The Edge of Graph Transformation — Graphs for Behavioural Specification

Arend Rensink

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. The title of this paper, besides being a pun, can be taken to mean either the *frontier of research* in graph transformation, or the *advantage of using* graph transformation. To focus on the latter: Why should anyone not already educated in the field adopt graph transformation-based methods, rather than a mainstream modelling language or a process algebra; or vice versa, what is holding potential users back? These questions can be further refined by focusing on particular aspects like usability (available tools) or power (available theory).

In this paper, we take a fresh and honest look at these issues. Our perspective is the use of graph transformation as a formalism for the specification and analysis of system behaviour. There is no question that the general nature of graphs is at once their prime selling point (essentially everything can be specified in terms of graphs) and their main drawback (the manipulation of graphs is complex, and many properties that are useful in more specialised formalisms no longer hold for general graphs).

The outcome of this paper is a series of recommendations that can be used to outline a research and development programme for the coming decade. This may help to stimulate the continued and increasing acceptance of graph transformation within the rest of the scientific community, thereby ensuring research that is relevant, innovative and on the edge.

1 Background

In this paper we take a look at the advantages and disadvantages of graph transformation as an underlying formalism for the specification and analysis of system behaviour. For this purpose we review criteria for such a basic formalism, and we discuss how well graph transformation meets them in comparison with other formalisms.

We will start with a couple of observations. First of all, the field of graph transformation is quite broad: there are many formalisms that differ in philosophy as well as technical details, and yet all fall under the header of graph transformation — for an overview see the series of handbooks [65,23,26]. One often used classification distinguishes the algorithmic approach, pioneered by Nagl [53] among others, from the algebraic approach, pioneered by Ehrig among others [27]. We do not claim to be comprehensive in this paper; although we will not always state so explicitly, in our own research we are leaning towards the algebraic interpretation as the one closest to other, non-graph-based formalisms for behavioural specification, such as process algebra and Petri nets. Our findings are undoubtedly coloured by this bias.

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 6–32, 2010.

The next observation is that the phrase "graph transformation for the specification of system behaviour" is up for more than one interpretation.

Graph transformation as a modelling language. In this approach, system states are directly captured through graphs, and system behaviour through graph transformation rules. Thus, graph transformation is on the interface with the user (designer/programmer). This is the approach assumed by many graph transformation tools: prime examples are FUJABA [30], which has an elaborate graphical front-end for this purpose, and AUGUR2 [44], which requires rather specialised graph transformation rules as input.

Graph transformation for operational semantics. In this interpretation, system behaviour is captured using another modelling language, for which there exists a graph transformation-based semantics. Thus, graph transformation is hiding "under the hood" of the other modelling language. There has been quite some research in the definition of such operational semantics: for instance, for statecharts [45], activity diagrams [35,28] and sequence diagrams [36,11] and for object-oriented languages [12,41,64].

Note that this distinction can also be made outside the context of behavioural specification: for instance in model transformation, where again there are proposals to use graph transformation as a modelling language, for instance in the form of Triple Graph Grammars [70], or to define a semantics for QVT [47].

Both scenarios have their attractions; however, the criteria for the suitability of the underlying formalism do depend on which of them is adhered to. For instance, in the first scenario (graph transformation as a modelling language), the visual nature of graphs is a big advantage: in fact, many articles promoting the use of graph transformation stress this point. On the other hand, this criterion is not at all relevant in the second scenario (graph transformation for operational semantics) — at least not to the average user of the actual modelling language, though it may still be important to parties that have to draw up, understand or use the operational semantics. Instead, in this second scenario, graph transformation is competing with formalisms with far less intuitive appeal but more efficient algorithms, for instance based on trees (as in ordinary Structural Operational Semantics) or other data structures. In this paper we consider both scenarios.

In the next section, we first discuss some of the existing variations on graphs, emphasising their relative advantages. In Section 3 we proceed to step through a number of criteria that are (to a varying degree) relevant in a formalism for behavioural specification and analysis, and we evaluate how well graph transformation does in meeting these criteria. In Section 4, we evaluate the results and come to some recommendations on topics for future research.

Disclaimer. It should perhaps be stressed that this paper represents a personal view, based on past experience of the author, not all of which is well documented. Where possible we will provide evidence for our opinions, but elsewhere we do not hesitate to rely on our subjective impressions and intuitions.

2 A Roadmap through the Graph Zoo

Whatever the precise scenario in which one plans to use graph transformation, the first decision to take is the actual notion of graphs to be used. There are many possible choices, and they can have appreciable impact. In this section we discuss some of the dimensions of choice. In this, our perspective is coloured in favour of the so-called algebraic approach. A related discussion, which distinguishes between *glueing* and *connecting* approaches, can be found in [6].

We limit ourselves to directed graphs, as these are ubiquitous in the realm of graph transformation. In addition, essentially all our graphs are edge-labelled, though in some cases the labels are indirectly assigned through typing. True node labels will only appear in the context of typing, though a poor man's substitute can be obtained by employing special edges.[1] We will not bother about the structure of labels: instead, a single set Lab will serve as a universe of labels for all our graphs.

We will refrain from formally defining graph morphisms for all the notions of graph under review, as the purpose of this section is not to give an exhaustive formal overview but rather to convey intuitions.

2.1 Nodes for Entities, Edges for Relations

The natural interpretation of a graph, when drawn in a figure, is that the nodes represent entities or concepts in the problem domain. Thus, each node has an identity that has a meaning to the reader. In the context of the figure, the identities are determined by the two-dimensional positions of the nodes, but obviously these are not the same as the identities in the reader's mind: instead, the reader mentally establishes a mapping from the nodes to the entities in his interpretation. Typically, though not universally, this mental mapping is an injection; or, if we think of the mental model as encompassing only those entities that are depicted in the graph, it is a bijection.

The edges play quite a different role in such a natural interpretation. Edges serve to connect nodes in particular ways: every edge stands for a relation between its end nodes, where the edge label tells what kind of relation it is. Thus, one particular difference with nodes is that edges do not themselves have an identity. This is reflected by the fact that, in this interpretation, it does not make sense for two nodes to be related "more than once" in the same way: in other words, there cannot be *parallel edges*, with the same end nodes and the same label. One may also think of this as a *logical* interpretation, in the sense that the collection of edges with a given label is analogous to a binary predicate over the nodes.

This "natural" or "logical" interpretation is captured by the following definition.

Definition 1 (simple graph). *A simple graph is a tuple $\langle V, E \rangle$, where V is an arbitrary set of nodes and $E \subseteq V \times \mathsf{Lab} \times V$ is a set of edges. There are derived functions $src, tgt\colon E \to V$ and $lab\colon E \to \mathsf{Lab}$ projecting each edge onto, respectively, its first, third and second component.*

[1] Note that this is not necessarily true outside our context of using graphs for behavioural modelling: for instance, in the field of graph grammars, explicit node labels may be necessary to distinguish non-terminals.

Choosing this notion of graphs has important consequences for the corresponding notion of transformation.

- *Simple graphs do not fit smoothly into the algebraic framework.* (This is not really surprising, as simple graphs are relational models rather than algebraic ones.) Unless one severely restricts the addition and deletion of edges (namely, to rules where an edge is only added or deleted if one of its end nodes is at the same time also added or deleted), the theory of *adhesive categories* that is the flagship of algebraic graph transformation does not apply. Instead, algebraically the transformation of simple graphs is captured better by so-called *single pushout rewriting*, which is far less rich in results. See Section 2.7 below for a more extensive discussion on these issues.
- *Adding a relation may fail to change a graph.* One of the basic operations of a graph transformation rule is to add an edge between existing nodes. In simple graphs, if an edge with that label already exists between those nodes, the rule is applicable but the graph is unchanged. This is in some cases counter-intuitive. It may be avoided by including a test for the absence of such an edge.
- *Node deletion removes all incident edges.* In the natural notion of simple graph (single-pushout) rewriting mentioned above, nodes can always be deleted by a rule, even if they have incoming or outgoing edges that the rule does not explicitly delete as well. From the edges-are-relations point of view this can indeed be reasonable, as the fact that there exist relations to a certain entity should not automatically prevent that entity from being deleted; nevertheless, this means that the effect of a rule cannot be predicted precisely without knowing the host graph as well. Again, this may be avoided by including a test for the absence of such edges into the rule.

The closest alternative to simple graphs, which does not share the above characteristics, is the following.

Definition 2 (multigraph). *A multigraph is a tuple* $\langle V, E, src, tgt, lab \rangle$*, where* V *is an arbitrary set of nodes,* E *an arbitrary set of edges,* $src, tgt \colon E \to V$ *are source and target function and* $lab \colon E \to \mathsf{Lab}$ *an edge labelling function.*

Thus, in comparison to simple graphs, the source, target and labelling functions are now explicitly given rather than derived from E, and E itself is no longer a Lab-indexed binary relation but rather an arbitrary set. Thus, edges have their own identity, and parallel edges are automatically supported.

In the algebraic approach, multigraphs (in contrast to simple graphs) fall smoothly into the rich framework of adhesive categories and double-pushout rewriting. Among other things, this ensures that rule application is always reversible, which implies that the phenomena listed under the last two bullets above do not occur: if a rule adds an edge then the graph always changes under rule application (an edge is always added, possibly in parallel to an existing edge); and a node cannot be deleted unless all incident edges are explicitly deleted as well.

2.2 Nodification

Ordinary graphs, be they simple or multi, have very little structure in themselves. If they are used to encode models with rich structure, such as (for instance) design diagrams,

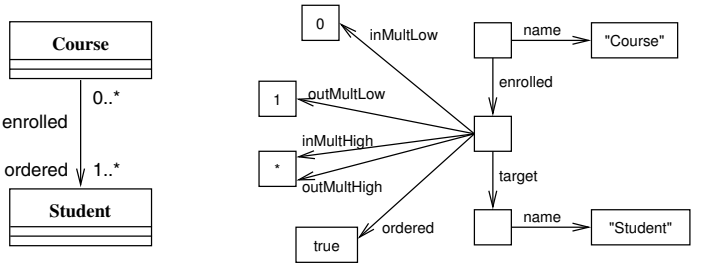

Fig. 1. A UML association type and its graph representation

then it may occur that the entity/relation intuition discussed above cannot be rigorously upheld. In particular, complex relationships, which should conceptually be represented by edges, may very well contain more information than what can be expressed by a single (binary) edge.

Example 1. For example, an association in a class diagram is conceptually a relation between two classes, but in addition to a name it typically has multiplicities and other modifiers (such as the fact that it is indexed or ordered). This is illustrated in Figure 1.

In such a case, the only solution is to introduce an additional node to capture the relation, with edges to the original source and target node, as well as additional edges to capture the remaining structure. We call this process *nodification*. Usually, one would prefer to avoid nodification, as it blows up the graphs, which has an adverse effect on visual appeal, understandability and complexity. This may be partially relieved by introducing a concrete syntax layer (syntactic graph sugar, such as edges with multiple labels) on top of the "real" graph structure, but that in itself also adds complexity to the framework.

Another context in which nodification occurs is in modelling relations among edges; for instance, if typing or traceability information is to be added to the graph. Rather than allowing "edges over edges", i.e., edges whose source or target is not a node but another edge, into the formalism itself, the typical solution is to nodify the edges between which a relation is to be defined.

Clearly, to minimise nodification it is necessary to move to a graph formalism with more inherent structure. A good candidate is *hypergraphs*.

2.3 Edges for Structure, Nodes for Glue

A hypergraph is a graph in which edges may have a different number of end nodes than two. Hypergraphs come in the same two flavours as binary graphs: simple and multi. However, especially multi-hypergraphs open up an intriguing alternative to the entity/relation intuition explored in Section 2.1, which has been exploited for instance in [2,3].[2]

[2] To complicate matters further, one can also extend simple graphs to n-ary rather than just binary relations. Though this is not a very popular model in graph transformation research, it is a faithful representation of, for instance, higher-degree relations in UML [55].

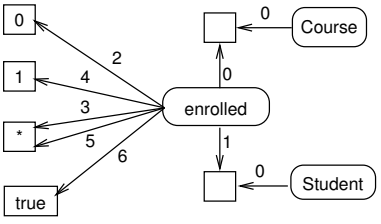

Fig. 2. Hyperedge representation of the right-hand graph of Figure 1

Definition 3 (hypergraph). *A hypergraph is a tuple* $\langle V, E, con, lab\rangle$*, where* $con\colon E \to V^*$ *is a connection function mapping each edge to its sequence of* tentacles*, and* $lab\colon E \to \mathsf{Lab}$ *is an edge labelling function. A node in a hypergraph is called* isolated *if it does not occur in the images of* con*.*

Usually, labels are used to determine the *arity*, i.e., the number of tentacles, of edges; that is, there is assumed to be an arity function $\alpha\colon \mathsf{Lab} \to \mathsf{Nat}$ such that $|con(e)| = \alpha(lab(e))$ for all $e \in E$. Though there is not an explicit notion of edge source, we can designate the first tentacle to be the source (which due to the possibility of nullary edges implies that src is a partial function).

Obviously, a binary (multi)graph is just a special case of a hypergraph in which all arities are 2. Edges with arity 0 and 1 are also potentially useful, to encode global state attributes and node labels, respectively. For instance, a 0-ary edge labelled Error (with no attached node) could be used to signify that the state is erroneous in some way; 1-ary edges Student (with source nodes only) can be used to label nodes that represent students. It is in the edges with higher arity, however, that extra structure can be encoded. For instance, the association edge of Figure 1 can be represented by a single, 7-ary edge, as shown in Figure 2. (In this figure we have followed the convention to depict hyperedges also as rounded rectangles, with numbered arrows to the tentacles. Thus, hyperedges look quite a bit like nodes.)

The shift in interpretation mentioned above lies in the fact that we may now regard edges to be the primary carrier of information, rather than nodes as in the entity/relation view. In particular, the entities can be thought of as represented by singular edges, like the Course- and Student-edge in Figure 2. The nodes then only serve to glue together the edges; in fact, they may be formally equated to the set of edge tentacles pointing to them. In particular, in this interpretation, an isolated node is meaningless and may be discarded ("garbage collected") automatically.

One of the attractions in this new interpretation is that it is no longer necessary to delete nodes: instead, it suffices to delete all incident edges, after which the node becomes isolated and is garbage collected. Thus, a major issue in the algebraic graph transformation approach becomes moot. (Formally speaking, for a rule that does not delete nodes, pushout complements are ensured to exist always.)

Example 2. One of the main sources of programming errors in C that is difficult to capture formally is the manual deallocation of memory. It is easy to erroneously deallocate memory cells that are still being pointed to from other parts of a program. However,

a graph transformation rule that attempts to delete such a cell will either remove all incident edges or be inapplicable altogether, depending on the way node deletion is handled. Neither captures the desired effect.

Using the interpretation discussed above (the glue/structure view, as opposed to the entity/relation view), however, this becomes straightforward to model: deallocation removes the singular edge encapsulating the value of a memory cell, but this does not give rise to node deletion as long as there is another ("stale") edge pointing to the node. However, any rule attempting to match such a stale edge and use its target node will fail (because the target node no longer carries the expected value edge) and hence give rise to a detectable error. The reallocation of improperly freed memory can be captured in this way as well.

Summarising: the special features of hypergraphs (coming at the cost of a more complex relation between edges and nodes) are:

- *Nodification can be avoided in many cases,* as hyperedges are rich enough to encode fairly complex structures.
- *Node deletion can be avoided altogether* if one follows the idea of garbage collecting isolated nodes, called the glue/structure interpretation above.

2.4 The Awkwardness of Attributes

Graphs are great to model referential structures, but do not model associated data of simple types (such as numbers, booleans and strings), usually called *attributes*, as conveniently. At first sight there is actually no problem at all: data values can easily be incorporated into graphs by treating them as nodes, and an attribute corresponds to an edge to such a data node. The problem is, however, that data values do not behave in all respects like nodes: they are not created or deleted, instead every data value always "exists" uniquely; moreover, the standard algebraic operations on these types can yield values from an infinite domain, which therefore formally must be considered part of every graph, even if it is not visualised.

Note that we have already implicitly used the representation of data values as nodes in Figures 1 and 2, which contain instances of natural numbers, booleans and strings.

In the last few years, the algebraic approach has converged on a formalisation (see [21]) in which data values are indeed special nodes, which are manipulated not through ordinary graph rules but through an extension relying on the standard algebras of the data types involved. Though satisfactory from a formal point of view, we still feel that the result is hybrid and conceptually somewhat awkward. This is to be regretted especially since competing formal verification techniques all concentrate on the manipulation of primitive data with very few higher-order structures (typically just arrays and records), which are then used to encode referential structures. This makes the comparison of strengths and weaknesses very skewed.

Edge attributes and nodification. Another issue is whether attributes should be restricted to nodes, or if edges should also be allowed to carry attributes. This is, in fact, connected to the choice of simple graphs versus multigraphs: only in multigraphs does the concept of edges with attributes make sense. Indeed, this capability in some cases

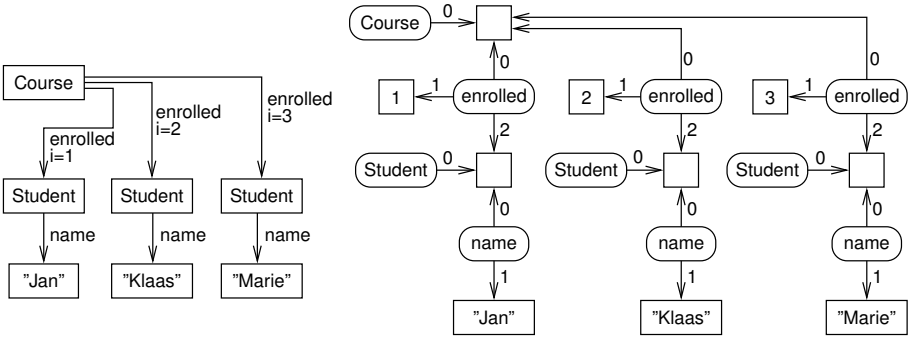

Fig. 3. Edge attributes or hyperedges for ordered edges

helps to avoid nodification, as we will show on an example below. However, the only way to capture edge attributes formally is to allow (attribute) edges to start at (non-attribute) edges. This makes the graph model more complex and less uniform, which feels like a heavy price to pay.

Example 3. A concept that comes up quite often in graph modelling is that of a list or sequence. For instance, the association shown in Figure 1 specifies that every Course has an associated *ordered* set of enrolled students. This means that an instance model may feature one Course with multiple associated Students, which are, moreover, ordered in the context of that Course.

One way to capture the ordering is to use indexing of the elements; and for this, edge attributes can be used. Apart from the label, every edge would receive a unique natural number-valued attribute, say i, as depicted by the left hand graph in Figure 3. The right hand side shows that the same effect can be achieved in hypergraphs.

The algorithmic alternative. The entire discussion above is driven by the intuition that attributes are just edges, albeit pointing to nodes with some special characteristics. Another point of view entirely is that attributes are distinct enough from edges to merit a different treatment altogether. For instance, attributes are not deleted and created, but are read and assigned; in the same vein, the concept of multiplicity typically does not apply to attributes.

In the algorithmic approach, there is in fact little objection to a separate treatment of edges and attributes: the latter are typically treated as local variables of a node. It is only in the algebraic approach that this meets objections. Depending on one's perspective, this can obviously be interpreted as a weakness of the algebraic approach, rather than the awkwardness of attributes.

Summarising: regarding attributes, we find that

– *The state-of-the-art algebraic formalisation of attributes feels awkward* and cannot compete with the ease of data manipulation in other verification techniques. On the other hand, there is no alternative that shares the same theoretical embedding, so if one values the algebraic approach then this is the only game in town.

– *Edge attributes come at a high conceptual cost,* which may be justified because they can help to avoid nodification. However, hypergraphs provide a more uniform way to achieve the same thing.
– *The algorithmic approach has no such difficulties* in incorporating attributes: they can be regarded as local variables of the nodes. This is a very pragmatic viewpoint, justification for which may be found in the fact that it reflects the practice in other modelling formalisms.

2.5 To Type or Not to Type

The graphs presented up to this point are not restricted in the way edges or edge labels can be used. In practice, however, the modelling domain for which graphs are used always constrains the meaningful combinations of edges. For instance, in the example of Figure 3 it would not make sense to include enrolled-edges between Student-nodes.

Such constraints are usually formulated and imposed through so-called *type graphs*. A type graph — corresponding to a *metamodel* in OMG terminology, see [54] — is itself a graph, typically in the same formalism as the object graphs, but often with additional elements in the form of inheritance and multiplicities (a formal interpretation of which can be found in [5,80]). In the presence of a type graph, a given object graph G is only considered to be correct if there exists a structure-preserving mapping τ, a *typing*, to the type graph T. Structure preservation means (among other things) that τ maps G-nodes to T-nodes and G-edges to T-edges.

In fact, we can include the typing itself into the definition of a graph. If, in addition, we select the nodes and edges of the type graph T from the set of labels (i.e., $V_T \cup E_T \subseteq \mathsf{Lab}$), then the typing subsumes the role of the edge labelling function: τ assigns a label not only to every edge of G but to every node as well.

Inheritance. Normally, a typing must map every edge consistently with its source and target node: that is, $\tau(src_G(e)) = src_T(\tau(e))$ and $\tau(src_G(e)) = src_T(\tau(e))$ for all $e \in E_G$. However, this can be made more flexible by including node inheritance. Node inheritance is typically encoded as a partial order over the nodes of the type graph. That is, there is a transitive and antisymmetric relation $\sqsubseteq \subseteq V_T \times V_T$; if $v \sqsubseteq w$ then we call v a subtype of w. Now it is sufficient if $\tau(src_G(e)) \sqsubseteq src_T(\tau(e))$ and $\tau(src_G(e)) \sqsubseteq src_T(\tau(e))$.

Since the purpose is to ensure that we are only dealing with correctly typed graphs, the matching as well as the application of transformation rules has to preserve types. Without going into formal detail (for which we refer to [5]), the intuition is that the type of a node in the rule's left hand side must be a supertype of the type of the matching host graph node. Thus, the rule can be defined abstractly (using general types) and applied concretely (on concretely typed graphs).

Example 4. Figure 4 shows a type graph (on the left), where the open triangular arrow visualises the subtyping relation $\mathsf{B} \sqsubseteq \mathsf{A}$. Of the four graphs on the right, both (i) and (ii) are correctly typed, and if (i) is the left hand side of a rule then it has two valid matches into (ii). (iii) is incorrect because the source type of the b-edge (A) is not a subtype of b's source (B).

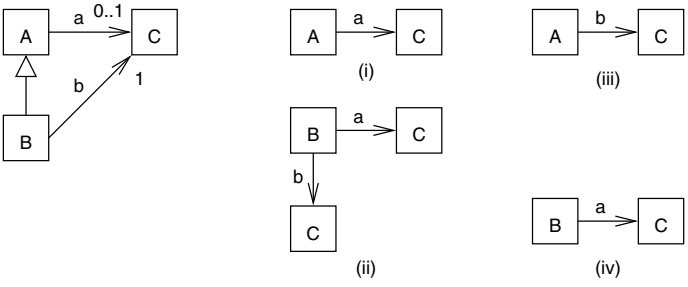

Fig. 4. A type graph (on the left) with correctly typed graphs (i) and (ii), and incorrectly typed graphs (iii) and (iv)

The above deals with node type inheritance only. In addition one may also consider a subtype or inheritance relation over edges. There is far less consensus on the usefulness and, indeed, meaning of edge type inheritance; see, for instance, [80] for one possible formalisation. In [42] we have shown that UML has three different notions of edge subtyping (subsets, redefinitions and unions). We therefore think of edge subtyping as a kind of special constraint, some more examples of which are briefly reviewed below.

Multiplicities. Also very common in type graphs are edge multiplicities. For instance, Figure 4 shows outgoing edge multiplicities, expressing that every A-node has at most one outgoing a-edge, and every B-node has *exactly* one outgoing b-edge. Graph (iv) is incorrect because it does not satisfy the multiplicity constraint on the b-edge.

Multiplicities can be attached both to outgoing and to incoming edges, and they specify "legal" ranges for the number of such edges in an object graph. In contrast to node inheritance, however, in general it is not decidable whether all graphs that can be constructed by a given transformation system are correct with respect to the edge identities. In such cases, the multiplicity constraint must be regarded as a property of which the correctness must be established through further analysis.

Special constraints. In general, as we have stated at the beginning of this subsection, typing is a way to impose constraints on graphs. Inheritance allows to refine these constraints, and multiplicities provide a way to strengthen them. However, the story does not end here: in the course of the years many special types of constraints have been proposed that can be seen as type enrichments. One has only to look at the UML standard [55] to find a large collection of such special constraints: abstractness, opposition, composition, uniqueness, ordering, as well as the edge subtype-related notions already mentioned above. An attempt to categorise these and explain them in terms of the constraints they impose can be found in [42]. Ultimately one can resort to a logic such as OCL to formulate application-specific constraints. It is our conviction that, where node inheritance and multiplicities have long proved their universal applicability and added value, one should be very reticent in adopting further, special type constraints.

Summarising: Regarding type graphs, we find that

– *Type graphs impose overhead on the formalism* as they have to be created, checked and maintained while creating an actual transformation system. Small prototype

graphs and rules can be developed faster if it is not a priori necessary to define a type graph and maintain consistency with it.
– *Type graphs strengthen the formalism* as they allow to document and check representation choices and provide a straightforward way to include node labels. Moreover, inheritance is a convenient mechanism for the generalisation of rules.

2.6 Special Graphs for Special Purposes

The classes of graphs we have discussed above impose very few artificial restrictions on the combinations of nodes and edges that are allowed. In fact, the only restriction is that in simple graphs parallel edges are disallowed. This lack of restrictions is usually considered to be part of the appeal of graphs, since it imparts flexibility to the formalism. (Of course, type graphs also impose restrictions, but those are user-defined rather than imposed by the formalism.)

However, there is a price to pay for this flexibility, in terms of time and memory: the algorithms and data structures for general graphs are more expensive than those for more dedicated data structures such as trees and arrays. If, therefore, the modelling domain at hand actually does not need this full generality, it makes sense to restrict to special graphs. Here we take a look at *deterministic* graphs.

Definition 4 (determinism). *We call a graph G* deterministic *if there is a* Lab*-indexed family of partial functions $(f_a \colon N_G \rightharpoonup E_G)_{a \in \mathsf{Lab}}$, such that*

$$lab_G(e) = a \wedge src_G(e) = v \;\Leftrightarrow\; f_a(v) = e \tag{1}$$

In other words, a graph is deterministic if, for every label a, every node has at most one outgoing a-labelled edge. Note that this is meaningful for all types of graphs we have discussed here (even hypergraphs) as they all have edge sources and edge labels. In terms of type graphs (Section 2.5), determinism corresponds to the specification of outgoing edge multiplicity 1 for all edges.

Advantages of determinism. Deterministic graphs offer advantages in time and memory consumption:

– There are cheaper data structures available. For instance, every node we can keep a fixed-sized list of outgoing edges, with one slot per edge label, holding the target node identity. The slot defaults to a null value if there is no outgoing edge with that label. For general graphs, instead we have to maintain a list of outgoing edges of indeterminate length.
– There are cheaper algorithms available. For instance, matching an outgoing edge will not cause a branch in the search space, as has been investigated in [16]. The same point has been made in [17], where so-called V-structures are introduced which share some characteristics of deterministic graphs.

Deterministic graphs in software modelling. In the domain of software modelling, edges typically stand for *fields* of structures or objects. Every field of a given object holds a value (which can be a primitive data value or a reference to another object).

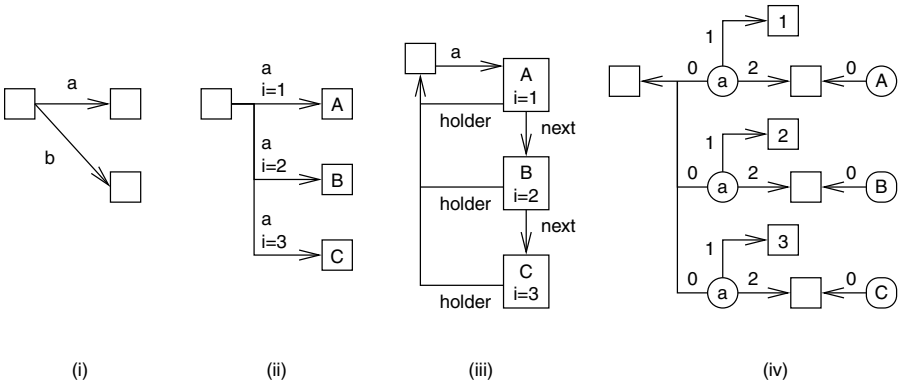

Fig. 5. (i) is a deterministic graph representing an object with fields a and b; (ii) is a non-deterministic edge-attributed graph modelling an array a; (iii) is a deterministic node-attributed graph, and (iv) a δ-deterministic hypergraph modelling the same array.

Such fields have names that are unique in the context of their holder (the structure or object); the field names are used as edge labels. Thus, the resulting graphs are naturally deterministic.

On the other hand, another common data structure is the *array*, which has indexed slots also holding values. Deterministic graphs are not very suitable to model such structures: the natural model of an array with local name a is a node with multiple outgoing a-edges, one for each slot of the array. A deterministic graph representation can be obtained by using a linked-list principle, possibly with backpointers. Both solutions are shown in Figure 5.

Maybe the most satisfactory solution is to use hypergraphs (in which the index is one of the tentacles, as in Figure 3) and broaden the concept of determinism.

Definition 5 (δ-determinism). *Let* $\delta\colon \mathsf{Lab} \to \mathsf{Nat}$ *be a* degree *function such that* $\delta(a) \leq \alpha(a)$ *for all* $a \in \mathsf{Lab}$. *A hypergraph* G *is called* δ*-deterministic if there is a* Lab*-indexed family of functions* $(f_a\colon V_G^{\delta(a)} \to E_G)_{a\in\mathsf{Lab}}$ *such that for all* $e \in E_G$*:*

$$lab_G(e) = a \wedge con(e)\!\downarrow_{\delta(a)} = \boldsymbol{v} \;\Leftrightarrow\; f_a(\boldsymbol{v}) = e$$

Here $\boldsymbol{v}$ denotes a sequence of nodes, and $-\!\downarrow_i$ for $i \in \mathsf{Nat}$ projects sequences to their first i elements. Thus, edges are no longer completely determined by their label and source node, but by their label and an initial subsequence of their tentacles (of length given by δ). Though this needs further investigation, we believe that some of the advantages of (strictly) deterministic graphs can be salvaged under this more liberal notion.

The standard notion of determinism (Definition 4) corresponds to degree 1 ($\delta(a) = 1$ for all $a \in \mathsf{Lab}$); graph (iv) in Figure 5 is δ-deterministic for $\delta(\mathsf{a}) = 2$ (and all other δ-values equal to 1).

2.7 The Pushout Scare

We feel it to be necessary to devote some words to an issue that, in our opinion, stands in the way of a broader acceptance of graph transformation, namely the intimate and inescapable connection of the algebraic approach to category theory. The only generally understood and accepted way to refer to the mainstream algebraic approach is the *double pushout* approach; the main alternative is called the *single pushout* approach. This automatically pulls the attention to the non-trivial pushout construction, a proper understanding of which requires at least a week of study to grasp only the bare essential concepts from category theory.

For those who stand above this theory, having mastered it long ago, it may be difficult to appreciate that this terminology repels potential users. In order to give more of an incentive to researchers from other areas to investigate graph transformation as a potentially useful technique, an alternative vocabulary must be developed, which does not suggest that knowing category theory is a prerequisite for using graph transformation. We propose the following:

Conservative graph transformation as an alternative term for the double pushout-approach. The term "conservative" refers to the fact that the deletion of graph elements carries some implicit application conditions, namely that all incident edges of deleted nodes are also explicitly deleted at the same time, and that the nodes and edges a rule attempts to delete are disjoint from those it intends to preserve. Thus, a rule may fail to be applicable even though a match exists.

Radical graph transformation as an alternative term for the single pushout-approach. The term "radical" refers to the fact that rules are always applicable (in a category where all pushouts exist, as they do for simple graphs) but the effect may go beyond what the rule explicitly specifies: dangling edges are deleted, and case of non-injective matches, nodes and edges may deleted that the rule appears to preserve.

2.8 Summary

In Table 1 we have summarised the five dimensions of the graph zoo discussed above. The dimensions are orthogonal: all combinations can be defined — however, some combinations make more sense than others. For instance, edge attribution in hypergraphs does not seem very useful, as additional tentacles provide a more uniform way to achieve the same modelling power. Other unlikely combinations are node attribution for hypergraphs, and edge attribution for simple graphs. Finally, note that the distinction between simple and multigraphs disappears if we restrict to deterministic graphs.

It should be noted that, again, we do not claim comprehensiveness of this overview. One dimension that we have omitted altogether is the notion of hierarchy, which by many researchers is considered to be an important enough notion to deserve a direct encoding into graphs; see, for instance [19,56], but also the distributed graphs of Taentzer [79,76] and Milner's bigraphs [51,52]. No doubt there are other dimensions we have missed.

Table 1. Dimensions of the graph zoo

Dimension	Value	Meaning	Advantages
Edge encoding	*simple*	no edge identity	edges are relations; simple
	multi	edges have own identity	better algebraic properties
Arity	*standard*	binary edges only	uniform, simple
	hyper	arbitrary tentacle count	avoids nodification
Attribution	*none*	no data attributes	simple
	nodes	only node attributes	necessary in practice
	all	node and edge attributes	avoids nodification
Typing	*no*	no explicit typing	simple, no overhead
	yes	type graph	documentation, node labels, inheritance
Determinism	*no*	general graphs	flexible
	yes	deterministic graphs	more efficient

3 Criteria for Behavioural Specification and Analysis

We will now step through a number of criteria for any formalism that is intended for specifying and analysing system behaviour. Behavioural analysis, in the sense used in this paper, at least includes some notion of state space exploration, be it partial, complete, abstract, concrete, guided, model checked, explicit, symbolic or a combination of those. Graph transformation tools that offer some functionality of this kind are, for instance, AUGUR2 [44], FUJABA [30] and GROOVE [58]; we want to stress, however, that we are more interested in the potential of the formalism than in actual tools.

As announced in the introduction, we will make a distinction based on whether the formalism is to be directly used as a modelling language, or as an underlying semantics for another language. For each of the criteria, we judge how well graph transformation does in comparison to other techniques — where we are thinking especially of SPIN [38], a state-of-the-art explicit-state model checker that uses a special-purpose language Promela for modelling.

3.1 Learning Curve

This term refers to the expected difficulties in learning to read and write specifications in a formalism or calculus. Another term might be "conceptual complexity." This in itself depends on a number of factors, not the least of which is editor tool support; but also general familiarity with the concepts of the formalism is important. Thus, a language like Promela, with concepts that are at least partly familiar from everyday programming languages, has a shallower learning curve than languages that heavily rely on more esoteric mathematical concepts.

The learning curve of graph transformation. Given adequate tool support for graph editing (which most graph transformation tools offer), the visual nature of specifications can appeal to intuition and make the first acquaintance with the formalism easy. However, there are several complications that require a more thorough understanding from the user, prime among which is the treatment of data attributes — see Section 2.4 for a more extensive discussion about this.

In general, we believe that the learning curve is directly related to the simplicity of the graph formalism: see Table 1 for our evaluation on this point. In particular, we have observed many times that novice users are *very fast* in creating prototype specifications (in a wide variety of domains) using GROOVE [58], which has about the simplest graph formalism imaginable. Hypergraphs, on the other hand, although offering several benefits (as argued in the previous section) definitely have a steeper learning curve.

Though we have complained above (Section 2.7) about the intimate connection with category theory, actually for the use of graph transformation in specifications an understanding of the underlying theory is not necessary; so we find that this does not put a burden on the learning curve.

Relevance. The learning curve of a formalism is, of course, very relevant if it is to be used directly as a modelling language: all users then have to be versed in reading and writing specifications in that language. However, the learning curve is less relevant if the formalism is to be used to define operational semantics. In the latter case, only a limited number of users will have to understand the formalism, namely those who have to understand or implement the semantics; and even fewer will have to write in it, namely those who define the semantics.

3.2 Suitability

It seems very obvious that a formalism to be used for specifying the behaviour of software systems should be especially suitable for capturing the specific features of software. For instance, the dynamic nature of memory usage (both heap and stack) and the referential (object) structures that are ubiquitous in most modern software should be natively supported by the formalism.

Yet it turns out that even formalisms that are not at all suitable in this sense, for instance because they only support fixed-size or very primitive data structures (like Promela), nevertheless do quite well in software verification. A prime example is the use of SAT-solvers for (bounded) software model checking (see [49,40]): the formulae being checked are extremely primitive and only finite behaviour can be analysed, but because the tools can deal with truly enormous datasets (they are very scalable, see Section 3.4 below), the results are quite good.

Suitability of graph transformation. Precisely the dynamic nature of software and its stress on referential structures make graph transformation an excellently suitable formalism. Indeed, this is one of it strongest points.

Relevance. Though, as pointed out above, there is a balance between suitability and scalability, there is no question that suitability (in the sense used here) is a very relevant criterion in judging a formalism.

3.3 Flexibility

Flexibility refers to the ease of modelling an arbitrary problem using a given formalism, without having to resort to "coding tricks" to make it fit. That is, if the model captures the problem without introducing extraneous elements, we call it suitable for

the modelling domain (as in Section 3.2 above), whereas if it does so for a broad range of domains, we call it flexible.

Flexibility of graph transformation. The flexibility of graph transformation is another of its strong points. Indeed, besides software analysis we have seen uses for GROOVE in the context of network protocol analysis, security scenarios, aspect-oriented software development, dynamic reconfiguration and model transformation.[3]

Relevance. In the dedicated domain of behavioural specification and analysis, the (general) flexibility of a formalism is less important than its (specific) suitability. Still, if the formalism is directly to be used as a modelling language, the ability to model concepts natively is very attractive. If, on the other hand, it is to be used for operational semantics, the importance of being flexible goes down, as the domain of application is then quite specialised.

3.4 Scalability

Scalability refers to the ability of a formalism to deal with large models. We typically call a formalism highly scalable if its computational complexity is a small polynomial in the model size. However, another notion of scalability refers to the *understandability* of larger models, i.e., their complexity to the human mind. We call this "visual scalability."

Visual scalability. Though small graphs (in the order of ten to twenty nodes) are typically easy to grasp and may provide a much more concise and understandable model than an equivalent textual description, this advantage is lost for larger graphs. Indeed, as soon as graphs start to be non-planar, even the best layouting algorithms cannot keep them from becoming very hard to understand. The typical solution in such a case is to decompose the graph into smaller ones, and/or to resort to a more text-based solution by introducing identifiers instead of edges.

Computational scalability. This is another term for computational or algorithmic complexity, which is a huge field of study on its own. In the context of behavioural analysis, algorithmic complexity is of supreme importance, as the models tend to become very large — this is the famous "state space explosion problem," which refers to the fact that state space tends to grow exponentially in the number of independent variables and in the number of independent components. Virtually all of the work in model checking is devoted to bringing down or cleverly representing the state space (through abstraction, compositionality or symbolic representation) or improving performance (through improved, concurrent algorithms).

Scalability of graph transformation. In general, graph transformation does not scale well, either visually or computationally. The former we have already argued above; for the latter, observe that the matching of left hand sides into host graphs, which is a

[3] As an aside, it should be mentioned that the flexibility of the graph transformation formalism does not only lie in the graphs themselves but also in the transformation rules and their composition into complete system specifications. In the current paper we have decided to omit this aspect altogether.

necessary step in rule application, is an NP-complete problem in the size of the left hand side, and polynomial in the size of the host graph where the size of the LHS (regarded as a constant) is the exponent. However, there are two alleviating factors:

- By using incremental algorithms, the complexity of matching becomes constant, at the price of more complex updating (i.e., part of the problem is shifted to the actual transformation). In [9] it is shown that this improves performance dramatically. Investigations in the context of verification are underway.
- In modelling software behaviour, typically matches do not have to be *found*: they are already determined by the context — as should be expected, given that programming languages are mostly deterministic (except for the effects of concurrency). In this setting, the complexity of matching may also be constant time: see [16].

In the context of state space exploration, another issue is the identification of previously encountered states. In general, this involves isomorphism checking of graphs, which is another (probably) non-polynomial problem. However, in [59,15] it has been shown that in practice isomorphism checking is feasible, and may give rise to sizable state space reduction in case the model has a lot of symmetry.

Relevance. Visual scalability is very relevant if the users need to visualise large models. Unless good decomposition mechanisms are available, this is indeed the case when the formalism is used as a modelling language, but less so if it is used for operational semantics.

As indicated above, computational scalability is supremely important, regardless of the embedding of the formalism.

3.5 Maturity

With maturity we refer to the degree in which all questions pertaining to the use of a formalism in the context of behavioural specification and analysis have been addressed and answered. We distinguish theoretical and practical maturity.

Theoretical maturity. We have already referred above (see Section 3.4) to the many existing techniques to combat the state space explosion problem: compositionality, abstraction and symbolic representations. We call a formalism theoretically mature if it is known if and how these techniques are applicable in the context of the formalism.

Practical maturity. Practical maturity, in this context, refers to the degree to which algorithms and data structures enabling the actual, efficient implementation of the formalism and corresponding analysis techniques have been investigated. Thus, we call a formalism practically mature if there is literature describing and evaluating such algorithms and data structures.

Maturity of graph transformation. Despite the impressive body of theoretical results in algebraic graph rewriting (recently collected in [22]), we think the formalism is not yet theoretically mature for the purpose of behavioural specification and analysis. Many of the concepts mentioned under Section 3.4 have hardly been addressed so far.

– *Abstraction.* Without some form of abstraction, it will never be possible to analyse arbitrary graph transformation systems, as these are typically infinite-state. Since it is not even decidable whether a given graph transformation system is finite-state, this is a severe restriction. Thus, we regard the existence of viable abstraction techniques as an absolute necessity for graph transformation to realise its full potential as a technique for behavioural specification and analysis. Consider: if all we can analyse is a finite, bounded fragment of the entire state space, then there is no advantage over existing model checking techniques in which the infinite-state behaviour cannot even be specified.

  The only fully worked out approach for abstraction, apart from several more isolated attempts described in [61,7], is the abstraction refinement work of [43], which is part of the Petri graph approach briefly mentioned in the discussion of hypergraphs (Section 2.3). Although the results are impressive, the rule systems that can be analysed are rather specialised (for instance, only rules that delete all edges of the left hand side are allowed). There is much more potential for abstraction techniques, as for instance the work on shape analysis (e.g., [66]) shows.
– *Compositionality.* This is another effective technique to combat state space explosion, which finds its origin in process algebra (see, e.g., [50]). A formalism is compositional if specifications of large, composed systems can be themselves obtained by composing models of subsystems (for instance, parallel components or individual objects). Those smaller models can then be analysed first; the results can be used to draw conclusions about the complete system without having to construct the entire state space explicitly.

  Graph transformation is, by nature, non-compositional. Instead it typically takes a whole-world view: the host graph describes the entire system. Though several approaches have been studied to glue together rules, most notably the work on rule amalgamation [75] (with the offshoot of distributed graph transformation in [79]), the best studied approach we know in which the host graph may be only partially known is the *borrowed context* work of [4]. This, however, falls short of enabling compositionality, since the host graph always grows, and never shrinks, by including a borrowed context; thus this does not truly reflect the behaviour of a single component. A very recent development on compositional graph transformation is [60].
– *Symbolic representation.* This refers to the implicit representation of a state space through a formula that holds on all reachable states and fails to hold on all unreachable ones. It has been shown that, by representing such formulae as Binary Decision Diagrams (BDDs), an enormous reduction with respect to an explicit state space representation can be achieved; see, e.g., [10].

  To the best of our knowledge, there have been no attempts whatsoever to develop similar symbolic representations for graphs. Clearly, the difficulties are great, as one of the main heuristics in the success of BDDs is finding a suitable ordering of the state vector; it is not at all easy to see how this translates to graph transformation, where the states are graphs which do not give rise to an unambiguous linearisation as vectors. For another thing, the graphs are not a priori bounded, which is a requirement for standard BDDs. Nevertheless, observing the success of BDDs in state-of-the-art model checking, they are certainly worth a deeper investigation.

As for practical maturity, fairly recently there has been increased attention for fast algorithms and data structures, especially for graph matching; for instance, see [32,39,9]. Another issue is isomorphism checking, which was studied in [59]. Furthermore, below we report on tool support for graph transformation, in the context of which there are continuous efforts to improve performance, stimulated by a series of tool contests.

Relevance. As we have indicated above, we believe that the theoretical maturity of the field is one of the prime criteria for the success of graph transformation for behavioural specification and analysis. It does not make a difference whether the formalism is to be used directly as a specification language or as an underlying operational semantics. As for practical maturity, this is also important, but it partially follows the theoretical insights.

3.6 Tooling

The final criterion we want to scrutinise is tooling. We discuss two aspects.

Tool support. In order to do anything at all with graph transformation in practice, as with any other formalism it is absolutely necessary to have tool support, in the sense of individual tools to create graphs and rules and perform transformations. Such tools should satisfy the normal usability requirements; in particular, they should be well enough supported for "external users" to work with them, and there should be some commitment to their stability and maintenance.

Standardisation. If we want to employ tools for graph transformation in a larger context, where the results are also used for other purposes, it is vital to have a means to communicate between tools. This requires a measure of interoperability, which in turn requires standardisation.

In fact one may distinguish two types of interoperability: one involves the interchange of graphs and rules, and the other involves communicating with other tools that are not graph-based. For the second type, the main issue is to adhere to externally defined formats; this can be solved on an individual bases by each tool provider. We will concentrate on the first type, which is a matter for the graph transformation community as a whole.

Tooling for graph transformation. Starting with PROGRES [69] and AGG [29] in the mid-80s, tool support for graph transformation has become increasingly available. We know of at least ten groups involved in a long-lasting, concerted effort to produce and maintain generally usable tools for graph transformation. Moreover, recently we have started a series of tool contest with the primary aim of comparing existing tools, and the secondary aim of creating a set of available case studies which can be used to try out and compare new tools or new functionality; see [62,63,48]. All this points to an extensive investment in tool support for graph transformation in general.

On the other hand, for the more specific purpose of behavioural specification and (especially) analysis, we find that the situation is less encouraging. The main special feature that is needed in this context is the ability to reason about all reachable graphs,

usually by exploring and analysing the state space systematically, rather than to efficiently apply a single sequence of rules (possibly with backtracking). Some (relatively) early work exists in the form of CHECKVML [68] and Object-Based Graph Grammars [18], both of which rely on a translation to the standard model checking tool SPIN [38] for the state space exploration. Our own tool GROOVE [58] has a native implementation of this functionality. A more recent development is AUGUR2 [44], which uses advanced results from Petri net theory to create a finite over-approximation (called an unfolding), already mentioned in the discussion on abstraction in Section 3.5, which can also cope with infinite state spaces. A more small-scale effort, also based on over-approximation, is reported in [8]. Yet another approach, based on backwards exploration and analysis, is implemented in the tool GBT [31]; see [67] for some results. Finally, [57] uses assertional (weakest-precondition) reasoning rather than state space exploration, which is also capable of dealing with infinite-state systems.

The (relative) scarcity of graph transformation tools for analysis is borne out by the fact that, in the last transformation tool contest [48], the "verification case" (concerning the correctness of a leader election protocol) received only 4 submissions.

The situation with respect to standardisation is not so good. In the past there have been serious attempts to define standards for the interchange of graphs (GXL , see [37]) and of graph transformation systems (GTXL , see [77,46]), both of which are XML-based standards; however, an honest assessment shows that there has not been much much activity of late, and the standards are not supported by many tools. We do not know of a single instance of such a standard being used to communicate between different tools.

A similar point is made in [34], where it is argued that the level on which GTXL has attempted to standardise is too low-level.

Relevance. It goes without saying that tool support is absolutely necessary for the success of graph transformation, if it is to be a viable formalism for behavioural specification and analysis. It may be less immediately obvious that the same holds for standardisation, in the sense used here (restricted to graph and rule standards). Yet it should be realised that there is a multitude of mature tools around, most of which are specialised towards one particular goal. A common, agreed-upon standard would enable tool chaining in which the strengths of each individual tool can be exploited to the maximum.

3.7 Summary

In Table 2 we summarise the results of this section, by scoring graph transformation on the criteria discussed above, and also rating the relevance of the criteria. We have used a scale consisting of the following five values, with a slightly different interpretation in the first two and the last two columns:

$--$ "Very weak", respectively "irrelevant."
$-$ "Meagre", respectively "less relevant."
0 "Reasonable", respectively "partially relevant."
$+$ "Good", respectively "relevant."
$++$ "Excellent", respectively "very relevant."

Table 2. Summary of criteria

Criterion		General graphs	Special graphs	Other techniques (SPIN)	Relevance for use as modelling language	Relevance for use as operational semantics
Learning curve		0	0	+	+	−
Suitability		++	+	−	+	0
Flexibility		++	0	−	+	−
Scalability	Visual	−−	−		+	−−
	Computational	−	+	++	++	++
Maturity	Theoretical	0	0	++	++	++
	Practical	+	+	++	+	+
Tooling	Support	+	+	++	++	+
	Standardisation	−	−	−	+	++

We have divided the criteria into those that are intrinsic to the formalism of graph transformation (the top five) and those that can be improved by further effort (the bottom four). We stress again that the scores are subjective.

Special graphs. The column "special graphs" in the table refers to the possibility of limiting graphs to a subclass with better computational properties. Here we are thinking especially of deterministic graphs (see Section 2.6). In Section 3.4 we have already cited [16] who point out that matching can be much more efficient for such graphs, in particular if it is already known where in the graph the next transformation is to take place, as is often the case when modelling software. Further evidence for the increased performance for special graphs is provided by the tool FUJABA [30], where the graph formalism is tuned for compilation to Java. For instance, the Sierpinski case study described in [78] shows that this can give rise to extremely high-performance code.

Other techniques. As stated at the start of this section, we have taken SPIN (see [38]) as a prototypical example of an "other technique" for the specification and analysis of system behaviour, because it is a well-known, mature tool that is actually being used successfully in practice. To our opinion, the qualities of SPIN are rather complimentary to those of graph transformation: it is not at all trivial to encode the desired behavioural aspects of a system into Promela, which is built upon a fixed set of primitives such as threads, channels and primitive data domains; however, once this task is done, the tool performs amazingly well. Surprisingly, SPIN , too, suffers from a lack of standardisation; it gets by because Promela itself is, in effect, an "industrial" standard.

Evaluation. There is no single strong conclusion to be drawn from this table. Here are some observations:

– Two of the criteria on which graph transformation does especially well, namely suitability and flexibility, are much less relevant when using graph transformation as an underlying semantics than when using it as a modelling language. The same holds for a criterion in which graph transformation does especially badly, namely visual scalability.

- Though theoretical maturity and standardisation are rated as average or worse, improving this is a matter of further research; the poor scalability, on the other hand, is intrinsic to graphs.
- The improved computational scalability of deterministic graphs may be enough to prefer those over arbitrary graphs, in the context of behavioural specification and analysis, also given that their suitability is hardly less than for general graphs. This is especially true if the formalism is to be used for operational semantics, since there the loss of flexibility is less relevant.

4 Recommendations

In this section we formulate a series of personal recommendations, based on the observations made in this paper.

The right graphs. Though the overview in Section 2 is primarily intended (as the section title says) to provide a roadmap through the possibly bewildering set of graph variations, one can also use it for another purpose, namely to select the "right" graph formalism, meaning the kind of graphs with the most advantages.

For us, the winner is hypergraphs (Section 2.3), based on the following observations:

+ Hypergraphs make nodification (see Section 2.2) unnecessary in many cases. Thus, graphs can remain simpler (in terms of node and edge count), which directly helps visual and computational scalability.
+ Using hypergraphs, node deletion can be avoided; instead, nodes can be "garbage collected" whenever they have become isolated. Thus, in terms of the algebraic approach, all pushout complements exist: in other words, the only condition for rule applicability is the existence of a matching.
+ Hyperedges can combine data-valued and node-values tentacles, and thus provide an elegant extension to attributed graphs.
− The variable arity of hyperedges is an added complication, both for the development of theory and for tool support. (We believe that this is outweighed by the fact that nodification can often be avoided.)
− The usual graphical convention for hypergraphs, where edges are depicted as node-like boxes and their tentacles as arrows from those to the "real" nodes, is cumbersome and unattractive. (We believe that alternative, more intuitive notations can be devised.)

Terminology. In Section 2.7, we have made some recommendations for alternatives to the scary category theoretical terminology of algebraic graph transformation: *conservative transformation* for the double pushout approach, and *radical transformation* for the single pushout approach. Even if these suggestions are not themselves taken up, we hope to inspire a discussion on this point.

Theoretical maturity. In Section 3.5 we have identified several open issues, the solution of which we believe to be of prime importance for the success of graph transformation in the area of behavioural specification and analysis. To reiterate, they are *abstraction*,

compositionality and *symbolic representation*. No doubt there are more such issues; however, we propose to address these in the coming years.

Standardisation. Standardisation efforts in the past have led to an XML-based standard for graph exchange, GXL (see [37]) and a related standard for the exchange of graph transformation systems, GTXL (see [77,46]). However, as noted in Section 3.6, these standards have not made as much impact as one would wish.

A good standard can bring many benefits:

- Each individual tool or application can use the standard for storing its own data structures; there is no need for repeating the difficult process of defining a storage format. (Note that this is only a benefit if the standard does not impose overhead that is unnecessary from the perspective of the tool or application at hand.)
- In the same vein, standards can be used as a basis for the exchange of data between different tools; in other words, for the interoperability of tools. This, in fact, is typically the main driving force behind the definition of standards. However, interoperability is not automatically achieved even between tools that use a given standard for storage, as they may fail to support the *entire* standard. Also, the task of making a tool compliant to a standard is one that will be undertaken only if the concrete benefits are clear; in practice it turns out that this is only rarely the case.
- If standards are human-readable, rather than just digestible to computers, then they may serve understandability. A person would need only to get versed in one notation, used across formalisms or tools, to easily read and interpret all models and specifications. Note, however, that the requirement of human-readability rules out XML-based formats.

We propose to revive the standardisation effort; if not on the grand scale of GXL and GTXL (which essentially have the ambition of encapsulating arbitrary graphs and graph transformation systems), then possibly among a small set of tools whose creators share the willingness to invest the necessary effort.

Cooperation. The last point we wish to make does not follow from any of the discussion in this paper, but is rather a call, or challenge. From approximately 1990 onwards, there has been almost continuous European level funding for graph transformation-based research, in the form of COMPUGRAPH [20], SEMAGRAPH [74], APPLIGRAPH [1], GETGRATS [33], SEGRAVIS [72] and SENSORIA [73]. Though some of these projects have a more theoretical focus and others a more practical, there seems little doubt that they have together fostered strong ties within the community and benefited both theory and practice of graph transformation.

However, the last of these projects (SENSORIA) is due to end in 2010. It is therefore high time to take a new initiative. As should be clear from this paper, we strongly feel that only by combining the best of theory and practice we can progress further. At least in the area of behavioural specification and analysis, graph transformation will only be able to prove its added value if we do not hesitate to tackle the open theoretical issues and at the same time continue working on tool support and standardisation.

Who will take up this gauntlet?

References

1. APPLIGRAPH: Applications of graph transformation (1994), http://www.informatik.uni-bremen.de/theorie/appligraph/
2. Baldan, P., Corradini, A., König, B.: A static analysis technique for graph transformation systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
3. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Inf. Comput. 206(7), 869–907 (2008)
4. Baldan, P., Ehrig, H., König, B.: Composition and decomposition of dpo transformations with borrowed context. In: [13], pp. 153–167
5. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 214–228. Springer, Heidelberg (2004)
6. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: [24], pp. 431–433
7. Bauer, J., Boneva, I., Kurbán, M.E., Rensink, A.: A modal-logic based graph abstraction. In: [25], pp. 321–335
8. Bauer, J., Wilhelm, R.: Static analysis of dynamic communication systems by partner abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
9. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. In: [25], pp. 396–410
10. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Logic in Computer Science (LICS), pp. 428–439. IEEE Computer Society, Los Alamitos (1990)
11. Ciraci, S.: Graph Based Verification of Software Evolution Requirements. PhD thesis, University of Twente (2009)
12. Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: [24], pp. 383–398
13. Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.): ICGT 2006. LNCS, vol. 4178. Springer, Heidelberg (2006)
14. Corradini, A., Montanari, U. (eds.): Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA). Electronic Notes in Theoretical Computer Science, vol. 2 (1995)
15. Crouzen, P., van de Pol, J.C., Rensink, A.: Applying formal methods to gossiping networks with mCRL and GROOVE. ACM SIGMETRICS Performance Evaluation Review 36(3), 7–16 (2008)
16. Dodds, M., Plump, D.: Graph transformation in constant time. In: [13], pp. 367–382
17. Dörr, H.: Efficient Graph Rewriting and Its Implementation. LNCS, vol. 922. Springer, Heidelberg (1995)
18. Dotti, F.L., Ribeiro, L., dos Santos, O.M., Pasini, F.: Verifying object-based graph grammars. Software and System Modeling 5(3), 289–311 (2006)
19. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. J. Comput. Syst. Sci. 64(2), 249–283 (2002)
20. Ehrig, H.: Introduction to COMPUGRAPH. In: [14], pp. 89–100
21. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundam. Inform. 74(1), 31–61 (2006)

22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Heidelberg (2006)
23. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages and Tools, vol. II. World Scientific, Singapore (1999)
24. Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.): ICGT 2004. LNCS, vol. 3256. Springer, Heidelberg (2004)
25. Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.): ICGT 2008. LNCS, vol. 5214. Springer, Heidelberg (2008)
26. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G. (eds.): Handbook of Graph Grammars and Computing by Graph Transformation. Concurrency, Parallelism, and Distribution, vol. III. World Scientific, Singapore (1999)
27. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: 14th Annual Symposium on Switching and Automata Theory, pp. 167–180. IEEE, Los Alamitos (1973)
28. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML activities using dynamic meta modeling. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 76–90. Springer, Heidelberg (2007)
29. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: language and environment. In: [23], http://tfs.cs.tu-berlin.de/agg/
30. The FUJABA toolsuite (2006), http://www.fujaba.de
31. GBT — graph backwards tool, http://www.it.uu.se/research/group/mobility/adhoc/gbt
32. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: Grgen: A fast spo-based graph rewriting tool. In: [13], pp. 383–397
33. GETGRATS: General theory of graph transformation systems (1997), http://www.di.unipi.it/~andrea/GETGRATS/
34. Gorp, P.V., Keller, A., Janssens, D.: Transformation language integration based on profiles and higher order transformations. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 208–226. Springer, Heidelberg (2009)
35. Hausmann, J.H.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, University of Paderborn (2005)
36. Hausmann, J.H., Heckel, R., Sauer, S.: Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In: Human-Centric Computing Languages and Environments (HCC), pp. 80–87. IEEE Computer Society, Los Alamitos (2001)
37. Holt, R.C., Schürr, A., Sim, S.E., Winter, A.: GXL: A graph-based standard exchange format for reengineering. Sci. Comput. Program. 60(2), 149–170 (2006)
38. Holzmann, G.: The SPIN Model Checker — Primer and Reference Manual. Addison-Wesley, Reading (2004)
39. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Karsten Ehrig, H.G. (ed.) Graph Transformation and Visual Modeling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 6 (2007)
40. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based bounded model checking for software verification. Theor. Comput. Sci. 404(3), 256–274 (2008)
41. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
42. Kleppe, A.G., Rensink, A.: On a graph-based semantics for UML class and object diagrams. In: Ermel, C., Lara, J.D., Heckel, R. (eds.) Graph Transformation and Visual Modelling Techniques (GT-VMT). Electronic Communications of the EASST, vol. 10. EASST (2008)

43. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 197–211. Springer, Heidelberg (2006)
44. König, B., Kozioura, V.: Augur 2 — a new version of a tool for the analysis of graph transformation systems. In: Bruni, R., Varró, D. (eds.) Graph Transformation and Visual Modeling Techniques (GT-VMT 2006). Electronic Notes in Theoretical Computer Science, vol. 211, pp. 201–210 (2008)
45. Kuske, S., Gogolla, M., Kreowski, H.J., Ziemann, P.: Towards an integrated graph-based semantics for UML. Software and System Modeling 8(3), 403–422 (2009)
46. Lambers, L.: A new version of GTXL: An exchange format for graph transformation systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs). Electronic Notes in Theoretical Computer Science, vol. 127, pp. 51–63 (March 2005)
47. Lengyel, L., Levendovszky, T., Vajk, T., Charaf, H.: Realizing qvt with graph rewriting-based model transformation. In: Karsai, G., Taentzer, G. (eds.) Graph and Model Transformation (GraMoT). Electronic Communications of the EASST, vol. 4 (2006)
48. Levendovszky, T., Rensink, A., Van Gorp, P.: 5th International Workshop on Graph-Based Tools: The contest (grabats) (2009), http://is.ieis.tue.nl/staff/pvgorp/events/grabats2009
49. McMillan, K.L.: Methods for exploiting sat solvers in unbounded model checking. In: Formal Methods and Models for Co-Design (MEMOCODE), p. 135. IEEE Computer Society, Los Alamitos (2003)
50. Milner, R.: Communication and concurrency. Prentice-Hall, Inc., Englewood Cliffs (1989)
51. Milner, R.: Bigraphical reactive systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 16–35. Springer, Heidelberg (2001)
52. Milner, R.: Pure bigraphs: Structure and dynamics. Inf. Comput. 204(1), 60–122 (2006)
53. Nagl, M.: Set theoretic approaches to graph grammars. In: Ehrig, H., Nagl, M., Rosenfeld, A., Rozenberg, G. (eds.) Graph Grammars 1986. LNCS, vol. 291, pp. 41–54. Springer, Heidelberg (1987)
54. OMG: Meta object facility (MOF) core specification, v2.0. Document formal/06-01-01, Object Management Group (2006), http://www.omg.org/cgi-bin/doc?formal/06-01-01
55. OMG: Unified modeling language, superstructure, v2.2. Document formal/09-02-02, Object Management Group (2009), http://www.omg.org/cgi-bin/doc?formal/09-02-02
56. Palacz, W.: Algebraic hierarchical graph transformation. J. Comput. Syst. Sci. 68(3), 497–520 (2004)
57. Pennemann, K.H.: Development of Correct Graph Transformation Systems. PhD thesis, University of Oldenburg, Oldenburg (2009), http://oops.uni-oldenburg.de/volltexte/2009/948/
58. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
59. Rensink, A.: Isomorphism checking in GROOVE. In: Zündorf, A., Varró, D. (eds.) Graph-Based Tools (GraBaTs). Electronic Communications of the EASST, European Association of Software Science and Technology, vol. 1 (September 2007)
60. Rensink, A.: Compositionality in graph transformation. In: Abramsky, S., et al. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 309–320. Springer, Heidelberg (2010)
61. Rensink, A., Distefano, D.S.: Abstract graph transformation. In: Mukhopadhyay, S., Roychoudhury, A., Yang, Z. (eds.) Software Verification and Validation, Manchester. Electronic Notes in Theoretical Computer Science, vol. 157, pp. 39–59 (May 2006)
62. Rensink, A., Taentzer, G.: Agtive 2007 graph transformation tool contest. In: [71], pp. 487–492, http://www.informatik.uni-marburg.de/~swt/agtive-contest

63. Rensink, A., Van Gorp, P.: Graph transformation tool contest 2008. Software Tools for Technology Transfer (2009) Special section; in preparation, http://fots.ua.ac.be/events/grabats2008.
64. Rensink, A., Zambon, E.: A type graph model for java programs. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 237–242. Springer, Heidelberg (2009)
65. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
66. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
67. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
68. Schmidt, Á., Varró, D.: Checkvml: A tool for model checking visual modeling languages. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 92–95. Springer, Heidelberg (2003)
69. Schürr, A.: Programmed graph replacement systems. In: [65], pp. 479–546
70. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: [25], pp. 411–425
71. Schürr, A., Nagl, M., Zündorf, A. (eds.): Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 5088. Springer, Heidelberg (2008)
72. SEGRAVIS: Syntactic and semantics integration of visual modelling techniques (2002), http://www.segravis.org/
73. SENSORIA: Software engineering for service-oriented overlay computers (2005), http://www.sensoria-ist.eu
74. Sleep, M.R.: SEMAGRAPH: The theory and practice of term graph rewriting. In: [14], pp. 268–276
75. Taentzer, G.: Parallel high-level replacement systems. Theor. Comput. Sci. 186(1-2), 43–81 (1997)
76. Taentzer, G.: Distributed graphs and graph transformation. Applied Categorical Structures 7(4) (1999)
77. Taentzer, G.: Towards common exchange formats for graphs and graph transformation systems. In: Uniform Approaches to Graphical Process Specification Techniques (UNIGRA). Electronic Notes in Theoretical Computer Science, vol. 44, pp. 28–40 (2001)
78. Taentzer, G., Biermann, E., Bisztray, D., Bohnet, B., Boneva, I., Boronat, A., Geiger, L., Geiß, R., Horvath, Á., Kniemeyer, O., Mens, T., Ness, B., Plump, D., Vajk, T.: Generation of Sierpinski triangles: A case study for graph transformation tools. In: [71], pp. 514–539
79. Taentzer, G., Fischer, I., Koch, M., Volle, V.: Distributed graph transformation with application to visual design of distributed systems. In: [26], ch. 6, pp. 269–340
80. Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 64–79. Springer, Heidelberg (2005)