects of execution steps are modelled by modifications to the state graphs, resulting in new graphs which differ (locally) from the previous ones. This gives rise to a transition system in which the states are graphs and the transitions graph transformations. For instance, the primary operations upon a circular buffer are the insertion and retrieval of values: these result in changes in the neighbourhood of the tail- and head-edges. Figure 2 shows an example transition system, in which the states are three-slot circular buffers built up in the same way as G in Figure 1, and the transitions are generated by the insertion ($\langle$put$\rangle$) and retrieval ($\langle$get$\rangle$) of values.

2.2 Graph morphisms, graph partitionings, and type graphs

As can be seen from the examples above, graphs essentially represent structural relations between nodes and properties of nodes, in the form of labelled edges. The identities chosen for the nodes, i.e., the particular choice of elements of $\mathbf{N}$, is not part of this structure: they only serve to distinguish the nodes from one another. For that reason, if we draw a graph we don't usually include the node identities at all: the nodes are then already distinguished by their position in the picture.

The fact that the choice of node identities is irrelevant is brought out most clearly by the concept of graph *morphism*: these are mappings between graphs that preserve the structurally important information but may change the underlying node identities.

Definition 2.3 (graph morphism) *Given two graphs G, H, a* graph morphism *f from G to H is a function $f\colon N_G \to N_H$, strictly extended to $\bot$, such that $(f(src_G(e)), \ell(e), f(tgt_G(e))) \in E_H$ for all $e \in E_G$.*

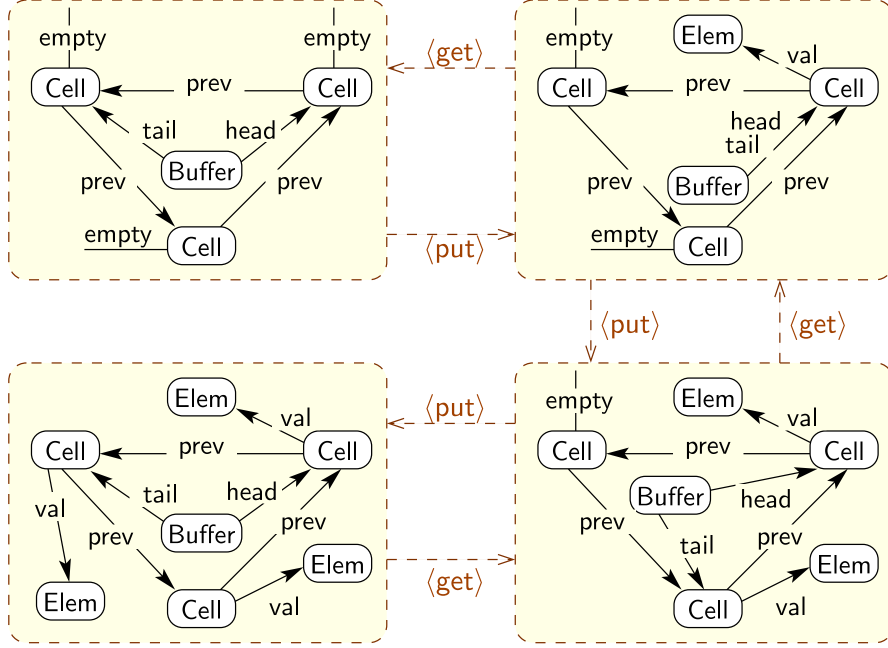


Figure 2: Graph transition system for a three-slot circular buffer; see G in Figure 1

$f\colon G \to H$ denotes that f is a morphism from G to H. A morphism f is called *injective* [*surjective, bijective*] if f is an injective [surjective, bijective] function both on nodes and edges. If f is bijective we also call it an *isomorphism*; furthermore, we write $G \cong H$ to indicate that there exists an isomorphism $f\colon G \to H$. Isomorphic graphs are thus structurally identical and only differ in the accidental choice of node identities.

In the following we also need the following (standard) notions of graph *partitioning*: for a given graph $G = (N, E)$, every node equivalence relation $\sim\, \subseteq N \times N$ gives rise to a partitioned graph $G/{\sim} = (N/{\sim}, E/{\sim})$ where

$$\begin{aligned} N/{\sim} &= \{[v]_\sim \mid v \in N\} \\ E/{\sim} &= \{[v]_\sim, a, [w]_\sim \mid (v, a, w) \in E\} \end{aligned}$$

(in which, as usual, $[v]_\sim = \{w \in N \mid v \sim w\}$ for all $v \in N$, and moreover $[\bot]_\sim = \bot$). Furthermore, we use $\pi_\sim\colon G \to G/{\sim}$ to denote the (surjective) morphism defined by $\pi_\sim(v) = [v]_\sim$ for all $v \in N$. Also, for an arbitrary partitioning Π of the set of nodes N and an arbitrary node $v \in N$, we use $[v]_\Pi$ to denote the unique $V \in \Pi$ such that $v \in V$; furthermore, $[\bot]_\Pi = \bot$.

The graphs used to model the states of a given software system are, of course, not arbitrary. For one thing, not all edges or combinations of edges are allowed. To take the circular buffer G of Figures 1 and 2, there are only eight labels occurring in any of the state graphs; Buffer, Cell and Elem only occur as node predicates of mutually exclusive nodes; prev-edges only occur between Cell-nodes; et cetera. Such information can be captured partially using *graph typings*.

Definition 2.4 (typing) *Let $G \in \mathbf{G}$ be arbitrary. A* typing *of G is a morphism $\tau\colon G \to T$, where $T \in \mathbf{G}$ is called a* type graph. *We call T a* type *of G and G an* instance *of T.*

We use $\mathbf{G}^T$ to denote the set of instances of a graph T. For instance, Figure 3 shows types for the state graphs in Figure 1.

The notion of graph typing, however, is rather weak: the existence of a morphism from a would-be instance graph to a would-be type graph can only forbid but never enforce the presence

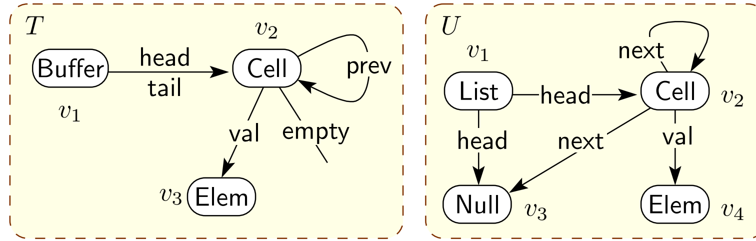


Figure 3: Type graphs T for circular buffers and U for linked lists

of certain edges in the instance. To take the type graph U for lists in Figure 3, the *intention* is that any instance obeys the following structural properties:

- Every List-labelled node has precisely one outgoing head-edge, to the first element of the corresponding list or to a Null-node;
- There is precisely one Null-node;
- Every Cell-node has precisely one outgoing val-edge to an Elem-node, and one outgoing next-edge, either to another Cell-node or to a Null-node;
- Every Cell-node has either one incoming next-edge or one incoming head-edge (so Cell-nodes are not shared).

Although H in Figure 1 indeed satisfies these intended properties, U has many instances that do not. For instance, the graph in Figure 4 does not conform to the intended list structure at all, but is nevertheless an instance of U. Clearly, the intended structure of state graphs cannot be fully enforced by graph types.

In the next sections we discuss a way to strengthen the notion of graph typing.

2.3 Integer programs

Since some of the results of this paper depend on integer programming, we recall the basic concepts here. An integer program is a set of linear equations for which we are only interested in solutions consisting of natural numbers. Often this is combined with an *optimisation* criterion: the problem is then not to find *any* solution but the one that minimises a given linear expression. In this paper the optimisation plays no role, or (alternatively) the optimisation function is a constant.

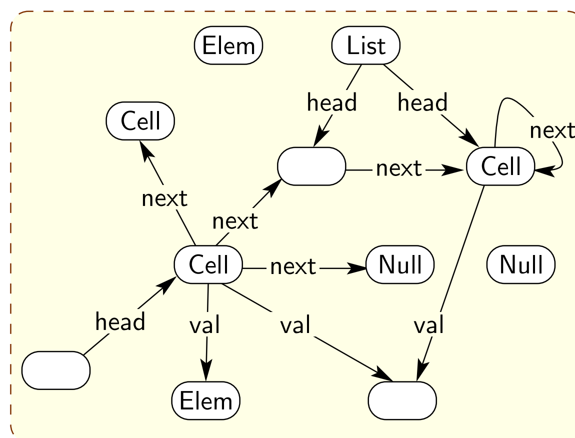


Figure 4: An instance of U in Figure 3 that does not have the intended list structure

A common form to state an integer program is as a pair A, b where A is an $m \times n$ integer matrix and b an m-dimensional integer vector. A solution is an n-dimensional natural vector x such that $Ax = b$. We use $A_{i,j}$ to denote the coefficients of A, b_i to denote the elements of b and x_j to denote the elements of x; thus $Ax = b$ is equivalent to a set of equalities

$$\textstyle\sum_{1 \leq j \leq n} A_{i,j} \cdot x_j = b_i \quad \text{for } 1 \leq i \leq m \enspace .$$

Note that that inequalities may be modelled as well, at the cost of a fresh variable to absorb the difference between the two sides of the inequality. We recall the following complexity result:

Theorem 2.5 (cf. [24]) *Checking whether an integer program admits a solution is NP-complete.*

Notation. We use $A_{_,j}$ to denote the j'th row of A, and for a vector b we use $\max |b|$ to denote the maximum absolute value occurring in b.

3 Shape logic

To strengthen the control offered by type graphs we need to formulate additional constraints on instances. The natural way to do this is through predicate logic. We first define a general *shape logic*, SL^T, which is a first-order graph logic, defined relative to a type graph T. Later on (Section 3.2) we restrict this to a fragment of which we show decidability.

For the purpose of the shape logics purpose we assume a countable set of variables $\mathbf{V}$. Formulae of SL^T are generated by the following grammar:

$$\phi \;\; ::= \;\; \mid x = x \mid x \stackrel{a}{-} \mid x \xrightarrow{a} x \mid \neg\phi \mid \phi \vee \phi \mid \forall x{:}\, v.\, \phi \enspace .$$

In this grammar, $a \in \mathbf{L}$ denotes an arbitrary label, $x \in \mathbf{V}$ an arbitrary (node) variable and $v \in N_T$ a node of the type graph T. ϕ denotes a *shape constraint* for G. We employ the usual abbreviations $\phi \Rightarrow \psi$, $\phi \wedge \psi$, $\exists x{:}\, v.\, \phi$ and $\exists! x{:}\, v.\, \phi$. Moreover, we also introduce so-called *counting quantifiers* as syntactic sugar: for any $n \in \mathbf{N}$, $\exists^n x{:}\, v.\, \phi$ expresses that there are *at least* n instances of v that satisfy ϕ (when substituted for x). Formally:

$$\exists^n x{:}\, v.\, \phi \quad \equiv \quad \exists x_1, \ldots, x_n{:}\, v.\, \bigwedge_{1 \leq i \leq n} \phi\{x_i/x\} \wedge \bigwedge_{1 \leq j \leq n, i \neq j} x_i \neq x_j$$

where $\phi\{y/x\}$ is the formula obtained by substituting every free occurrence of x by y. Note that $\exists^0 x{:}\, v.\, \phi$ is equivalent to $\mathbf{tt}$, $\exists^1 x{:}\, v.\, \phi$ is equivalent to $\exists x{:}\, v.\, \phi$ and $\exists! x{:}\, v.\, \phi$ is equivalent to $\exists x{:}\, v.\, \phi \wedge \nexists^2 x{:}\, v.\, \phi$.

The predicates $x \stackrel{a}{-}$ and $x \xrightarrow{a} y$ express that there exists a (unary resp. binary) a-labelled edge in a would-be instance of T, from the node denoted by x and leading (in the case of a binary edge) to the node denoted by y. It is sometimes convenient to write $x \xrightarrow{a} \bot$ rather than $x \stackrel{a}{-}$; i.e., we use $\bot$ ($\notin \mathbf{V}$) as a pseudo-variable.

We consider only formulas that are *well-typed* according to T, in the sense that each free variable x in a formula has an associated type node $v_x \in N_T$ such that $(v_x, a, \bot) \in E_T$ for all propositions $x \stackrel{a}{-}$ and $(v_x, a, v_y) \in E_T$ for all propositions $x \xrightarrow{a} y$. Note that quantification indeed imposes such an association between variables in the formula and type nodes. We do not work this out formally here.

Example 3.1 (list shape constraints) *Using shape constraints, we can give a much tighter characterisation of instances than enforced by the type graph alone, more closely capturing the intended structure. For instance, we can add the following constraints to U in Figure 3:*

$$\forall x{:}\, v_1.\, x \xrightarrow{\mathsf{List}} \wedge ((\exists! y{:}\, v_2.\, x \xrightarrow{\mathsf{head}} y \wedge \nexists y{:}\, v_3.\, x \xrightarrow{\mathsf{head}} y) \vee (\exists! y{:}\, v_3.\, x \xrightarrow{\mathsf{head}} y \wedge \nexists y{:}\, v_2.\, x \xrightarrow{\mathsf{head}} y)) \quad (1)$$
$$\forall x{:}\, v_2.\, x \xrightarrow{\mathsf{Cell}} \wedge (\exists! y{:}\, v_4.\, x \xrightarrow{\mathsf{val}} y) \quad (2)$$
$$\forall x{:}\, v_2.\, (\exists! y{:}\, v_2.\, x \xrightarrow{\mathsf{next}} y \wedge \nexists y{:}\, v_3.\, x \xrightarrow{\mathsf{next}} y) \vee (\exists! y{:}\, v_3.\, x \xrightarrow{\mathsf{next}} y \wedge \nexists y{:}\, v_2.\, x \xrightarrow{\mathsf{next}} y) \quad (3)$$
$$\forall x{:}\, v_2.\, (\exists y{:}\, v_1.\, y \xrightarrow{\mathsf{head}} x \wedge \nexists y{:}\, v_2.\, y \xrightarrow{\mathsf{next}} x) \vee (\nexists y{:}\, v_1.\, y \xrightarrow{\mathsf{head}} x \wedge \exists! y{:}\, v_2.\, y \xrightarrow{\mathsf{next}} x) \quad (4)$$
$$\exists! y{:}\, v_3.\, \mathbf{tt} \wedge \forall x{:}\, v_3.\, x \xrightarrow{\mathsf{Null}} \quad (5)$$
$$\forall x{:}\, v_4.\, x \xrightarrow{\mathsf{Elem}} \quad (6)$$
$$\exists x{:}\, v_4.\, \nexists y{:}\, v_2.\, y \xrightarrow{\mathsf{val}} x \quad (7)$$

Constraint (1) *expresses that every v_1-node has the* List*-property and a unique outgoing* head*-edge. Constraint* (2) *states that every v_2-node should have the* Cell*-property and a unique outgoing* val*-edge. Constraint* (3) *states that every* Cell*-node should have a unique outgoing* next*-edge, either to a* Cell*-node or to a* Null*-node. Constraint* (4) *expresses a* no-sharing *property of* Cell*-nodes: each* Cell*-node has either an incoming* head*-edge or a unique incoming* next*-edge. Constraint* (5) *states that there should be a unique v_3-node, with the* Null*-property. Constraint* (6) *states that every v_4-node should have the* Elem*-property. Finally,* (7) *expresses that there exists an* Elem*-node* not *reachable from a* Cell*-node.*

The unique U-typing of H in Figure 1 clearly satisfies all these constraints except (7)*, whereas the typing of the graph in Figure 4* only *satisfies* (7)*.*

Note that we can *not* formulate a constraint in SL to express that every v_1-instance should be connected (through a head-edge followed by a sequence of next-edges) to the unique v_3-instance. This is a consequence of the fact that first-order graph logic cannot express connectedness (see, e.g., Courcelle [9]).

Example 3.2 (ordered elements) *As another example, suppose that there is a natural ordering over* Elem*-nodes, expressed through additional* leq*-labelled edges; that is, $(v, \mathsf{leq}, w) \in E$ for an instance graph implies $v < w$ in the assumed natural ordering. This would be represented by an additional edge (v_4, leq, v_4) in U of Figure 3. Furthermore suppose that we require the elements in the list to be in ascending order. The following two shape constraints express the transitivity of the* Elem*-ordering and the ascending order of the list:*

$$\forall x{:}\, v_4.\, \forall y{:}\, v_4.\, \forall z{:}\, v_4.\, x \xrightarrow{\mathsf{leq}} y \wedge y \xrightarrow{\mathsf{leq}} z \Rightarrow x \xrightarrow{\mathsf{leq}} z \quad (8)$$
$$\forall c_1{:}\, v_2.\, \forall c_2{:}\, v_2.\, \forall x_1{:}\, v_4.\, \forall x_1{:}\, v_4.\, c_1 \xrightarrow{\mathsf{next}} c_2 \wedge c_1 \xrightarrow{\mathsf{val}} x_1 \wedge c_2 \xrightarrow{\mathsf{val}} x_2 \Rightarrow x_1 \xrightarrow{\mathsf{leq}} x_2 \quad (9)$$

The semantics of shape constraints is defined by the typings that satisfy them. Formally, satisfaction of a formula $\phi \in \mathsf{SL}^T$ is defined by a ternary predicate $\tau, \theta \vDash \phi$, where $\tau{:}\, G \to T$ is a typing and $\theta{:}\, fv(\phi) \to N_G$ a valuation of the free variables of ϕ (extended strictly to $\bot$) such that $\tau(\phi(x)) = v_x$ for all $x \in fv(\phi)$. For a given θ, $x \in \mathbf{V}$ and $v \in \mathbf{N}$, $\theta\{v/x\}$ denotes the valuation that equals θ on all $y \neq x$ and maps x to v.

$$\begin{array}{ll} \tau, \theta \vDash x = y & \text{if } \theta(x) = \theta(y) \\ \tau, \theta \vDash x \xrightarrow{a} y & \text{if } (\theta(x), a, \theta(y)) \in E_G \\ \tau, \theta \vDash \neg\phi & \text{if } \tau, \theta \nvDash \phi \\ \tau, \theta \vDash \phi \vee \psi & \text{if } \tau, \theta \vDash \phi \text{ or } \tau, \theta \vDash \psi \\ \tau, \theta \vDash \forall x{:}\, v.\, \phi & \text{if } \tau, \theta\{w/x\} \vDash \phi \text{ for all } w \in \tau^{-1}(v). \end{array}$$

Shape logic clearly strengthens the notion of typing. Unfortunately, it is too powerful for our purpose. As explained in the introduction, we intend to use shape constraints to characterise state abstractions; this makes it desirable, if not necessary, for the logic to be decidable. Being a (non-monadic) first order logic, SL does not meet that criterion.

Corollary 3.3 *Satisfiability of* SL*-formulae is not decidable.*

Instead, in this paper we concentrate on a reduced fragment which only allows to express *local* shape properties: *multiplicities* of single node instances, and of their incoming and outgoing edges.

3.1 Multiplicities

A multiplicity is an abstract indication of set cardinality; for instance, in the context of graphs, the set of instances of a given (type) node, or the set of edges from a given instance node. A well-known example are the UML multiplicities (see Example 3.5 below).

Definition 3.4 (multiplicity algebra) *A* multiplicity algebra *is a tuple* $\langle \mathbf{M}, _ : _, \bigsqcap, \mathbf{0}, \mathbf{1} \rangle$ *where*

- $\mathbf{M}$ *is a set of* multiplicities*;*
- $_ : _ \subseteq \mathbf{N} \times \mathbf{M}$ *is a* membership *predicate. For all* $\mu \in \mathbf{M}$ *we denote* $\mathbf{N}^\mu = \{m \in \mathbf{N} \mid m : \mu\}$*;*
- $\bigsqcap : \mathbf{2^M} \to \mathbf{M}$ *is* intersection*, with* $m : \bigsqcap M$ *iff* $\forall \mu \in M.\, m : \mu$ *(hence* $\mathbf{N}^{\bigsqcap M} = \bigcap_{\mu \in M} \mathbf{N}^\mu$*);*
- $\mathbf{0} \in \mathbf{M}$ *is the* zero multiplicity*, such that* $\mathbf{N^0} = \{0\}$*;*
- $\mathbf{1} \in \mathbf{M}$ *is the* singular multiplicity*, such that* $\mathbf{N^1} = \{1\}$.

It follows that any multiplicity algebra $\mathbf{M}$ also contains the *inconsistent multiplicity* $\bot = \bigsqcap \mathbf{M}$ (thus $\mathbf{N}^\bot = \emptyset$) and a *largest multiplicity* $\top = \bigsqcap \emptyset$ (thus $\mathbf{N}^\top = \mathbf{N}$). Some auxiliary concepts:

- We call $\mathbf{M}$ *interval based* if for all $\mu \in \mathbf{M}$, the lower bound $\lfloor \mu \rfloor = \min \{m \in \mathbf{N} \mid m : \mu\}$ and upper bound $\lceil \mu \rceil = \max \{m \in \mathbf{N} \mid m : \mu\}$ (where $\min \emptyset = \max \mathbf{N} = \omega$ and $\max \emptyset = 0$) are such that $\mathbf{N}^\mu = \{i \in \mathbf{N} \mid \lfloor \mu \rfloor \leq i \leq \lceil \mu \rceil\}$.
- We say that $\mathbf{M}$ has *collective complements* if for all $\mu \in \mathbf{M}$, there is a set $\bar{\mu} \subseteq \mathbf{M}$ such that $\mu \sqcap \nu = \bot$ for all $\nu \in \bar{\mu}$, and $\mathbf{N}^\mu \cup \bigcup_{\nu \in \bar{\mu}} \mathbf{N}^\nu = \mathbf{N}$.
- We define *multiplicity addition* $\oplus : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$ as follows:

$$\mu_1 \oplus \mu_2 = \bigsqcap \{\mu \in \mathbf{M} \mid m_1 : \mu_1 \wedge m_2 : \mu_2 \Rightarrow m_1 + m_2 : \mu\} \ .$$

  It follows that $\mathbf{N}^{\mu_1 \oplus \mu_2} \supseteq \mathbf{N}^{\mu_1} + \mathbf{N}^{\mu_2}$ (where addition is extended pointwise to sets).

- We define *set multiplicity* $\#_\mu V \in \mathbf{M}$ for an arbitrary finite set V, as follows:

$$\#_\mu V = \bigsqcap \{\mu \in \mathbf{M} \mid \# V : \mu\} \ .$$

- We define an *ordering* $\sqsubseteq \, \subseteq \mathbf{M} \times \mathbf{M}$ over multiplicities, derived from $\sqcap$ as usual: $\mu_1 \sqsubseteq \mu_2$ iff $\mu_1 \sqcap \mu_2 = \mu_1$. It follows that $\mu_1 \sqsubseteq \mu_2$ if and only if $\mathbf{N}^{\mu_1} \subseteq \mathbf{N}^{\mu_2}$.

In the sequel we will only consider interval-based multiplicity algebras. In particular we will use, apart from the obligatory elements enforced by the definition, the multiplicities $\uparrow$ for *at least one* (hence $m : \uparrow$ if $m > 0$) and $\Uparrow$ for *more than one* (hence $m : \Uparrow$ if $m > 1$). Note that $\{\bot, \mathbf{0}, \mathbf{1}, \uparrow, \Uparrow, \top\}$ has collective complements; in particular, $\bar{\mathbf{0}} = \uparrow$ and $\bar{\mathbf{1}} = \{\mathbf{0}, \Uparrow\}$. The corresponding membership predicate, addition operator and intersection lattice are summarised in Table 5.

$m :$ _	condition
$\top$	**tt**
$\Uparrow$	$m > 1$
$\uparrow$	$m > 0$
1	$m = 1$
0	$m = 0$
$\bot$	**ff**

$\oplus$	$\top$	$\Uparrow$	$\uparrow$	**1**	**0**	$\bot$
$\top$	$\top$	$\Uparrow$	$\uparrow$	$\uparrow$	$\top$	$\bot$
$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\bot$
$\uparrow$	$\uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	$\uparrow$	$\bot$
1	$\uparrow$	$\Uparrow$	$\Uparrow$	$\Uparrow$	**1**	$\bot$
0	$\top$	$\Uparrow$	$\uparrow$	**1**	**0**	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Table 5: Multiplicity membership, addition and ordering

Example 3.5 *The multiplicities in the UML are denoted as ranges $i..j$ where $i \in \mathbf{N}$ and $j \in \mathbf{N} \cup \{*\}$ with $i \leq j$ — where $*$ denotes "arbitrarily many" and satisfies $k < *$ and $*+k = k+* = *$ for all $k \in \mathbf{N}$. To model the inconsistent multiplicity as well (which is not considered in the UML) we introduce the special element $*..*$. This gives rise to an (interval based) multiplicity algebra with collective complements, much richer than $\mathbf{M}$ above:*

- $m : i..j$ *if and only if* $i \leq m \leq j$
- $\bigsqcap_c i_c..j_c$ *equals* $\max_c i_c.. \min_c j_c$ *if* $\max_c i_c \leq \min_c j_c$*, or* $*..*$ *otherwise.*
- **0** *is given by* $0..0$
- **1** *is given by* $1..1$.

It follows that

- $\lfloor i..j \rfloor = i$ *if* $i \neq *$ *and* $\lfloor *..j \rfloor = \omega$*;*
- $\lceil i..j \rceil = j$ *if* $j \neq *$, $\lceil i..* \rceil = \omega$ *if* $i \neq *$ *and* $\lceil *..* \rceil = 0$*;*
- $\bot$ *equals* $*..*$ *and* $\top$ *equals* $0..*$*;*
- $i_1..j_1 \oplus i_2..j_2$ *equals* $(i_1{+}i_2)..(j_1{+}j_2)$.
- $\overline{0..j}$ *equals* $\{\,(j{+}1)..*\,\}$ *and* $\overline{i..j}$ *for* $i > 0$ *equals* $\{\,0..(i{-}1), (j{+}1)..*\,\}$.

In fact, the UML multiplicities form the largest *interval based multiplicity algebra: for any interval in $I \subseteq \mathbf{N}$ of the form $I = \{i, \ldots, j\}$ where $i \in \mathbf{N}$ and $j \in \mathbf{N} \cup \{\omega\}$ there is a UML multiplicity μ such that $\mathbf{N}^\mu = I$.*

3.2 Local shape logic

On the basis of a multiplicity algebra we now define *local shape logic*, LSL^T, as follows:

$$\begin{aligned} \xi &::= v \mid \xrightarrow{a} v \mid \xleftarrow{a} v \mid \xrightarrow{a} . \\ \phi &::= {}^{\mu}[\xi] \mid \neg\phi \mid \phi \vee \phi \mid \forall_v \phi . \end{aligned}$$

ξ stands for a (node or edge set) *expression*; these are used in combination with abstract multiplicities $\mu \in \mathbf{M}$. Such expressions may contain explicit node identities $v \in N_T$; moreover, some of them also use an implicit *current node*, which is the node of G (the instance graph) on which the expression is being evaluated. We recognise:

- *Node instance expressions* of the form v, which stands for the instances of v;

LSL^T	SL^T	
${}^{\mu}[\xi]$	$\mathbf{ff}$	if $\mu = \bot$
	$\exists^{\lfloor\mu\rfloor} y\colon v_\xi.\ \varphi_\xi$	if $\lceil\mu\rceil = \omega$
	$\exists^{\lfloor\mu\rfloor} y\colon v_\xi.\ \varphi_\xi \wedge \nexists^{\lceil\mu\rceil+1} y\colon v_\xi.\ \varphi_\xi$	otherwise
$\forall_v \phi$	$\forall x\colon v.\ \phi$	

ξ	v_ξ	φ_ξ
v	v	$\mathbf{tt}$
$\xrightarrow{a} v$	v	$x \xrightarrow{a} y$
$\xleftarrow{a} v$	v	$y \xrightarrow{a} x$
$\stackrel{a}{-}$	arbitrary	$x \stackrel{a}{-}$

Table 6: Translation (de-sugaring) from LSL^T to SL^T

- *Outgoing edge expressions* of the form $\xrightarrow{a} v$, which stands for the a-labelled edges from the current node to some instance of v;
- *Incoming edge expressions* of the form $\xleftarrow{a} v$, which stands for the dual concept — the a-edges from v-instances to the current node.
- *Unary edge expressions* of the form $\stackrel{a}{-}$, which stands for the (empty or singleton) set of a-labelled unary edges from the current node. Again, we sometimes use $\xrightarrow{a}\bot$ to denote $\stackrel{a}{-}$.

Again, there is an implicit typing condition imposed by T: the formula ϕ under a quantifier $\forall_v \phi$ may only contain expressions $\xrightarrow{a} w$, $\xleftarrow{b} u$ and $\stackrel{c}{-}$ for $(v, a, w), (u, b, v), (v, c, \bot) \in E_T$. We use $\exists_v \phi$ to denote the dual of $\forall_v \phi$.

Example 3.6 *The constraints in Example 3.1 have equivalent counterparts in* LSL^T*:*

(1′). $\forall_{v_1}({}^{\uparrow}[\stackrel{\mathsf{List}}{-}] \wedge ({}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_3] \vee {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_3]))$

(2′). $\forall_{v_2}({}^{\uparrow}[\stackrel{\mathsf{Cell}}{-}] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{val}} v_4])$

(3′). $\forall_{v_2}({}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{next}} v_3] \vee {}^{\mathbf{0}}[\xrightarrow{\mathsf{next}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_3])$

(4′). $\forall_{v_2}({}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{0}}[\xleftarrow{\mathsf{next}} v_2] \vee {}^{\mathbf{0}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{next}} v_2])$

(5′). ${}^{\mathbf{1}}[v_3] \wedge \forall_{v_3}{}^{\uparrow}[\stackrel{\mathsf{Null}}{-}]$

(6′). $\forall_{v_4}{}^{\uparrow}[\stackrel{\mathsf{Elem}}{-}]$

(7′). $\exists_{v_4}{}^{\mathbf{0}}[\xleftarrow{\mathsf{val}} v_2]$

The following example shows that LSL^T is indeed less expressive than SL^T.

Example 3.7 *The constraints in Example 3.2 do* not *have an equivalent counterpart in* LSL^T*. Informally, the reason is that they express structural properties involving more than two nodes at a time.*

LSL^T formulae can be regarded either as sugared SL^T-formulae, or through an independent semantics. We show the two interpretations and their equivalence.

Local shape logic as a fragment of shape logic. We first show how LSL^T-formulae can be seen as special (sugared) SL^T-formulae. In this view, there is a specific variable x that stands for the "current node". Table 6 defines the de-sugaring translation from LSL^T to SL^T, using two auxiliary functions on expressions, $v_\xi \in N_T$ to retrieve the node of the type graph involved in an expression, and $\varphi_\xi \in \mathsf{SL}^T$ to retrieve the de-sugared formula underlying the expression.

For instance, de-sugaring the LSL-constraints in Example 3.6 according to Table 6 yields the original constraints in Example 3.1.

A direct semantics for local shape logic. Alternatively, we can interpret LSL^T directly. The semantics of the expressions ξ is given by the following function, where $\tau\colon G \to T$ is a typing and $u \in N_G$:

$$\begin{aligned}
[\![\xrightarrow{a} v]\!]_{\tau,u} &= \{(u,a,w) \in E_G \mid v = \tau(w)\} \\
[\![\xleftarrow{a} v]\!]_{\tau,u} &= \{(w,a,u) \in E_G \mid v = \tau(w)\} \\
[\![\overset{a}{-}]\!]_{\tau,u} &= \{(u,a,\bot) \in E_G\} \\
[\![v]\!]_{\tau,u} &= \{w \in N_G \mid v = \tau(w)\}
\end{aligned}$$

We now define a satisfaction relation $\tau, u \vDash_{\mathsf{LSL}} \phi$ for $\phi \in \mathsf{LSL}^T$. The rules for negation and disjunction are as always; we just show the special constructors of local shape logic.

$$\begin{array}{ll}
\tau, u \vDash_{\mathsf{LSL}} {}^{\mu}[\xi] & \text{if } \#[\![\xi]\!]_{\tau,u} : \mu \\
\tau, u \vDash_{\mathsf{LSL}} \forall_v \phi & \text{if } \tau, w \vDash \phi \text{ for all } w \in \tau^{-1}(v)\ .
\end{array}$$

The following theorem states that the direct and indirect semantics coincide. Henceforth, we will rely on the direct semantics and drop the suffix $_{\mathsf{LSL}}$.

Theorem 3.8 *For all local shape constraints $\phi \in \mathsf{LSL}^T$, typings $\tau\colon G \to T$ and nodes $u \in N_G$, $\tau, \{u/x\} \vDash \phi$ if and only if $\tau, u \vDash_{\mathsf{LSL}} \phi$.*

Theory and notation of local shape logic. We introduce some more concepts and notational shorthand for LSL. In the following, let ϕ, ψ are two LSL^T-formulae, let $\tau\colon G \to T$ be an arbitrary typing and let $u \in N_G$ be arbitrary.

- ϕ is *closed* if all sub-formulae ${}^{\mu}[\overset{a}{-}]$, ${}^{\mu}[\xrightarrow{a} w]$ and ${}^{\mu}[\xleftarrow{a} w]$ are in the scope of some $\forall_v$. (Note that this is the case iff the de-sugared SL-version of ϕ is closed, i.e., contains no free variables.)
- ${}^{\mu}[\Xi]$ for a finite set of expressions Ξ will express that the *sum* of the multiplicities of the expressions in Ξ equals μ, i.e., $\mu = \bigoplus_{\xi \in \Xi} \mu_\xi$ where for all $\xi \in \Xi$, μ_ξ is the smallest multiplicity w.r.t. $\sqsubseteq$ such that ${}^{\mu_\xi}[\xi]$. Alternatively,

$$ {}^{\mu}[\{\xi_i\}_i] \equiv \bigvee_{\mu = \bigoplus_i \mu_i} \bigwedge_i {}^{\mu_i}[\xi_i] \tag{10}$$

  For the multiplicity theory in this paper this comes down to

$$\begin{array}{lcl}
{}^{\top}[\{\xi_i\}_i] & \Leftrightarrow & \bigwedge_i {}^{\top}[\xi_i] \\
{}^{\Uparrow}[\{\xi_i\}_i] & \Leftrightarrow & \bigvee_i ({}^{\Uparrow}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\top}[\xi_j]) \vee \bigvee_{i \neq j} ({}^{\uparrow}[\xi_i] \wedge {}^{\uparrow}[\xi_j] \wedge \bigwedge_{j \neq k \neq i} {}^{\top}[\xi_k]) \\
{}^{\uparrow}[\{\xi_i\}_i] & \Leftrightarrow & \bigvee_i ({}^{\uparrow}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\top}[\xi_j]) \\
{}^{\mathbf{1}}[\{\xi_i\}_i] & \Leftrightarrow & \bigvee_i ({}^{\mathbf{1}}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\mathbf{0}}[\xi_j]) \\
{}^{\mathbf{0}}[\{\xi_i\}_i] & \Leftrightarrow & \bigwedge_i {}^{\mathbf{0}}[\xi_i] \\
{}^{\bot}[\{\xi_i\}_i] & \Leftrightarrow & \bigvee_i ({}^{\bot}[\xi_i] \wedge \bigwedge_{j \neq i} {}^{\top}[\xi_j])
\end{array}$$

- ${}^{\mu}[\xrightarrow{a} V]$ for a finite set of nodes V is equivalent to ${}^{\mu}[\{\xrightarrow{a} v \mid v \in V\}]$ and ${}^{\mu}[\xleftarrow{a} V]$ is equivalent to ${}^{\mu}[\{\xleftarrow{a} v \mid v \in V\}]$

For instance, we may abbreviate constraint (4′) of Example 3.6 to $\forall_{v_2}{}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1 \mid \xleftarrow{\mathsf{next}} v_2]$ and constraint (3′) to $\forall_{v_2}{}^{\mathbf{1}}[\xrightarrow{\mathsf{next}}\{v_2, v_3\}]$. As another example, in the circular buffer type graph T in Figure 3 we should have $\forall_{v_2}{}^{\mathbf{1}}[\xrightarrow{\mathsf{empty}} \mid \xrightarrow{\mathsf{val}} v_3]$.

Apart from the usual axioms of Boolean algebra, we have the following special local shape logic properties:

$$
\begin{aligned}
{}^{\top}[\xi] &\Leftrightarrow \mathbf{tt} && (11)\\
{}^{\perp}[\xi] &\Leftrightarrow \mathbf{ff} && (12)\\
{}^{\mu_1}[\xi] \wedge {}^{\mu_2}[\xi] &\Leftrightarrow {}^{\mu_1 \sqcap \mu_2}[\xi] && (13)\\
\neg\, {}^{\mu}[\xi] &\Leftrightarrow \textstyle\bigvee_{\nu \in \bar{\mu}} {}^{\nu}[\xi] && (14)\\
{}^{\mathbf{0}}[\xrightarrow{a}] &\vee {}^{\mathbf{1}}[\xrightarrow{a}] && (15)
\end{aligned}
$$

$$
\begin{aligned}
\forall_v(\phi_1 \wedge \phi_2) &\Leftrightarrow \forall_v\phi_1 \wedge \forall_v\phi_2 && (16)\\
\forall_v(\mathbf{ff}) &\Leftrightarrow {}^{\mathbf{0}}[v] && (17)\\
\forall_v(\phi_1 \vee \phi_2) &\Leftrightarrow \phi_1 \vee \forall_v\phi_2 \quad \text{if } \phi_1 \text{ is closed} && (18)\\
\forall_v\phi &\Rightarrow \forall_v(\phi \vee \psi) && (19)\\
\exists_v\phi \wedge \forall_v\psi &\Rightarrow \exists_v(\phi \wedge \psi) && (20)\\
\forall_v(\phi \vee \psi) &\Rightarrow \exists_v\phi \vee \forall_v\psi && (21)
\end{aligned}
$$

We will show that satisfiability, implication and equivalence of local shape logic are decidable. An important role in the proof is played by the *normal form* of a formula. Since the normal forms themselves also give some more intuition into the possibilities and limitations of LSL^T, we present them separately.

Proposition 3.9 (LSL normal form) *Assume that* $\mathbf{M}$ *has collective complements and let* $T = (N, E)$ *be an arbitrary graph. For every closed formula* $\phi \in \mathsf{LSL}^T$ *there is an equivalent formula of the form*

$$\bigvee_{c \in C} \bigwedge_{v \in N} \left({}^{\mu^{v,c}}[v] \wedge \left(\bigwedge_{i \in I_{v,c}} \exists_v \lambda^{v,i} \right) \wedge \forall_v \left(\bigvee_{i \in I_{v,c}} \lambda^{v,i} \right) \right) . \qquad (22)$$

In this formula, C is a (possibly empty) set of cases, *each $\mu^{v,c}$ stands for the multiplicity of v ($\in N$) in case c ($\in C$), $I_{v,c}$ is an index set, and each $\lambda^{v,i}$ is a conjunction of edge predicates, of the form*

$$\lambda^{v,i} = \left(\bigwedge_{e=(w,a,v) \in E} {}^{\mu^{e,i}_{\leftarrow}}[\xleftarrow{a} w] \right) \wedge \left(\bigwedge_{e=(v,a,w) \in E} {}^{\mu^{e,i}_{\rightarrow}}[\xrightarrow{a} w] \right) \qquad (23)$$

with incoming edge multiplicities $\mu^{e,i}_{\leftarrow}$ and outgoing/unary edge multiplicities $\mu^{e,i}_{\rightarrow}$. All multiplicities are consistent, i.e., not $\perp$; moreover, unary edge multiplicities are in $\{\top, \mathbf{1}, \mathbf{0}\}$.

Proof sketch. ϕ can be transformed to normal form in several steps.

1. Remove negations. This is done by moving negation "inside", applying (14) at the predicate level.

2. Remove all closed predicates from under $\forall_v$- and $\exists_v$-quantifiers. This is done using (16)–(18) and the dual properties for $\exists_v$, together with the usual boolean axioms for conjunction and disjunction.

3. Rewrite the resulting formula to the form

$$\bigvee_{p\in P}\bigwedge_{v\in N}\phi^{v,p}$$

where for all $p \in P$ and $v \in N$

$$\phi^{v,p} = \Big(\bigwedge_{k\in K_{v,p}} {}^{\mu^k}[v]\Big) \wedge \Big(\bigwedge_{j\in J_{v,p}} \exists_v \psi^j\Big) \wedge \forall_v \psi^{v,p}$$

where the $K_{v,p}$ and $J_{v,p}$ are finite index sets and $\psi^{v,p}$ and ψ^j for $j \in J_{v,p}$ are non-closed (and hence quantifier-free) formulae over edge predicates. This is done using standard laws of predicate logic.

4. Rewrite each $\phi^{v,p}$ above to the form

$${}^{\mu^{v,p}}[v] \wedge \bigvee_{c\in C_{v,p}}\Big(\bigwedge_{i\in I_c} \exists_v \lambda^{v,i}\Big) \wedge \Big(\forall_v \bigvee_{i\in I_c} \lambda^{v,i}\Big)$$

where the $C_{v,p}$ and I_c are finite index sets and all $\lambda^{v,i}$ are conjunctions of edge predicates. This is possible due to (13) and Lemma 3.11.

5. Rewrite each $\lambda^{v,i}$ above to the form in (23) using (11)–(13).

6. Finally, distribute $\bigwedge_{v\in N}$ over $\bigvee_{c\in C_{v,p}}$ to put ϕ in disjunctive form as in (22). □

The following is an auxiliary result for Lemma 3.11.

Lemma 3.10 *For all finite sets of* LSL*-formulae* Ψ*, the following equivalence holds:*

$$\forall_v \bigvee \Psi \iff \bigvee_{\Phi\subseteq\Psi}\Big(\bigwedge_{\phi\in\Phi} \exists_v \phi \wedge \forall_v \bigvee \Phi\Big)$$

Proof sketch. The right-to-left implication is a direct consequence of (19). The other direction is essentially due to (21), which implies that for arbitrary $\phi \in \Psi$

$$\forall_v \bigvee \Psi \Longrightarrow \exists_v \phi \vee \forall_v \bigvee (\Psi \setminus \phi)$$

and hence

$$\forall_v \bigvee \Psi \Longrightarrow \bigwedge_{\phi\in\Psi} (\exists_v \phi \vee \forall_v \bigvee(\Psi \setminus \phi)) \ .$$

From this we may conclude

$$\forall_v \bigvee \Psi \Longrightarrow (\bigwedge_{\phi\in\Psi} \exists_v \phi \wedge \forall_v \bigvee \Psi) \vee \bigvee_{\phi\in\Psi} \forall_v \bigvee (\Psi \setminus \phi) \ .$$

The proof proceeds by induction on the size of Ψ. □

Lemma 3.11 *Assume that* $\mathbf{M}$ *has collective complements, and let* J *be a finite index set. If* ψ *and* ψ^j *for* $j \in J$ *are quantifier-free* LSL*-formulae over edge predicates, then*

$$\phi \;=\; \left(\bigwedge_{j\in J} \exists_v \psi^j\right) \wedge \forall_v \psi$$

is equivalent to an LSL*-formula of the form*

$$\bigvee_{c\in C}\left(\bigwedge_{i\in I_c} \exists_v \lambda^i\right) \wedge \left(\forall_v \bigvee_{i\in I_c} \lambda^i\right) \tag{24}$$

where C *is a finite index set,* I_c *is a finite index set for all* $c \in C$*, and all* λ^i *are conjunctions of edge predicates.*

Proof sketch. First note that ϕ can be rewritten as follows, due to (20):

$$\phi \iff \left(\bigwedge_{j\in J} \exists_v \psi \wedge \psi^j\right) \wedge \forall_v \psi$$

For all $K \subseteq J$ let

$$\psi^K \;=\; \psi \wedge \bigwedge_{k\in K} \psi^k \wedge \bigwedge_{k\in K\setminus J} \neg\psi^k \;.$$

It follows that $\psi \wedge \psi^j \Leftrightarrow \bigvee_{j\in K\subseteq J} \psi^K$ for all $j \in J$, and $\psi \Leftrightarrow \bigvee_{K\subseteq J} \psi^K$. As a consequence ϕ can be rewritten again:

$$\phi \iff \left(\bigwedge_{j\in J} \bigvee_{j\in K\subseteq J} \exists_v \psi^K\right) \wedge \left(\forall_v \bigvee_{K\subseteq J} \psi^K\right)$$

Now for all $K \subseteq J$ let $\bigvee_{i\in I_K} \lambda^i$ be the disjunctive normal form of ψ^K, where all I_K are disjoint index sets and all λ^i are conjunctions of edge predicates. This exists due to (14). It follows that

$$\phi \iff \left(\bigwedge_{j\in J} \bigvee_{j\in K\subseteq J, i\in I_K} \exists_v \lambda^i\right) \wedge \left(\forall_v \bigvee_{K\subseteq J, i\in I_K} \lambda^i\right)$$

By distributing $\wedge$ over $\vee$ this can be rewritten to the form

$$\phi \iff \bigvee_{h\in H}\left(\bigwedge_{i\in I_h} \exists_v \lambda^i\right) \wedge \left(\forall_v \bigvee_{i\in I} \lambda^i\right)$$

where $I = \bigcup\{I_K \mid K \subseteq J\}$ and $(I_h)_{h\in H}$ is an H-indexed family of index sets such that $I_h \subseteq I$ for all $h \in H$. Using Lemma 3.10, finally, this can be rewritten to

$$\phi \iff \bigvee_{h\in H, I_h\subseteq J\subseteq I} \left(\bigwedge_{i\in J} \exists_v \lambda^i\right) \wedge \left(\forall_v \bigvee_{i\in J} \lambda^i\right)$$

where the right hand side equals the required form (24) if we set $C = \{J \mid h \in H, I_h \subseteq J \subseteq I\}$ and $I_c = c$ for all $c \in C$. □

Theorem 3.12 *Let* $T \in \mathbf{G}$ *be arbitrary and let* $\mathbf{M}$ *be an arbitrary multiplicity algebra with collective complements. For every formula* $\phi \in \mathsf{LSL}^T$ *there is a set of integer programs* $\mathbf{S}_\phi$ *such that* ϕ *is satisfiable if and only if some* $(A, b) \in \mathbf{S}_\phi$ *admits a solution.*

Proof. Assume $T = (N, E)$, and let $\phi \in \mathsf{LSL}^T$. W.l.o.g. assume ϕ to be in normal form (see Proposition 3.9); i.e., $\phi = \bigvee_{c \in C} \phi^c$ where

$$\phi^c \;=\; \bigwedge_{v \in N} \Big({}^{\mu^{v,c}}[v] \wedge \Big(\bigwedge_{i \in I_{v,c}} \exists_v \lambda^{v,i} \Big) \wedge \forall_v \Big(\bigvee_{i \in I_{v,c}} \lambda^{v,i} \Big) \Big) \enspace .$$

and the $\lambda^{v,i}$ are as in (23). We will show that for each ϕ^c there is an integer program A, b such that ϕ^c is satisfiable if and only if A, b admits a solution; thus $\mathbf{S}_\phi$ consisting of all these A, b is the set of integer programs required in the theorem.

The vector of variables in A, b consists of cardinalities $m^v, m^{v,i}, m^e \in \mathbf{N}$ for all $v \in N$, $i \in I_{v,c}$ and $e \in E$, where m^v is the number of v-instances, $m^{v,i}$ the number of v-instances in sub-case i, and m^e the number of e-instances. These are required to satisfy the following constraints (if the upper bounds on the right hand side do not equal ω, otherwise there is no constraint):

$$\begin{array}{rcl}
m^v & = & \sum_{i \in I_{v,c}} m^{v,i} \\
m^v & \geq & \lfloor \mu^{v,c} \rfloor \\
m^v & \leq & \lceil \mu^{v,c} \rceil \\
m^{v,i} & \geq & 1 \quad \text{for all } i \in I_{v,c} \\
m^e & \geq & \sum_{i \in I_{src(e),c}} \lfloor \mu^{e,i}_{\leftarrow} \rfloor \cdot m^{src(e),i} \\
m^e & \leq & \sum_{i \in I_{src(e),c}} \lceil \mu^{e,i}_{\leftarrow} \rceil \cdot m^{src(e),i} \\
m^e & \geq & \sum_{i \in I_{tgt(e),c}} \lfloor \mu^{e,i}_{\rightarrow} \rfloor \cdot m^{tgt(e),i} \\
m^e & \leq & \sum_{i \in I_{tgt(e),c}} \lceil \mu^{e,i}_{\rightarrow} \rceil \cdot m^{tgt(e),i} \enspace .
\end{array}$$

We show below that this set of constraints has a solution in $\mathbf{N}$ if and only if ϕ^c is satisfiable.

if. Assume $\tau \models \phi_c$ for some typing $\tau : G \to T$. For all $u \in N_G$ let $i_u \in I_{\tau(u),c}$ be such that $\tau, u \models \lambda^{\tau(u),i_u}$. The above set of inequations is then satisfied by

$$\begin{array}{rcl}
m^v & = & \#\tau^{-1}(v) \\
m^{v,i} & = & \#\{u \mid \tau(u) = v, i = i_u\} \\
m^e & = & \#\tau^{-1}(e)
\end{array}$$

only if. Assume that we have a solution. This gives rise to an instance G of T satisfying ϕ^c as follows: The nodes and edges of G are numbered instances of T-nodes and T-edges, viz. $N_G = \bigcup_{v \in N_T} N^v_G$ and $E_G = \bigcup_{e \in E_T} E^e_G$ where

$$\begin{array}{rcl}
N^v_G & = & \{v_{i,j} \mid i \in I_{v,c}, 1 \leq j \leq m^{v,i}\} \\
E^e_G & = & \{e_k \mid 1 \leq k \leq m^e\} \enspace .
\end{array}$$

By construction there are partial injective mappings $f^e_{\leftarrow} : N^{src(e)}_G \rightharpoonup E^e_G$ and $f^e_{\rightarrow} : N^{tgt(e)}_G \rightharpoonup E^e_G$ for all $e \in E_T$, defined on those $v_{i,j}$ such that $\mu^{e,i}_{\leftarrow}$ (resp. $\mu^{e,i}_{\rightarrow}$) equals either $\mathbf{1}$ or $\uparrow$. Either $f^e_{\leftarrow}$ or $f^e_{\rightarrow}$ (but not both) may fail to be surjective, namely if $m^e > m^e_{\leftarrow}$, which implies that e is source unbounded at some $p \in I_{src(e),c}$, resp. $m^e > m^e_{\rightarrow}$, which implies that e is target

unbounded at some $q \in I_{tgt(e),c}$. On the basis of these mappings we define

$$\begin{array}{rcl} src_G : e_k & \mapsto & \begin{cases} v_{i,j} & \text{if } f^e_{\leftarrow}(v_{i,j}) = e_k \\ v_{p,1} & \text{if } e_k \notin cod(f^e_{\leftarrow}) \end{cases} \\ \ell_G : e_k & \mapsto & \ell_T(e) \\ tgt_G : e_k & \mapsto & \begin{cases} v_{i,j} & \text{if } f^e_{\rightarrow}(v_{i,j}) = e_k \\ v_{q,1} & \text{if } e_k \notin cod(f^e_{\rightarrow}). \end{cases} \end{array}$$

The typing $\tau\colon G \to T$ maps every $v_{i,j}$ to v. □

The decidability result follows immediately, using Theorem 2.5.

Corollary 3.13 (decidability) *For an arbitrary graph $T \in \mathbf{G}$, satisfiability of LSL^T is decidable.*

3.3 From integer programs to local shape logic

Having seen, in the Theorem 3.12, that local shape logic formulae can be reduced to sets of integer programs, it is natural to ask whether the inverse connection also holds, i.e., whether any integer program can be reduced to a local shape constraint. In the remainder of this section we will show that the answer is positive.

For an arbitrary integer program A, b (see Section 2.3) let $T_{A,b} = (N, E)$ where

$$\begin{array}{rcl} N & = & \{v_j \mid 1 \leq j \leq n\} \\ & & \cup \{v_{j,p} \mid 1 \leq j \leq n, 1 \leq p \leq \max|A_{-,j}|\} \\ & & \cup \{w_q \mid 1 \leq q \leq \max|b|\} \\ & & \cup \{u_i \mid 1 \leq i \leq m\} \\ E & = & \{(v_j, a, v_{j,p}) \mid 1 \leq j \leq n, 1 \leq p \leq \max|A_{-,j}|\} \\ & & \cup \{(u_i, c, v_{j,p}) \mid 1 \leq i \leq m, 1 \leq j \leq m, 1 \leq p \leq |A_{i,j}|\} \\ & & \cup \{(u_i, c, w_q) \mid 1 \leq i \leq m, 1 \leq q \leq |b_i|\}\ . \end{array}$$

N thus contains four kinds of nodes:

- v_j and $v_{j,p}$ for $1 \leq j \leq m$, corresponding to variable x_j. p ranges from 1 to the maximum coefficient that x_j receives in A. The edge multiplicities of the $(v_j, a, v_{j,p})$-edges will ensure that in a graph satisfying the shape constraint, each $v_{j,p}$ has the same number of instances as v_j; this number will be the solution in x_j.
- w_q, which will be constrained to have a single instance. q ranges from 1 to the maximum number occurring in b.
- u_i for $1 \leq i \leq n$, corresponding to the i'th equation. Each u_i has c-edges to variable nodes $v_{j,p}$ for $p = 1, \ldots, |A_{i,j}|$ and to constant nodes w_q for $q = 1, \ldots, |b_i|$. The shape constraints will enforce that the number of instances of u_i equals, on the one hand, the sum of the numbers of instances of all $v_{j,p}$ for $1 \leq p \leq A_{i,j}$ and of w_q for $1 \leq q \leq -b_i$ (which equals $\sum_{A_{i,j} \geq 0} A_{i,j} \cdot x_j + \sum_{b_i \leq 0} -b_i$) and, on the other hand, the sum of the numbers of instances of all $v_{j,p}$ for $1 \leq p \leq -A_{i,j}$ and of w_q for $1 \leq q \leq b_i$ (which equals $\sum_{A_{i,j} \leq 0} -A_{i,j} \cdot x_j + \sum_{b_i \geq 0} b_i$).

The corresponding shape constraint is given by

$$\phi_{A,b} = \bigwedge_{1 \leq j \leq m} \left(\lambda_j \wedge \bigwedge_{1 \leq p \leq \max|A_{-,j}|} \lambda_{j,p} \right) \wedge \bigwedge_{1 \leq q \leq \max b} \psi_q \wedge \bigwedge_{1 \leq i \leq n} \phi_i \tag{25}$$

where for all $1 \le j \le m$, $1 \le p \le \max |A_{_,j}|$, $1 \le q \le \max b$ and $1 \le i \le n$:

$$\begin{array}{rcl}
\lambda_j & = & {}^{\top}[v_j] \wedge \forall_{v_j} \bigwedge_{1 \le p \le |A_{i,j}|} {}^{\mathbf{1}}[\xrightarrow{a} v_{j,p}] \\
\lambda_{j,p} & = & {}^{\top}[v_{j,p}] \wedge \forall_{v_{j,p}} ({}^{\mathbf{1}}[\xleftarrow{a} v_j] \wedge \bigwedge_{1 \le i \le n, 1 \le p \le |A_{i,j}|} {}^{\mathbf{1}}[\xleftarrow{c} u_i]) \\
\psi_q & = & {}^{\mathbf{1}}[w_q] \wedge \forall_{w_q} \bigwedge_{1 \le i \le n, 1 \le q \le |b_j|} {}^{\mathbf{1}}[\xleftarrow{c} u_i] \\
\phi_i & = & {}^{\top}[u_i] \wedge \forall_{u_i} ({}^{\mathbf{1}}[\xrightarrow{c} \{v_{j,p} \mid 1 \le j \le n, 1 \le p \le A_{i,j}\} \cup \{w_q \mid 1 \le q \le -b_i\}] \wedge \\
 & & {}^{\mathbf{1}}[\xrightarrow{c} \{v_{j,p} \mid 1 \le i \le n, 1 \le j \le -A_{i,j}\} \cup \{w_q \mid 1 \le q \le b_i\}])
\end{array}$$

Theorem 3.14 *Let A, b be an integer program and $\phi_{A,b}$ the corresponding shape constraint as defined above. A vector $x \in \mathbf{N}^m$ is a solution to $Ax = b$ if and only if $f \vDash \phi_{A,b}$ for some graph morphism f such that $x_j = \#f^{-1}(v_j)$ for all $j = 1, \ldots, m$.*

Proof sketch.

If Assume $f \vDash \phi_{A,b}$ and let $x_j = \#f^{-1}(v_j)$ for $j = 1, \ldots, m$ and $y_i = \#f^{-1}(u_i)$ for $i = 1, \ldots, n$. For all $1 \le j \le m$ and $1 \le p \le \max |A_{_,j}|$, λ_j and $\lambda_{j,p}$ together imply

$$\begin{array}{rcl}
x_j & = & \#f^{-1}(v_j, a, v_{j,p}) \\
\#f^{-1}(v_{j,p}) & = & \#f^{-1}(v_j, a, v_{j,p}) \\
\#f^{-1}(v_{j,p}) & = & \#f^{-1}(u_i, a, v_{j,p}) \text{ if } 1 \le p \le |A_{i,j}|
\end{array}$$

It follows that

- $\sum_{1 \le p \le A_{i,j}} \#f^{-1}(u_i, c, v_{j,p})$ equals $A_{i,j} \cdot x_j$ if $A_{i,j} \ge 0$;
- $\sum_{1 \le p \le -A_{i,j}} \#f^{-1}(u_i, c, v_{j,p})$ equals $-A_{i,j} \cdot x_j$ if $A_{i,j} \le 0$.

Furthermore, ψ_q implies $\#f^{-1}(w_q) = 1$ and hence

$$\#f^{-1}(u_i, c, w_q) = 1 \text{ if } 1 \le q \le |b_i|.$$

Hence we have

- $\sum_{1 \le q \le b_i} \#f^{-1}(u_i, c, w_q)$ equals b_i if $b_i \ge 0$;
- $\sum_{1 \le q \le -b_i} \#f^{-1}(u_i, c, w_q)$ equals $-b_i$ if $b_i \le 0$.

Finally, ϕ_i implies

$$\begin{array}{rcl}
y_i & = & \sum_{1 \le j \le m} \sum_{1 \le p \le A_{i,j}} \#f^{-1}(u_i, c, v_{j,p}) + \sum_{1 \le q \le -b_i} \#f^{-1}(u_i, c, w_q) \\
 & = & \sum_{1 \le j \le m, A_{i,j} \ge 0} A_{i,j} \cdot x_j - \sum_{b_i \le 0} b_i \\
y_i & = & \sum_{1 \le j \le m} \sum_{1 \le p \le -A_{i,j}} \#f^{-1}(u_i, c, v_{j,p}) + \sum_{1 \le q \le b_i} \#f^{-1}(u_i, c, w_q) \\
 & = & \sum_{1 \le j \le m, A_{i,j} \le 0} -A_{i,j} \cdot x_j + \sum_{b_i \ge 0} b_i
\end{array}$$

It follows that $\sum_{1 \le j \le m} A_{i,j} \cdot x_j = b_i$.

Only if. For $i = 1, \ldots, n$ let $y_i = \sum_{1 \leq j \leq m, A_{i,j} \geq 0} A_{i,j} \cdot x_j + \sum_{b_i \leq 0} -b_i$. We define the following graph G:

$$\begin{aligned}
N_G &= \{v_{j,k} \mid 1 \leq j \leq m, 1 \leq k \leq x_j\} \\
&\quad \cup \{v_{j,p,k} \mid 1 \leq j \leq m, 1 \leq p \leq \max|A_{_,j}|, 1 \leq k \leq x_j\} \\
&\quad \cup \{w_q \mid 1 \leq q \leq \max|b|\} \\
&\quad \cup \{u_{i,k} \mid 1 \leq i \leq n, 1 \leq k \leq y_i\} \\
E_G &= \{(v_{j,k}, a, v_{j,p,k}) \mid 1 \leq j \leq m, 1 \leq p \leq \max|A_{_,j}|, 1 \leq k \leq x_j\} \\
&\quad \cup \{(u_{i,I_i(j,p,k)}, c, v_{j,p,k}) \mid 1 \leq i \leq n, 1 \leq j \leq m, 1 \leq p \leq |A_{i,j}|, 1 \leq k \leq x_j\} \\
&\quad \cup \{(u_{i,J_i(q)}, c, w_q) \mid 1 \leq i \leq n, 1 \leq q \leq |b_j|\}
\end{aligned}$$

Here $I_i\colon \mathbf{N}^3 \to \mathbf{N}$ and $J_i\colon \mathbf{N} \to \mathbf{N}$ for $1 \leq i \leq n$ are index functions that define the one-to-one-connection of instances of u_i to instances of $v_{j,p}$ and w_q, such that for all $1 \leq j \leq m$, $1 \leq p \leq |A_{i,j}|$, $1 \leq k \leq x_j$ and $1 \leq q \leq |b_j|$:

$$\begin{aligned}
I_i(j,p,k) &= \begin{cases} \sum_{1 \leq l < j, A_{i,l} \geq 0} A_{i,l} \cdot x_l + (p-1) \cdot x_j + k & \text{if } A_{i,j} \geq 0 \\ \sum_{1 \leq l < j, A_{i,l} \leq 0} -A_{i,l} \cdot x_l + (p-1) \cdot x_j + k & \text{if } A_{i,j} \leq 0 \end{cases} \\
J_i(q) &= \begin{cases} \sum_{1 \leq j \leq m, A_{i,j} \geq 0} A_{i,j} \cdot x_j + q & \text{if } b_i \geq 0 \\ \sum_{1 \leq j \leq m, A_{i,j} \leq 0} -A_{i,j} \cdot x_j + q & \text{if } b_i \leq 0 \end{cases}
\end{aligned}$$

The morphism $f\colon G \to T_{A,b}$, finally, is defined in the obvious way: it maps each $v_{j,k}$ to v_j, each $v_{j,p,k}$ to $v_{j,p}$, each w_q to w_q and each $u_{i,k}$ to u_i. It is now straightforward to check that $f \vDash \phi_{A,b}$. □

Example 3.15 *For instance, for the integer program consisting of the equations* $2x_1 + x_2 = 2$ *and* $x_2 - 3x_3 = -1$*, the type graph* $T_{A,b}$ *and shape constraint* $\phi_{A,b}$ *are shown in Figure 7.* $G_{A,b}$ *in this figure is a graph instance modelling the solution* $(x_1, x_2, x_3) = (0, 2, 1)$ *to this program, constructed according to the proof of Theorem 3.14.*

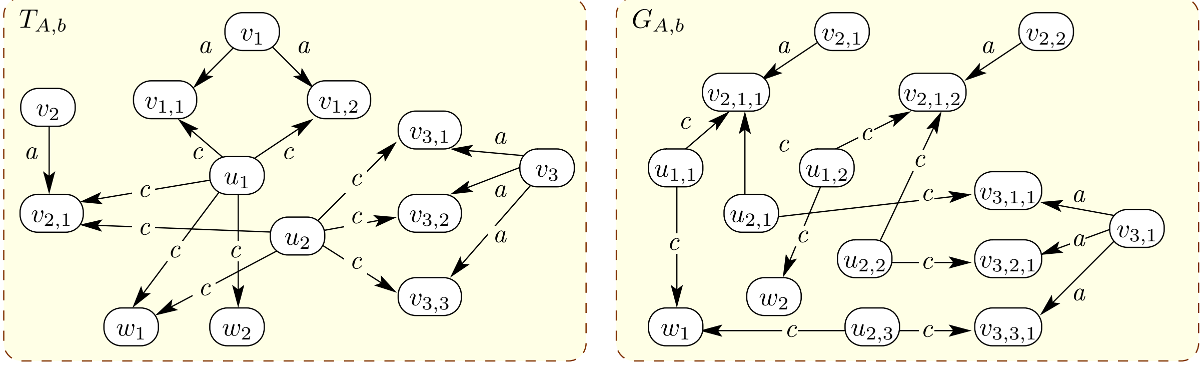
4 Shape graphs

We now define the combination of a type graph with a shape constraint; the resulting models will take over the role of type graphs (Definition 2.4) in capturing the intended structure of state graphs.

Definition 4.1 (shape graph) *A* shape graph *is a tuple* $\Gamma = (T, \phi)$ *where* T *is a type graph and* $\phi \in \mathsf{LSL}^T$ *a satisfiable local shape constraint. A* Γ-shaping *(or just* shaping*) is a typing* $\tau\colon G \to T$ *such that* $\tau \vDash \phi$*. We will call* Γ *a* shape of G *and* G *an* instance of Γ.

We use N_Γ, E_Γ, T_Γ, ϕ_Γ to denote the components of Γ. It should be noted that a shape graph is actually determined (almost) completely by the shape constraint: the type graph merely provides the collection of predicates to build the constraint from.

Any state graph has many different shapes; in fact, they form a hierarchy in which one shape is more *abstract* than another if it allows more instances.



$$
\begin{aligned}
\phi_{A,b} = \; & \forall_{v_1}({}^{\mathbf{1}}[\xrightarrow{a} v_{1,1}] \wedge {}^{\mathbf{1}}[\xrightarrow{a} v_{1,2}]) \wedge \forall_{v_2}{}^{\mathbf{1}}[\xrightarrow{a} v_{2,1}] \wedge \forall_{v_3}({}^{\mathbf{1}}[\xrightarrow{a} v_{3,1}] \wedge {}^{\mathbf{1}}[\xrightarrow{a} v_{3,2}] \wedge {}^{\mathbf{1}}[\xrightarrow{a} v_{3,3}]) \wedge \\
& \forall_{v_{1,1}}({}^{\mathbf{1}}[\xleftarrow{a} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_1]) \wedge \forall_{v_{1,2}}({}^{\mathbf{1}}[\xleftarrow{a} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_1]) \\
& \forall_{v_{2,1}}({}^{\mathbf{1}}[\xleftarrow{a} v_2] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_1] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_2]) \wedge \\
& \forall_{v_{3,1}}({}^{\mathbf{1}}[\xleftarrow{a} v_3] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_2]) \wedge \forall_{v_{3,2}}({}^{\mathbf{1}}[\xleftarrow{a} v_3] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_2]) \wedge \forall_{v_{3,3}}({}^{\mathbf{1}}[\xleftarrow{a} v_3] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_2]) \wedge \\
& {}^{\mathbf{1}}[w_1] \wedge \forall_{w_1}({}^{\mathbf{1}}[\xleftarrow{c} u_1] \wedge {}^{\mathbf{1}}[\xleftarrow{c} u_2]) \wedge {}^{\mathbf{1}}[w_2] \wedge \forall_{w_2}{}^{\mathbf{1}}[\xleftarrow{c} u_1] \wedge \\
& \forall_{u_1}({}^{\mathbf{1}}[\xrightarrow{c}\{v_{1,1}, v_{1,2}, v_{2,1}\}] \wedge {}^{\mathbf{1}}[\xrightarrow{c}\{w_1, w_2\}]) \wedge \forall_{u_2}({}^{\mathbf{1}}[\xrightarrow{c}\{v_{2,1}, w_1\}] \wedge {}^{\mathbf{1}}[\xrightarrow{c}\{v_{3,1}, v_{3,2}, v_{3,3}\}])
\end{aligned}
$$

Figure 7: The integer program $2x_1{+}x_2 = 2$, $x_2{-}3x_3 = -1$, with solution $(x_1, x_2, x_3) = (0, 2, 1)$

Example 4.2 *One possible shape for the graph H in Figure 1 is U in Figure 3 augmented with the formulae in Example 3.6. Another, more abstract typing is induced by the left hand graph in Figure 8, with a shape constraint containing the following subformula:*

$$\forall_{v_2}{}^{\mathbf{1}}[\xrightarrow{\mathsf{next}} v_2] \wedge ({}^{\mathbf{1}}[\xrightarrow{\mathsf{Cell}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{val}} v_3] \vee {}^{\mathbf{1}}[\xrightarrow{\mathsf{Null}} v_2]) \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1 \mid \xleftarrow{\mathsf{next}} v_2]$$

This shape graph is strictly more abstract because it allows instances containing arbitrarily many Null*-nodes, whereas the original shape graph specifies that there should be precisely one such. Yet another, more* concrete *typing for the list shape is shown on the right hand side of Figure 8: here the head* Cell*-node is explicitly distinguished from the others. The intended shape constraints for v_2 and v_3 should be clear and are omitted here.*

To formalise the notion of a more abstract shape graph we have to relate shape constraints over different type graphs. The following extends a morphism $f{:}\,T \to U$ to a mapping $f^{-1}{:}\,\mathsf{LSL}^U \to \mathsf{LSL}^T$ in the reverse direction — where $\xi\langle N\rangle$ denotes the expression ξ with N ($\subseteq \mathbf{N}$) replaced for v_ξ (see Table 6).

$$f^{-1}({}^{\mu}[\xi]) \quad = \quad {}^{\mu}[\xi\langle f^{-1}(v)\rangle]$$

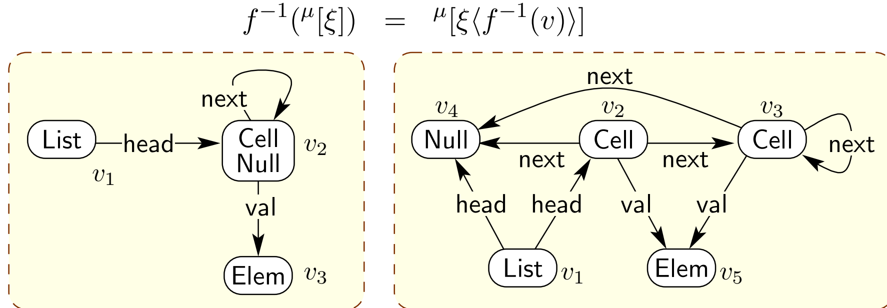


Figure 8: Alternative list shapes (see Example 4.2)

$$
\begin{array}{rcl}
f^{-1}(\neg\phi) & = & \neg f^{-1}(\phi_1) \\
f^{-1}(\phi \vee \psi) & = & f^{-1}(\phi) \vee f^{-1}(\psi) \\
f^{-1}(\forall_v \phi) & = & \bigwedge_{v=f(w)} \forall_w f^{-1}(\phi) \ .
\end{array}
$$

Definition 4.3 (shape abstraction) *Given two shape graphs* Γ, Δ, *a* shape abstraction morphism *(or just* abstraction*) from* Γ *to* Δ *is a morphism* $\alpha\colon T_\Gamma \to T_\Delta$ *such that* $\phi_\Gamma \Rightarrow \alpha^{-1}(\phi_\Delta)$. *The abstraction is* precise *(or* non-lossy*) if in fact* $\phi_\Gamma \Leftrightarrow \alpha^{-1}(\phi_\Delta)$.

We write $\alpha\colon \Gamma \to \Delta$ to denote that α is a shape abstraction morphism from Γ to Δ. Note that, as suggested by the use of the word "morphism", abstraction morphisms compose.

Example 4.4 *Taking into account the shape constraints discussed in Examples 3.6 and 4.2, the following mapping* α_1 *defines an abstraction from the right hand side of Figure 3 to the left hand side of Figure 8, whereas* α_2 *defines an abstraction from the right hand side of Figure 3 to the left hand side of Figure 8.*

$$
\begin{array}{rcl}
\alpha_1 & = & \{(v_1, v_1), (v_2, v_2), (v_3, v_2), (v_4, v_3)\} \\
\alpha_2 & = & \{(v_1, v_1), (v_2, v_2), (v_3, v_2), (v_4, v_3), (v_5, v_4)\} \ .
\end{array}
$$

The following is immediate.

Proposition 4.5 *If* $\tau\colon G \to \Gamma$ *is a shaping and* $\alpha\colon \Gamma \to \Delta$ *an abstraction morphism, then* $\alpha \circ \tau\colon G \to \Delta$ *is a shaping.*

A shape graph can be seen as a *specification* of the set of state graphs of which it is a shape. Alternatively we may consider *sets* of shape graphs. We call two sets of shape graphs, $\mathbf{\Gamma}$ and $\mathbf{\Delta}$, *equivalent* if they specify the same sets of instances:

$$
\mathbf{\Gamma} \equiv \mathbf{\Delta} \quad :\Leftrightarrow \quad \forall G \in \mathbf{G} : (\exists \Gamma \in \mathbf{\Gamma} : \tau\colon G \to \Gamma) \iff (\exists \Delta \in \mathbf{\Delta} : \tau\colon G \to \Delta) \ .
$$

For instance, if there exists a precise abstraction from Γ to Δ then $\{\Gamma\} \equiv \{\Delta\}$. Furthermore, if $\phi_\Gamma = \bigvee_c \phi_c$ then $\{T_\Gamma, \phi_c\}_c \equiv \{\Gamma\}$. Vice versa, we may define the *disjoint union* of a set of shape graphs, $\Delta = \biguplus \mathbf{\Gamma}$, by taking the disjoint union of the underlying type graphs T_Γ for $\Gamma \in \mathbf{\Gamma}$ and defining $\phi_\Delta \equiv \bigvee_{\Gamma \in \mathbf{\Gamma}} \phi'_\Gamma$ where ϕ'_Γ extends ϕ_Γ with a conjunct expressing $\mathbf{0}$-multiplicities for all nodes and edges not in T_Γ; then $\{\Delta\} \equiv \mathbf{\Gamma}$.

4.1 Monomorphic shapes

Due to the combination of conjunction and disjunction in shape constraints, shape graphs are not easy to visualise, or to reason about on an intuitive level — which may not be important from a formal point of view but makes shape graphs less useful when it comes to conveying ideas and principles. We now sketch a *monomorphic* fragment of local shape logic for which a visual representation can be given that we believe is easy to grasp; and we show that every shape graph is equivalent to a set of monomorphic shape graphs.

Definition 4.6 (monomorphic shape graph) *Let* Γ *be a shape graph.*

- Γ *is* pseudo-monomorphic *if there are partitionings* Λ, Π *of* N_Γ*, such that* ϕ_Γ *is equivalent to*

$$\begin{array}{ll} \bigwedge_{v \in N} & {}^{\mu^{[v]_\Lambda}}[[v]_\Lambda] \wedge \\ \bigwedge_{(v,a,w)\in E} & \forall_v {}^{\mu^{v,a,[w]_\Pi}}[\xrightarrow{a}\{u \in [w]_\Pi \mid (v,a,u) \in E\}] \wedge \\ & \forall_w {}^{\mu^{[v]_\Pi,a,w}}[\xleftarrow{a}\{u \in [v]_\Pi \mid (u,a,w) \in E\}] \end{array} \tag{26}$$

  where all multiplicities are consistent.

- Γ *is* monomorphic *if it is pseudo-monomorphicand* $\Lambda = \{\{v\}\}_{v\in N}$ *— so each* ${}^{\mu^{[v]_\Lambda}}[[v]_\Lambda]$ *can be replaced by* ${}^{\mu^v}[v]$*.*

Thus, the multiplicity of each edge $e = (v, a, w)$ in a [pseudo-]monomorphic shape graph is determined by precisely one (combined) incoming edge predicate $\xleftarrow{a}[v]_\Pi$ for w with multiplicity $\mu^{[v]_\Pi,a,w}$, and precisely one (combined) outgoing edge predicate $\xrightarrow{a}[w]_\Pi$ for v with multiplicity $\mu^{v,a,[w]_\Pi}$. Note that a [pseudo-]monomorphic shape graph is completely characterised by the type graph T, the partitioning Π [and Λ], and the multiplicities μ^v $[\mu[v]_\Lambda]$ for all $v \in N$ and $\mu^{v,a,[w]_\Pi}, \mu^{[v]_\Pi,a,w}$ for all $(v, a, w) \in E$.

We use the term *monomorphic* for such shape graphs because, in a sense, they stand for a single shape only — as evidenced by the fact that their shape constraints are in conjunctive form. In fact, the non-determinism in a monomorphic shape constraint is "pushed" into the syntactic sugar, i.e., the collective predicates of the form ${}^{\mu}[\Xi]$ where Ξ is a *set* of (incoming our outgoing edge) expressions. This is complemented by using *sets* of monomorphic shape graphs to represent states — which corresponds to non-determinism on the outermost level. Indeed, below we show that sets of monomorphic shape graphs are equally expressive as (general) shape graphs. In the meanwhile, the single-shape property gives rise to the desired graphical representation.

Example 4.7 *The shape graph consisting of U in Figure 3 as a type graph and the conjunction of* $(1')$–$(6')$ *in Example 3.6 as a shape constraint is* not *monomorphic: Constraint* $(4')$ *contains a disjunction* ${}^{\mathbf{1}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{0}}[\xleftarrow{\mathsf{next}} v_2] \vee {}^{\mathbf{0}}[\xleftarrow{\mathsf{head}} v_1] \wedge {}^{\mathbf{1}}[\xleftarrow{\mathsf{next}} v_2]$ *that cannot be rewritten to the appropriate form. Indeed, this is a typical example where the local shape is* not *singular: the constraint expresses that a* Cell*-node* either *has an incoming* head*-edge (if it is the first element of the list)* or *an incoming* next*-edge (if it is any element but the first). For the shape graph to be monomorphic, these two cases should be embodied in distinct nodes of the type graph.*

The more concrete type graph on the right hand side of Figure 8 does make this distinction and is, in fact, monomorphic.

Note that the similar disjunction ${}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_3] \vee {}^{\mathbf{0}}[\xrightarrow{\mathsf{head}} v_2] \wedge {}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} v_3]$ *in Constraint* $(1')$ *does* not *violate the principles of monomorphic shape, since it is equivalent to* ${}^{\mathbf{1}}[\xrightarrow{\mathsf{head}} \{v_2, v_3\}]$*.*

Graphical representation of monomorphic shapes. We can depict a monomorphic shape graph as follows:

- Nodes and edges with **0** multiplicity are omitted.
- Each node $v \in N$ receives an "outgoing edge port" for each a such that $(v, a, w) \in E$ (for some w); all outgoing a-edges are drawn as starting from this port, and the multiplicity $\mu^{v,a,[w]_\Pi}$ is written at the port as well — unless it equals $\top$, in which case it is omitted.
- Likewise, v receives an "incoming edge port" for each a such that $(w, a, v) \in E$ (for some w), where all incoming a-edges end; $\mu^{[w]_\Pi,a,v}$ is written there unless it equals $\top$.

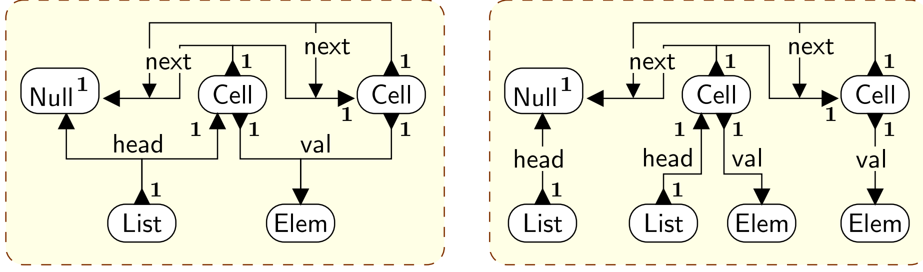


Figure 9: Two monomorphic list shapes

- Node multiplicities unequal to $\top$ are written inside the nodes.

For instance, the right hand shape graph of Figure 8 can be displayed as the left hand graph of Figure 9. The right hand graph also represents a list structure, but makes explicit distinctions between empty and non-empty lists, and between the elements contained in the first list cell and those contained in others. (The latter implies that there can be no sharing of Elem-nodes between head cells and subsequent cells; hence the left hand shape is more abstract than the right hand shape.) Note that the multiplicities are written at the other end of the edge than in the UML usage.

Another interesting observation is that the shape constraint $\phi_{A,b}$ corresponding to an integer program A, b (see (25)) is actually monomorphic. For instance, the graphical representation of Figure 7 is given in Figure 10. It follows from this (since shape constraints reduce to sets of integer programs and each integer program to a monomorphic shape graph) that *sets* of monomorphic shape graphs are equally expressive to arbitrary (sets of) shape graphs. We now give a direct statement.

Theorem 4.8 *For any shape graph there exists an equivalent set of monomorphic shape graphs.*

This can be proved in two steps: first we convert a shape graph into an equivalent set of pseudo-monomorphic shape graphs (Lemma 4.9), and then we convert each pseudo-monomorphic shape graph into an equivalent set of monomorphic shape graphs (Lemma 4.10).

Lemma 4.9 *For any shape graph there exists an equivalent set of pseudo-monomorphic shape graphs.*

Proof sketch. Let Γ be an arbitrary shape graph. Let the normal form of ϕ_Γ be given by $\bigvee_c \phi_c$ (see Proposition 3.9), and for all c let $\Gamma_c = (T_\Gamma, \phi_c)$. Clearly $\Gamma \equiv \{\Gamma_c\}_c$.

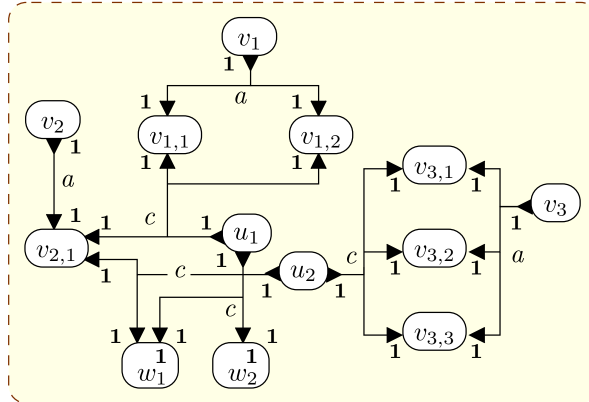


Figure 10: Monomorphic representation of the IP $2x_1+x_2 = 2$, $x_2-3x_3 = -1$

Now for each c define a new shape graph $\Delta_c = (U_c, \psi_c)$ where $U_c = (N_c, E_c)$ with

$$\begin{aligned} N_c &= \{(v,i) \mid v \in N_\Gamma, i \in I_{v,c}\} \\ E_c &= \{((v,i), a, (w,j)) \mid (v,a,w) \in E_\Gamma, i \in I_{v,c}, j \in I_{w,c}\} \end{aligned}$$

Thus, we split the nodes $v \in N_\Gamma$ according to the cases $I_{v,c}$ distinguished in the normal form. Let $\alpha \colon N_c \to N_\Gamma$ be defined by $(v,i) \mapsto v$ for all $(v,i) \in N_c$; define $\Lambda = \Pi = \{\alpha^{-1}(v) \mid v \in N_\Gamma\}$, and for all $v \in N_\Gamma$, $e = (v,a,w) \in E_\Gamma$, $i \in I_v$ and $j \in I_w$ let

$$\begin{aligned} \mu^{\alpha^{-1}(v)} &= \mu^{v,c} \\ \mu^{\alpha^{-1}(v),a,(w,j)} &= \mu_{\leftarrow}^{e,c,j} \\ \mu^{(v,i),a,\alpha^{-1}(w)} &= \mu_{\rightarrow}^{e,c,i} \end{aligned}$$

where the multiplicities on the left hand side are those required in Definition 4.6 and the multiplicities on the right hand side are those obtained from ϕ_c according to Proposition 3.9. Obviously this gives rise to a pseudo-monomorphic shape graph Δ_c; it can be shown that $\alpha \colon \Delta_c \to \Gamma_c$ is a precise abstraction, and hence $\Delta_c \equiv \Gamma_c$. It follows that $\mathbf{\Delta} = \{\Delta_c\}_c$ fulfils the proof obligation. □

Lemma 4.10 *For any pseudo-monomorphic shape graph there exists an equivalent set of monomorphic shape graphs.*

Proof sketch. Let Γ be an arbitrary pseudo-monomorphic shape graph. We have to remove combined instance predicates ${}^{\mu}[V]$. This is done by expanding all such combined predicates in ϕ_Γ according to (10) and rewriting the resulting formula to a disjunctive form $\bigvee_{c \in C} \phi_c$, equivalent to ϕ_Γ, where each ϕ_c is monomorphic. It follows that $\mathbf{\Delta} = \{(T_\Gamma, \phi_c)\}_c$ fulfils the proof obligation. □

For instance, although — as pointed out in Example 4.7 — the shape graph arising from U in Figure 3 is itself not monomorphic, it is equivalent to (the singleton set consisting of) the monomorphic shape graph in Figure 9. On the other hand, the left hand shape in Figure 8 gives rise to almost the same monomorphic shape graph, but without the **1**-multiplicity at the Null-node.

4.2 Canonical shapes

Among the many different (monomorphic) shapes of a given state graph, it is useful to single out one that can be generated from the graph automatically, and whose size is bounded, not by the size of the state graph but by some global constant. For this purpose, we introduce the notion of a *canonical shape*. The idea of a canonical shape is that nodes are distinguished only if their local structure is sufficiently different in the first place; otherwise, they should be identified by the shaping. Note that this is complementary to the idea underlying monomorphic shapes, which is about restricting the circumstances under which nodes may be *identified* — namely, only if their local structure is sufficiently similar.

To define the canonical shape of a graph, we just have to give a criterion for when two nodes are "sufficiently different." In this paper we take a very straightforward notion: the multiplicities of the sets of incoming and outgoing edges should be the same for each edge label. For an arbitrary graph $G \in \mathbf{G}$, define *characteristic* functions $\gamma_{\rightarrow}^G, \gamma_{\leftarrow}^G \colon N \times \mathbf{L} \to \mathbf{M}$ as follows:

$$\begin{aligned} \gamma_{\rightarrow}^G \colon (v,a) &\mapsto \#_\mu\{u \mid (v,a,u) \in E\} \\ \gamma_{\leftarrow}^G \colon (v,a) &\mapsto \#_\mu\{u \mid (u,a,v) \in E\} \ . \end{aligned}$$

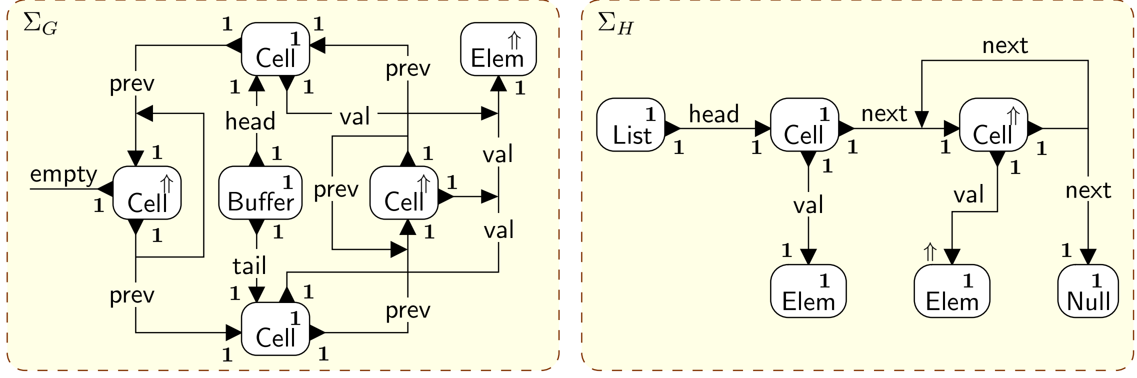


Figure 11: Canonical shapings of the circular buffer and linked list states in Figure 1

We consider $v, w \in N$ to be shape-indistinguishable if the characteristic functions yield the same result for all edge labels:

$$v \sim_G w \quad :\Leftrightarrow \quad \forall a \in \mathbf{L} : (\gamma^G_{\leftarrow}(v,a), \gamma^G_{\rightarrow}(v,a)) = (\gamma^G_{\leftarrow}(w,a), \gamma^G_{\rightarrow}(w,a)) \enspace .$$

The canonical shaping of a graph G will equate all $\sim_G$-equivalent nodes of G; the edge multiplicities are determined by the characteristic function.

Definition 4.11 (canonical shaping) *Let G be a graph and $i \in \mathbf{N}$. The* canonical shaping *of G is given by the projection morphism $\pi_{\sim_G}$ to the monomorphic shape graph Σ with $T_\Sigma = G/{\sim_G}$, $\Pi = \{N_G\}$ and for all $V \in N_\Sigma$ and $(V, a, W) \in E_\Sigma$*

$$\begin{aligned} \mu^V &= \#_\mu V \\ \mu^{(V,a,N_G)} &= \gamma^G_{\rightarrow}(v,a) \;\; \textit{for any } v \in V \\ \mu^{(N_G,a,W)} &= \gamma^G_{\leftarrow}(w,a) \;\; \textit{for any } w \in W. \end{aligned}$$

It follows from the definition of $\gamma^G_{\leftarrow}$ and $\gamma^G_{\rightarrow}$ that the choice of $v \in V$ resp. $w \in W$ in the side conditions above does not matter. It also follows that the only multiplicities that can possibly appear in a canonical shaping constructed according to this definition are $\mathbf{1}$ and $\Uparrow$.

Example 4.12 *Figure 11 shows the canonical shapes Σ_G and Σ_H of the circular buffer instance G and the list instance H in Figure 1. There are several noteworthy points:*

- *The (only)* Elem*-node in Σ_G has multiplicity $\Uparrow$ but incoming edge multiplicity $\mathbf{1}$. This indicates that there are multiple instances of this node, but they are not shared.*
- *Σ_H has two distinct* Elem*-nodes, one of which is shared whereas the other is not. The fact that their incoming edges start at distinct* Cell*-nodes is a coincidence due to the particular state graph of which this is a canonical shape: it is unlikely that this particular distinction is relevant in the application being modelled.*
- *Σ_H is strictly more concrete than both shape graphs in Figure 9.*

Characterising canonical shape graphs. By construction, canonical shapes are monomorphic; but we can actually characterise them more closely than that. This characterisation allows us to derive that the number of distinct (i.e., inequivalent) canonical shape graphs is finite.

Like for ordinary graphs, for an arbitrary monomorphic shape graph Γ we define characteristic functions $\gamma^\Gamma_\rightarrow, \gamma^\Gamma_\leftarrow: N \times \mathbf{L} \rightarrow \mathbf{M}$, as follows:

$$\begin{array}{rcl} \gamma^\Gamma_\rightarrow: (v,a) & \mapsto & \bigoplus \{\mu^{v,a,[w]_\Pi} \mid (v,a,w) \in E\} \\ \gamma^\Gamma_\leftarrow: (v,a) & \mapsto & \bigoplus \{\mu^{[w]_\Pi,a,v} \mid (w,a,v) \in E\} \ . \end{array}$$

As before, this gives rise to an equivalence over the nodes of the shape graph:

$$v \sim_\Gamma w \quad :\Leftrightarrow \quad \forall a \in \mathbf{L} : (\gamma^\Gamma_\leftarrow(v,a), \gamma^\Gamma_\rightarrow(v,a)) = (\gamma^\Gamma_\leftarrow(w,a), \gamma^\Gamma_\rightarrow(w,a)) \ .$$

A shape graph is called canonical if the underlying node partitioning equates all nodes, and node equivalence (defined as above) implies identity.

Definition 4.13 (canonical shape graph) *A monomorphic shape graph Γ is called* canonical *if $\Pi = \{N_\Gamma\}$, and $\sim_\Gamma$ is the identity relation over N_Γ.*

The following proposition states that this is indeed a proper characterisation of the canonical shapings obtained from Definition 4.11.

Proposition 4.14 *Σ_G is a canonical shape graph for any $G \in \Gamma$.*

It should perhaps be pointed out that, in contrast to monomorphic shapes, canonical shapes are not as expressive as general shape graphs. This can be seen for instance from the fact (proved below) that there are only finitely many canonical shape graphs, whereas the number of inequivalent LSL formulae, and thus the number of inequivalence shape graphs, is infinite. It should therefore not come as a surprise that the shapes used in Theorem 3.14 to encode integer programs (see (25)) are not canonical; e.g., in Figure 10, $v_{1,1} \sim v_{1,2}$, $v_{3,1} \sim v_{3,2} \sim v_{3,3}$ and $u_1 \sim u_2$. (Nor does it help to provide additional structure, for instance in the way of distinguishing node predicates, since the global set of labels $\mathbf{L}$ is required to be finite.)

We propose canonical shape graphs as a workable state space abstraction. The following theorem states that canonical shape graphs are bounded in size and number, although the general bound is way too large to lead to tractable state spaces.

Theorem 4.15

1. *The total number of canonical shape graphs is $O(2^{n^2 \cdot \#\mathbf{L}} \cdot \#\mathbf{M}^n)$ where $n = \#\mathbf{M}^{2\#\mathbf{L}}$.*
2. *The number of canonical shape graphs over a type graph (N,E) is $O(\#\mathbf{M}^{n+2m})$, where $n = \#N$ and $m = \min(\#E, \#N \cdot \#L)$.*

Proof.

1. The characteristic functions $\gamma^G_\leftarrow$ and $\gamma^G_\rightarrow$ have $\#\mathbf{M}^{\#\mathbf{L}}$ possible outcomes; so the number of nodes that $\sim_G$ can at most distinguish is $n = \#\mathbf{M}^{2\#\mathbf{L}}$. We need not consider graphs with fewer nodes since these are generated through $\mathbf{0}$ node multiplicities. Furthermore, over n nodes we can define $2^{n^2 \cdot \#\mathbf{L}}$ graphs. On each of these graphs only the μ^v are yet to be chosen, since the other multiplicities are fixed by $\gamma^G_\leftarrow$ and $\gamma^G_\rightarrow$; there are n such choices to be made.

2. The number of canonical shape graphs over (N,E) is equal to the number of possible combinations of μ^v for $v \in N$ and $\mu^{v,a,N}, \mu^{N,a,w}$ for $(v,a,w) \in E$. Of the former there are $\#\mathbf{M}^{\#N}$ and of the latter no more than $\#\mathbf{M}^{\#E}$ and also no more than $\#\mathbf{M}^{\#N \cdot \#L}$. $\square$

The upper bound can be improved by observing that the range of $\#_{\mu^-}$ actually is not $\mathbf{M}$ but something like a *base* $\underline{\mathbf{M}}$ of $\mathbf{M}$, which can be characterised as those elements of $\mathbf{M}$ that are not the conjunction of others. For instance, in our choice of multiplicity algebra, the base equals $\underline{\mathbf{M}} = \{\mathbf{0}, \mathbf{1}, \Uparrow\}$. It is the size of the base $\#\underline{\mathbf{M}}$ (= 3 in our case), rather than $\#\mathbf{M}$, that should be the actual parameter in Theorem 4.15.

This evil upper bound notwithstanding, canonical shape graphs do provide an automatic abstraction from possibly unbounded state graphs to a finite operational model. It will remain to be seen how well the abstraction performs in practice. The second part of the theorem shows that the situation already improves if we consider only *typed* state spaces in the first place, i.e., there is a "global" type graph. In practice, a likely scenario is that there is even a global shape graph of which all canonical shapes are instances.

5 Conclusion

We review some related work, sketch some possible extensions and variations on the theory presented here, and outline the next steps in our research programme.

5.1 Summary and related work

We discuss below some previous work, which is in some cases quite close to that presented in the current paper. In this comparison we stress similarities not differences. Before doing so, however, we point out some aspects of our approach that we have not seen in most of the other papers cited below:

- The explicit use of a multiplicity algebra. Besides leaning on an intuition fed by the common use of multiplicities in class diagrams, this makes the approach more flexible: to obtain better (i.e., more precise) shape constraints one just has to go to a more expressive multiplicity algebra.

  On the other hand, explicit counting quantifiers, as found in (some variants of) description logic and also in $\mathcal{C}^2$ — see below — clearly are at least as powerful as a multiplicity algebra.

- The combination of logic and type graphs; in particular, the fact that an LSL formula express properties of typings, i.e., of graph morphisms to the type graph associated with the formula, rather than of graphs directly. This is motivated by the fact that we find graphs a very appealing and concise model closely representing the kind of structures we are interested in analysing (viz., program states). The requirement that a typing exist clearly imposes a structural constraint on graphs; the same constraint, when expressed as a formula in first order logic, does not come close to offering the same appeal or conciseness. We have somewhat more to say on this point in the discussion on expressiveness, below.

- The graphical representation of monomorphic shapes. This reinforces the point made just now about the appeal and conciseness of graphs. For instance, the monomorphic shape in Figure 10, expressed as an explicit constraint in LSL, is the 6-line formula given in Figure 7 — which would be even longer if it were not accompanied by the type graph.

  Among the related work discussed below, only the shape graphs by Sagiv et al. have a comparable graphical representation.

- The automatic, bounded abstraction to canonical shapes. Note that this is only possible in the presence of a finite multiplicity algebra, so here the fact that a multiplicity algebra is less powerful than counting quantifiers is crucial.

Shape logics. The logic presented here is closely related to logics developed for *static analysis*, especially based on shape graphs, with Benedikt, Reps and Sagiv [5] as a prime example — but see also [14, 19]. The differences between their work and that presented here stem from the context in which it was carried out. As we stated in the introduction, our primary interest is the verification of systems on the basis of a graph grammar semantics. For that reason we concentrate on uniform graphs as state models. This contrasts with the approach of [5, 14, 19], which is more language-oriented and models states as *stores*, in which pointer variables are distinguished from field selectors. Our setup has led us to consider shape logic, SL, which is a first-order graph logic that is symmetric with respect to the direction of edge traversal. The logic L_r defined in [5], while sharing many characteristics of SL, contains predicates expressing the existence of paths between pointer variables and node sharing along (forward) paths from pointer variables — where, however, the pointer variables are fixed, i.e., no quantification is possible. This has the advantage of making L_r decidable, in clear contrast to SL. We believe that the decidable fragment LSL studied here is independent from L_r.

In a setting that is more similar to ours, Baldan, König and König [4] have recently defined a graph abstraction and monadic second-order graph logic for essentially the same purpose as SL. The differences are twofold: their graph abstractions are not graphs but Petri nets, and more importantly, their logic does not have nodes but edges as basic entities. Still, it will be very interesting to compare the approaches in more detail and investigate if a combination is possible and worthwhile.

Spatial logic. Another logic for graphs is studied by Cardelli, Gardner and Ghelli in [7]. This so-called *spatial* logic is intended for the purpose of *querying* graphs but looks to be suited also for *typing* them. It allows quantification over both nodes and edges and looks to be properly encompassing both SL and LSL. The expressiveness of spatial logic is between first-order and monadic second-order graph logic. The authors announce that decidability of their logic is under study.

Separation logic. Also fairly recently, O'Hearn, Reynolds and Yang have proposed, in a series of papers [23, 30, 27], a branch of logic for reasoning about dynamic storage allocation called *separation logic*. The primary purpose is to extend Hoare logic to a setting with mutable shared data. A prime feature of the logic is the ability to express the partitioning of a graph into disjoint subgraphs using a primitive combinator (*separating conjunction*) — a feature also present in spatial logic. Apart from this commonality, however, the work on spatial logic is more similar in spirit to that on shape logic by Sagiv et al., discussed above: both are explicitly concerned with the analysis of algorithms on data structures, and the logic is used to express shape invariants or, in the case of separation logic, pre- and post-conditions, of data storage spaces (heaps).

Description logic. Only when putting the finishing touch on this paper we became aware of quite a close connection with *description logic*, a long-standing approach to knowledge representation. A good overview can be obtained from the recent handbook [2]. In the terminology of Baader and Nutt [3], there is a close correspondence of LSL to $\mathcal{ALEN}$, the assertional language with existential quantification and number restrictions (where number restrictions are the counterpart

of our multiplicity algebra) — although there are minor discrepancies such as the fact that in [3], number restrictions may not appear on nodes but just on edges.

One important feature of description logic that is missing altogether from LSL is the *intersection of concepts* — which in our setting would be represented best by an *inheritance* or "*is-a*" relation between nodes. In the theory of graph rewriting, we have only seen this in a recent paper by Ferreira and Ribeiro [13]. Taking inspiration from description logic, node inheritance is an issue we intend to study more closely in the future.

Expressiveness and decidability. One of our main results is the decidability of LSL satisfiability (Corollary 3.13). In the light of what is known about certain fragments of first order logic this is not very surprising — even though we became aware of the facts recounted below only after finishing this work.

First note that, despite our choice (see above) to define the semantics of LSL over morphisms rather than graphs, there is a straightforward way to translate any shape graph, or even any SL^T-formula ϕ, to an equivalent (standard) first-order logic formula ψ, in the sense that a graph satisfies ψ if and only if it has a morphism to T satisfying ϕ. The translation consists of taking the node identities $v \in N$ of the type graph T as additional unary "node type" predicates and insisting upon $\forall x. \bigvee_{v\in N} x \overset{v}{-}$ (every node has an associated node type) and $\forall x.x \overset{v}{-} \Rightarrow \neg(x \overset{w}{-})$ for all $v, w \in N$ (node types are unique). Any formula of the form $\forall v{:}\, \phi.$ in SL^T then becomes $\forall x.v \overset{v}{-} \Rightarrow \phi$.

Clearly, then, any LSL formula containing just multiplicities $\uparrow$, i.e., just predicates of the form $^{\uparrow}[\xi]$, is equivalent to a formula of first order logic on two variables only, a fragment that is sometimes denoted $\mathcal{L}^2$. This fragment was shown a long time ago to be decidable by Scott [29] and later to have the finite model property by Mortimer [22]; more recently satisfiability was shown to be in NEXPTIME (and therefore in fact NEXPTIME-complete) [17]. Moreover, an *arbitrary* formula of LSL is equivalent to a formula in first-order logic on two variables *with counting quantifiers*, $\mathcal{C}^2$. This fragment, too, was shown decidable, in [18].

In fact, as a step in the decidability proof of $\mathcal{C}^2$, [18] proves that any closed formula in $\mathcal{C}^2$ is equivalent to a conjunction of sub-formulae of the forms (i) $\forall x.\exists^{<n}.\phi$, (ii) $\forall x.\exists^{\geq n}.\phi$ or (iii) $\forall x.\forall y.\phi$, where the ϕ are quantifier-free. Conjunction- and negation-free formulae of forms (i) and (ii) can be expressed directly within LSL, provided that $\mathbf{M}$ is rich enough to express the required multiplicities. However, deciding LSL enriched with formulae of form (iii) would require non-linear equations (the cardinality of the edge set equals the product of two node cardinalities), which would lead us beyond integer programs and hence would invalidate our proof strategy.

Thus, with hindsight, LSL is strictly less expressive than $\mathcal{C}^2$ and Corollary 3.13 is a consequence of a known decidability result.

5.2 Variations and extensions

The local shape logic as presented here can be extended in several respects while retaining its basic properties.

- The multiplicity algebra presented here can be extended. A stronger multiplicity algebra will lead to a refined notion of canonical shape (see also below), at the cost of increasing the size of the state space (see Theorem 4.15).
- We have formulated (local) shape logic on plain graphs. It might be worthwhile investigating an extension to *hyper-graphs*, in which the local edge predicates are of the form $a(\vec{v}, \bullet, \vec{w})$ rather than $\xrightarrow{a} v$ or $\xleftarrow{a} v$, where a is now an n-ary predicate (or hyper-edge) and $\vec{v}, \vec{w}$ are finite

vectors such that $\#\vec{v}+1+\#\vec{w} = n$. Again, we conjecture that this extension will not influence the decidability of the logic.

- The canonical shape of a graph G (Definition 4.11) is controlled by the notion of "sufficiently different" nodes. We have now chosen a very simple distinguishing criterion, which however can be refined in an elegant way using the idea of *bisimulation approximations* — actually, a mixture of *back-and-forth bisimulation* (see [10]) and *resource bisimulation* (see [8]). For this purpose redefine the characteristic functions of Section 4.2 to be parameterised by an existing equivalence:

$$\begin{aligned} \gamma^G_{\rightarrow}[\sim]\colon (v,a,u) &\mapsto \#_\mu\{u' \mid u' \sim u, (v,a,u') \in E_G\} \\ \gamma^G_{\leftarrow}[\sim]\colon (v,a,u) &\mapsto \#_\mu\{u' \mid u' \sim u, (u',a,v) \in E_G\} \ . \end{aligned}$$

  Now define $\sim^0_G = N_G \times N_G$ and for all $i \in \mathbf{N}$ let $\sim^{i+1}_G$ be the equivalence over N_G generated by the characteristic functions $\gamma^G_{\rightarrow}[\sim^i]$ and $\gamma^G_{\leftarrow}[\sim^i]$. Clearly the relation $\sim_G$ defined in Section 4.2 equals $\sim^1_G$. Instead one may use a finer approximation (a higher i) as the basis of the construction in Definition 4.11. It should be noted, however, that the upper bound for the state space size computed in Theorem 4.15 is non-elementary in i.

5.3 Future work

As stated in the introduction, this work is part of a project to develop verification methods for graph grammars. In that project, the following steps are:

- Studying the extension of shape graphs with node inheritance, as in [13] and Description Logic [2].
- Defining graph transformations over shape graphs, in such a way that the state graph transformations are (over-)approximated as closely as possible; that is, if G is transformed into H by some graph production rule P, then P should transform *any* shape of G into a shape of H; and if P transforms Γ into Δ then P should transform *some* instance of Γ into an instance of Δ.
- Defining a modal logic for reasoning about graphs that can be model checked on the level of shape graphs. A proposal for a logic is made in [26], but we also intend to study the connection and possible integration with [4]. Previous work with Distefano and Katoen [11, 12] suggests that abstract model checking may be feasible.
- Implementing shape graph transformation and model checking in a demonstrator tool; see [25] for a preliminary version of such a tool, which we intend to extend in this direction.

References

[1] M. Arends. Graph grammars for Java bytecode simulation. Master's thesis, University of Twente, 2003. In preparation. 2

[2] F. Baader, editor. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003. 28, 30, 31

[3] F. Baader and W. Nutt. Basic description logics. In Baader [2], chapter 2, pages 47–99. 28, 29

[4] P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In R. Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 255–272. Springer-Verlag, 2003. 28, 30

[5] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In S. D. Swierstra, editor, *Programming Languages and Systems — 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 2–19. Springer-Verlag, 1999. 28

[6] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution. In R. Heckel, T. Mens, and M. Wermelinger, editors, *Software Evulotion through Transformation*, volume 74 of *Electronic Notes in Theoretical Computer Science*, 2002. 2

[7] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer et al., editor, *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 597–610. Springer-Verlag, 2002. 28

[8] F. Corradini, R. De Nicola, and A. Labella. Graded modalities and resource bisimulation. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 1738 of *Lecture Notes in Computer Science*, pages 381–393. Springer-Verlag, 1994. 30

[9] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. v. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 193–239. Elsevier Science Publishers, 1990. 8

[10] R. De Nicola, U. Montanari, and F. W. Vaandrager. Back and forth bisimulations. In J. C. M. Baeten and J. W. Klop, editors, *Concur '90*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer-Verlag, 1990. 30

[11] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer Academic Publishers, 2002. 2, 30

[12] D. Distefano, A. Rensink, and J.-P. Katoen. Who is pointing when to whom: On model-checking pointer structures. CTIT Technical Report TR–CTIT–03–12, Department of Computer Science, University of Twente, 2003. Preliminary version. 2, 30

[13] A. P. L. Ferreira and L. Ribeiro. Towards object-oriented graphs and grammars. In U. Nestmann and P. Stevens, editors, *Formal Methods for Open Object-Oriented Distributed Systems (FMOODS)*, 2003. 29, 30

[14] P. Fradet and D. Le Métayer. Shape types. In *24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–39. ACM, ACM Press, 1997. 28

[15] M. Gogolla. Graph transformations on the UML metamodel. In J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, and J. B. Wells, editors, *ICALP Workshop on Graph Transformations and Visual Modeling Techniques*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000. 2

[16] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998. 2

[17] E. Grädel, P. G. Kolatis, and M. Y. Vardi. On the decision problem for two-variable first-order logic. *The Bulletin of Symbolic Logic*, 3(1):53–69, Mar. 1997. 29

[18] E. Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 306–317. IEEE, Computer Society Press, 1997. 29

[19] N. Klarlund and M. I. Schwartzbach. Graph types. In *Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 196–205. ACM, ACM Press, Jan. 1993. 28

[20] S. Kuska, M. Gogolla, R. Kollmann, and H.-J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *IFM 2002*, volume 2235 of *Lecture Notes in Computer Science*, pages 11–28. Springer-Verlag, 2002. 2

[21] A. Lozano. Graph grammars for Java simulation. Master's thesis, University of Twente, 2003. In preparation. 2

[22] M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975. 29

[23] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001. 28

[24] C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981. 7

[25] A. Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at `http://www.cs.utwente.nl/~groove`, 2003. 30

[26] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and S. L. Presti, editors, *Proceedings of the* 3rd *Workshop on Automated Verification of Critical Systems*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003. 30

[27] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE, Computer Society Press, 2002. 28

[28] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, 20(1):1–50, Jan. 1998. 2

[29] D. Scott. A decision method for validity of sentences in two variables. *J. Symb. Log.*, 27:477, 1962. 29

[30] H. Yang and P. O'Hearn. A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416. Springer-Verlag, 2002. 28