Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

On Model Checking Generic Topologies

Marius Bozga and Radu Iosif

Report n^o TR-2004-7

April 23, 2004

On Model Checking Generic Topologies

Marius Bozga and Radu Iosif

April 23, 2004

Abstract

This paper develops a method for verifying properties related to the topologies of finite graph structures. We introduce a generic framework for modeling graphs, based on Kleene algebras, and define an expressive logic which uses equivalences of regular expressions, allowing to state structural properties, such as: subgraph isomorphism, existence of cliques and strongly connected components, planarity, etc. We study the satisfiability and model checking problems for this logic and find that the former is undecidable and the latter is NP-hard. Second, we give a model checking algorithm and propose some heuristics for reducing its complexity. Finally, we describe possible uses of our technique to solve specific verification problems, such as shape analysis or checking consistency of UML models.

Keywords: Graph topologies, Kleene algebra, Model checking

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { On Model Checking Generic Topologies},
authors = { Marius Bozga and Radu Iosif},
institution = { Verimag Technical Report },
number = {TR-2004-7},
year = { },
note = { }
}

1 Introduction

The model checking problem for a logic L gives a formula φ (written in L), and a structure $\mathcal{S}$ and asks whether the structure satisfies the formula: $\mathcal{S} \models \varphi$? Model checking problems are usually decidable on finite and regular models, and a large number of algorithms has been published over the past twenty years. Typical uses of these algorithms are in verification of reactive systems, see e.g. [6].

In the recent years, work has been done to develop logics that are useful in reasoning about topologies of systems. Such logics focus on graphs, involving properties ranging from simple ones e.g., reachability, circularity to more difficult ones e.g., isomorphism, planarity. Uses of graph logics are in static analysis of programs (i.e., shape analysis [14]), verification of database-like structures, websites, high-level software specifications, etc. Other uses could possibly be found in computer graphics, involving e.g., visualizations of graphs.

In this paper we use the storeless graph model introduced first by Jonkers [12] and Deutsch [7] to describe the configuration of heaps occurring during the execution of programs. In this model each vertex is identified with the set of incoming traces, and, if the graph is finite, this set can be shown to be regular. On top of this model we define a logic called Disunification in Kleene Algebras (DKA), which is essentially first-order logic where atomic propositions are equivalences of regular expression with variables. Variables are themselves interpreted as regular languages denoting graph vertices in the storeless model. Briefly, the contributions of this paper are:

- A generalization of the classical storeless model using Kleene algebras. This allows us to model larger classes of graphs than the deterministic graphs considered in [12], [7] and [3], for instance non-deterministic and modal graphs.

- A study of the complexity of satisfiability and model checking problems for DKA. We found that, while the logic is undecidable, the model checking problem is NP-hard, but not NP-complete.

- A study of the expressive power of the DKA logic and a comparison with other two logics for graphs, namely the propositional μ-calculus [9] and the Monadic Second Order Logic (MSOL) [5].

- An algorithm for model checking. Since the problem is intractable in general, we propose a technique to improve performance, and also a symbolic representation of sets, using a new class of decision diagrams operating on Kleene algebras.

Finally, concrete applications of the framework are discussed, such as shape analysis, checking integrity of databases and high-level system specifications, etc.

1.1 Related Work

The most exhaustive work on graphs and graph logic is, to the best of our knowledge, the work of Courcelle (see [5] for a survey). His study of graph rewriting and Monadic Second Order Logic of graphs has found unfortunately, little applications in software verification. In this paper we sketch a comparison between the logics DKA and MSOL (with quantification over sets of nodes only) and found that both can express most of the properties we retain interesting to describe graph topology. However, the two logics are formally incomparable. One difference is that in DKA we can express (a limited version) of subgraph isomorphism, whereas in MSOL this is provably impossible to express. On the other hand, graph planarity is expressible in MSOL, whereas in DKA we could not find a proper specification. A more formal comparison, that considers also MSOL with edge set quantification, is planned as future work.

Another logic of interest is L_r [2], which allows the use of regular expressions to express reachability constraints in the heap. This logic is also shown to be decidable. However, the lack of quantification yields a lesser expressive power than the one of DKA. It is also not hard to imagine a formal embedding from L_r into DKA.

Recent work by de Alfaro [1] describes the use of (a restricted version of) the propositional μ-calculus to verify the structural properties of websites. However, as discussed in Section 3, the expressive power of the μ-calculus is well below the one of DKA.

Moreover, both L_r and μ-calculus are subsets of MSOL, more precisely of the calculus of n-successor trees (SnS), whereas DKA is obviously not. For instance, the property of confluence (i.e. the existence of a node with two predecessors) can be easily expressed in DKA.

2 The Logic of Disunification in Kleene Algebras

In this section we describe the logic used in this paper to express topological properties of finite structures. We call this logic Disunification in Kleene Algebras, or short DKA, in analogy with the disunification problems studied in [4]. To begin with, we introduce a semantic model that describes graph structures in terms of sets of regular languages of traces. Despite the apparent lack of intuition, this model has several advantages over the classical ones (describing vertices and edges explicitly) such as identifying isomorphic structures. The trace model is used next to define the semantics of the DKA logic. Finally we study the complexity of the satisfiability and model checking problems for DKA.

2.1 The Storeless Graph Model

Consider a *finitely rooted* directed graph, that is, a graph in which the number of vertices with no incoming edges is non-zero and finite. We associate these vertices a special meaning, that of *entry points* and denote them by $e_1, \ldots, e_k$. Next, assume that the edges of the graph are labeled with symbols from a finite alphabet Σ such that, for each vertex in the graph the outgoing edges are labeled with different symbols. In other words, we say that the graph is *deterministic*. Without loss of generality, we add a new node ι to the graph and connect it to the entry points $e_1, \ldots, e_k$ by means of some new edges labeled $r_1, \ldots, r_k$. Let Π be the set of root labels i.e., $\Pi = \{r_1, \ldots, r_k\}$. Formally, the graph is a tuple $G = \langle V, E, \iota, \Pi, \Sigma \rangle$, where V denotes the original set of vertices ($\iota \notin V$), $E \subseteq (\{\iota\} \times \Pi \times V) \cup (V \times \Sigma \times V)$ is the edge relation, ι is the root node, Π is the root alphabet and Σ the edge alphabet.

Now each vertex $v \in V$ can be associated a deterministic finite state automaton $\mathcal{A}_v = \langle V \cup \{\iota\}, E, \iota, v\rangle$ where V and E are the set of states and transition relation respectively, ι the initial state and v the final state. The language $\mathcal{L}_v$ recognized by this automaton uniquely identifies v, and therefore the graph G can be uniquely represented (modulo isomorphism) by the set $\{\mathcal{L}_v \mid v \in V\}$. For historical reasons [12], [7] call this the *storeless* representation of G. The following definition captures the concept, where $\cdot$ is the concatenation of words extended to languages pointwise:

Definition 1 (Deterministic Graph) *Given a finite root alphabet* Π *and an edge alphabet* Σ*, such that* $\Pi \cap \Sigma = \emptyset$*, a* graph $\mathcal{G} \subseteq \mathcal{P}(\Pi \cdot \Sigma^*)$ *is either the empty set or a finite set* $\{\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n\}$ *satisfying the following conditions, for all* $1 \leq i, j \leq n$*:*

(D1) non-emptiness: $\mathcal{L}_i \neq \emptyset$*, and (D'1) finite rooting:* $\forall r \in \Pi \,.\, \{r\} \in \mathcal{G}$

(D2) prefix closure and right regularity:

$$\forall x \in \mathcal{L}_i \; [\forall y, z \in \Sigma^+ [x = y \cdot z \Rightarrow \exists\, 1 \leq k \leq n \; [y \in \mathcal{L}_k \textit{ and } \mathcal{L}_k \cdot z \subseteq \mathcal{L}_i]]]$$

(D3) determinism: $i \neq j \Rightarrow \mathcal{L}_i \cap \mathcal{L}_j = \emptyset$*,*

A proof of equivalence of Definition 1 with the classical node-edge model can be found in [3]. Other approaches in the literature e.g., [12], [7], consider the languages in the graph as equivalence classes of a prefix-closed right-regular relation (*alias*), in the sense of the Myhill-Nerode Theorem.

2.2 Syntax and Semantics of DKA

A *semiring* $(K, \sqcup, \odot, \mathbf{0}, \mathbf{1})$ consists of a commutative monoid $(K, \sqcup, \mathbf{0})$ and a monoid $(K, \odot, \mathbf{1})$, such that $\odot$ is left-right distributive over $\sqcup$ and $x \odot \mathbf{0} = \mathbf{0} \odot x = \mathbf{0}$. If, in addition, $\odot$ is a commutative operation, we say that the semiring is *commutative*. A semiring is said to be *complete* if $\bigsqcup_{i \in I} x_i \in K$ for any (possibly infinite) indexing set I and any $x_i \in K$. The distributivity of $\odot$ extends naturally to infinite $\sqcup$ sums. A *Kleene algebra* is a complete semiring. We define $x^\star = \bigsqcup_{i \geq 0} x^i$, where $x^0 = \mathbf{1}$, $x^1 = x$ and $x^{i+1} = x^i \odot x$ for some $x \in K$. In other words, a Kleene algebra is an algebra of regular expressions over an arbitrary semiring K. If the multiplicative monoid $(K, \odot, \mathbf{1})$ is free with base $B_K \subseteq K$ we say that the algebra K is *free* with base B_K. In other words, all elements of K can be generated only by applications of the operations $\sqcup$, $\odot$ and $\star$ to the elements of B_K. This level of abstraction mimics closely the well-known semantics of regular languages, and is adopted for further extensions of the framework, beyond the classical Σ^* algebra. Such extensions will be discussed in the end of this section.

The following operations are defined on top of the basic $\sqcup$, $\odot$ and $\star$. Let $x \sqsubseteq y \iff x \sqcup y = y$, $x \sqcap y = \bigsqcup\{z \in K \mid z \sqsubseteq x \text{ and } z \sqsubseteq y\}$, $\overline{x} = \bigsqcup\{z \in K \mid z \sqcap x = \mathbf{0}\}$ and $x^{-1}y = \bigsqcup\{z \in K \mid x \odot z \sqsubseteq y\}$. The last operation on K is the abstract counterpart of the *left derivate* from language theory. Notice how Definition 1 can be rephrased in terms of arbitrary Kleene algebras, as in the following:

Definition 2 (Generalized Graph) *Given a free Kleene algebra* $(K, \sqcup, \odot, \star, \sqsubseteq, \mathbf{0}_K, \mathbf{1}_K)$ *with disjoint subsets of generators* $\Pi, \Sigma \subseteq B_K$ *and a commutative semiring* $(L, \oplus, \otimes, \leq, \mathbf{0}_L, \mathbf{1}_L)$ *with* $\mathbf{0}_L \neq \mathbf{1}_L$*, a* generalized graph *over* K*,* L *is a mapping* $\mathcal{G} : K \to L$ *satisfying:*

(G1) non-emptiness: $\forall x \in K \,.\, \mathcal{G}(x) \neq \mathbf{0}_L \Rightarrow x \neq \mathbf{0}_K$

(G'1) finite rooting: $\forall r \in \Pi \,.\, \mathcal{G}(r) \neq \mathbf{0}_L$

(G2) prefix closure and right regularity:

$$\forall x \in K \,.\, \mathcal{G}(x) \neq \mathbf{0}_L \Rightarrow \forall y, z \in K \,.\, x = y \odot z \Rightarrow \\ \exists\, t \in K \,.\, \mathcal{G}(t) \neq \mathbf{0}_L \text{ and } y \sqsubseteq t \text{ and } t \odot z \sqsubseteq x$$

The generalized graph is said to be deterministic if, moreover:

(G3) $\forall x, y \in K \,.\, \mathcal{G}(x) \neq \mathbf{0}_L$ *and* $\mathcal{G}(y) \neq \mathbf{0}_L$*and* $x \neq y \Rightarrow x \sqcap y = \mathbf{0}_k$*.*

A generalized graph is said to be *finite* if $\mathcal{G}(x) = \mathbf{0}_L$ in all but a finite number of points $x \in K$. The set of all graphs over a free Kleene algebra K and a semiring L is denoted by $\Gamma[K, L]$. Unless otherwise stated, L is the boolean semiring $(\mathbb{B}, \vee, \wedge, \bot, \top)$, which turns $\mathcal{G}$ into a set of elements of K. In this case we forget L and denote the set of all graphs over K by $\Gamma[K]$.

Figure 1 gives the syntax and the semantics of the logic of Disunification in Kleene Algebras, or short DKA. The terms ρ are regular expressions containing variables and the formulas are those of first-order logic with atomic propositions of the form $\rho_1 = \rho_2$. The function $FV(\rho)$ returns the set of all free variables occurring inside ρ. The semantics of a DKA formula is given with respect to a free Kleene algebra K, a semiring L, and a graph $\mathcal{G} \in \Gamma[K, L]$. The semantics of regular expressions is given by interpreting $\cdot$ as $\odot$, $*$ as $\star$, $\cup$ as $\sqcup$, $\cap$ as $\sqcap$ and so on, in the given K-algebra. We use a special symbol $\mathcal{A}$ to denote the base (alphabet) of the free K-algebra. As usual, $\emptyset$ represents the empty set. Equality of expressions amounts to checking equivalence in the K-algebra. Note that we interpret quantified variables only over the languages representing the entities in the graph, according to Definition 2. This restriction is intended to relate graphs and DKA formulas and allow to use the logic for expressing properties of graphs.

We adopt the following notation as a shorthand for the remainder of this paper: $\rho_1 \not\parallel \rho_2 \triangleq \rho_1 \cap \rho_2 \neq \emptyset$ and $\rho_1 \parallel \rho_2 \triangleq \rho_1 \cap \rho_2 = \emptyset$. In particular, when writing $X \not\parallel \rho$, the meaning is that the node represented by X is reachable via some path from ρ.

This general representation has been found suitable for the study of the decision problems related to DKA, as well as to introduce a generic model checking algorithm. It is to be noticed that all results in the rest of the paper are given for the generalized heap model. Below we present two possible instantiations of the generalized framework.

2.2.1 Non-deterministic Graphs

Thanks to the abstract definition of a graph, using Kleene algebras, we can accommodate non-deterministic graphs as follows. Let K be the algebra of languages over the alphabet A with

$$
\begin{array}{rcl}
\rho & ::= & k \in K \mid X \in Var \mid \mathcal{A} \mid \emptyset \mid \rho_1 \cdot \rho_2 \mid \rho^* \mid \rho_1 \cup \rho_2 \mid \rho_1 \cap \rho_2 \mid \bar{\rho} \mid \rho_1^{-1}\rho_2 \\
\varphi & ::= & \rho_1 = \rho_2 \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \exists X\ \varphi
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G} \in \Gamma[K, L] & & \nu : Var \to \mathcal{G} \\
[\![\mathcal{A}]\!] & = & B_K \\
[\![\rho]\!]_\nu & = & [\![\rho[\nu(F\vec{V}(\rho))/F\vec{V}(\rho)]]\!] \\
[\![\rho_1 = \rho_2]\!]_\nu = 1 & \iff & [\![\rho_1]\!]_\nu = [\![\rho_2]\!]_\nu \\
[\![\exists X\ \varphi]\!]_{\mathcal{G},\nu} = 1 & \iff & \exists\ \rho \in K\ .\ \mathcal{G}(\rho) \neq \mathbf{0}_L \text{ and } [\![\varphi]\!]_{\mathcal{G},[X \to \rho]\nu} = 1
\end{array}
$$

Figure 1: Disunification in Kleene Algebras

multiplicities from $\mathbb{N}$, $A^{\mathbb{N}} = (A^* \mapsto \mathbb{N}, \boxplus, \boxtimes, \overline{\emptyset}, \overline{\epsilon})$. A language $L \in A^{\mathbb{N}}$ represents a multiset of sequences, where for $\sigma \in A^*$, $L(\sigma)$ gives the number of occurrences of sequence σ in the language. The operations are as follows: for any $F, G : A^* \to \mathbb{N}$, $s \in A^*$, $(F \boxplus G)(s) = F(s) + G(s)$, $(F \boxtimes G)(s) = \Sigma_{s=ut} F(u)G(t)$, $\overline{\emptyset}$ is the function that maps everything into 0, and $\overline{\epsilon}$ is the function that maps everything except ϵ into 0 and ϵ into 1, where ϵ is the empty word. Let L be the semiring of natural numbers with classical addition and multiplication.

When we represent non-deterministic graphs as sets of languages, the same language may denote two or more vertices. Take for example the graph in Figure 2 (a). This graph can be described by the function mapping $\{a\}$ and $\{ab\}$ into 2, and everything else into 0. An ad-hoc notation for this graph is $\{\{a\}^2, \{ab\}^2\}$, where the superscript marks the multiplicity of an element. However, describing the graph as a multiset of regular languages is not enough, as one can see in Figure 2 (b). Here we must take into account the multiple occurrence of word ab in the two sets on the bottom. Hence the representation of this graph is (using our ad-hoc notation) $\{\{a\}^2, \{(ab)^2\}^2\}$. It is important in this case that the word ab appears twice in the languages that represent the bottom nodes in Figure 2 (b).

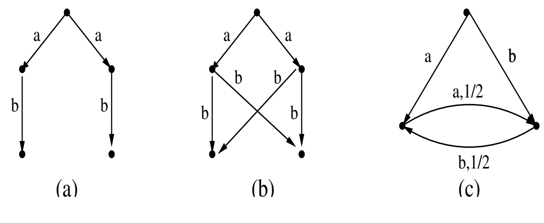


Figure 2: Non-deterministic and Modal Graphs

2.2.2 Modal Graphs

Modal graphs are graphs in which edges *do* exist or *may* exist. Modal graphs are a useful tool to express lack of precise information in defining abstractions. Let $\mathbb{K}$ the algebra of Kleene's

3-valued logic [11] i.e., $\mathbb{K} = (\{0, \frac{1}{2}, 1\}, \vee_3, \wedge_3, 0, 1)$, where $0 \vee_3 \frac{1}{2} = \frac{1}{2}$, $1 \wedge_3 \frac{1}{2} = \frac{1}{2}$, $0 \wedge_3 \frac{1}{2} = 0$, $1 \vee_3 \frac{1}{2} = 1$ and so on. Let $A^{\mathbb{K}}$ be the algebra of languages over the alphabet A with multiplicities from $\mathbb{K}$, defined in a similar way as $A^{\mathbb{N}}$. A deterministic modal graph over the alphabet A is an element of $\Gamma[A^{\mathbb{K}}, \mathbb{B}]$ while a non-deterministic modal graph is an element of $\Gamma[A^{\mathbb{K}}, \mathbb{N}]$.

For instance, the deterministic modal graph in Figure 2 (c) is represented as $\{a(a^{\frac{1}{2}}b^{\frac{1}{2}})^*, b(b^{\frac{1}{2}}a^{\frac{1}{2}})^*\}$. In our notation, we have used superscripts directly on the symbols, to mimic the multiplicity of the edges. The semantics of a regular expression with symbol multiplicities is a function returning for each word in the language its multiplicity, defined as the product between the multiplicities of its symbols. In our case, the word $aabab$ occurs in $a(a^{\frac{1}{2}}b^{\frac{1}{2}})^*$ with multiplicity $1 \wedge_3 \frac{1}{2} \wedge_3 \frac{1}{2} \wedge_3 \frac{1}{2} \wedge_3 \frac{1}{2} = \frac{1}{2}$ times.

2.3 The Complexity of DKA

In this section we investigate the satisfiability and the model checking problems for the DKA logic defined over an arbitrary free Kleene algebra K. The special case in which K is a commutative algebra needs further attention, as for instance the general undecidability result, for the satisfiability problem, does not apply here [10].

We show next that satisfiability of DKA is undecidable in general and the model checking problem (given a graph $\mathcal{G}$, is it a model for a given formula φ?) is NP-hard. All proofs use reductions from two well-known tiling problems, defined in the following.

Definition 3 (Tiling Problems) *Let C be a set of colors and $T = \{t_1, \ldots, t_m\} \subseteq C \times C \times C \times C$ be a set of tiles. A tile $t = (t^l, t^r, t^u, t^d)$ is a square and each edge is marked with a color in the following order: left, right, up, down. A tiling of (a region of) the grid $\mathbb{N} \times \mathbb{N}$ is a (partial) assignment of tiles to points such that any two adjacent tiles have matching edge colors.*

We consider the following tiling problems [8], given a set of tiles T over a set of colors C:

(T1) Can T tile $\mathbb{N} \times \mathbb{N}$? This problem is undecidable.

(T2) Given $n \in \mathbb{N}$, can T tile $n \times n$? This problem is NP-complete.

The following results use reductions from (T1) and (T2), respectively. Proofs are deferred to the Appendix.

Lemma 1 *The satisfiability problem for DKA for non-commutative Kleene algebras is undecidable.*

The reduction from the proof of Lemma 1 does not hold in case of commutative Kleene algebras. Moreover, the satisfiability problem in commutative Kleene algebras can be effectively solved e.g., using the method from [10].

Lemma 2 *The model checking problem for DKA is NP-hard.*

It seems that the complexity assessment from Lemma 2 cannot be easily improved. For instance, the model checking problem doesn't appear to be NP-complete, for the following reason. Assume that we have an assignment of the existentially quantified variables in a DKA formula to regular languages. The problem is to test that the assignment satisfies the formula, which amounts to checking equivalence of regular expressions. However, this problem requires exponential time in general, see e.g. [16]. In what concerns space complexity, we have not been able to provide a reduction from the horizontally-bounded tiling problem: given n there exists a tiling of $n \times m$ by T for some $m \in \mathbb{N}$, such that $(0, 0)$ maps to c_0 and $(0, m)$ maps to c_1 ? Such a reduction would have provided us with a better knowledge of space complexity of the problem as well (PSPACE-hardness).

3 Expressing Graph Properties

This section is concerned with the expressiveness of the DKA logic. We discuss possible uses of the formalism to describe basic properties of graphs. Relations with other formalisms for graphs, such as Monadic Second Order Logic (MSOL) [5], or propositional μ-calculus [9] are discussed. Basic knowledge of these formalisms is assumed in the following.

3.0.1 Path Properties

The following properties describe the existence of paths in the structure. We consider the predicate $reach(X, Y) = Y \nparallel X\mathcal{A}^+$ stating that node Y is reachable from node X by some non-empty path. A node Y is said to be *next* to a node X if the following predicate holds: $next(X, Y) = Y \nparallel X\mathcal{A} \wedge \forall Y' . Y' \nparallel X\mathcal{A} \rightarrow Y = Y'$. In other words, if Y is the only neighbor of X. A path from X to Y is *linear* if there is no branching i.e., if all the nodes on the path have only one successor:

$$\begin{aligned} linear(X, Y) \quad = \quad & reach(X, Y) \wedge \forall Z . \neg Z = Y \wedge \big(X = Z \vee reach(X, Z)\big) \wedge \\ & reach(Z, Y) \rightarrow \exists Z' . \neg Z' = Z \wedge next(Z, Z') \end{aligned}$$

The existence of a cycle containing both X and Y is given by the following predicate: $cycle(X, Y) = reach(X, Y) \wedge reach(Y, X)$, and if we are only interested by stating the existence of a cycle containing X, the following can be used: $\exists Y . cycle(X, Y)$.

Path properties are typical uses of the μ-calculus, where the a-edge relation between two nodes satisfying the atomic propositions φ and ψ is given by $\varphi \wedge \langle a \rangle \psi$. Reachability of a φ node from the current node is stated using the least fixed point operator e.g., $\mu X. \varphi \wedge \langle a \rangle X$. However, unlike DKA, in μ-calculus there is no way of referring explicitly to (nor quantifying over) graph nodes. This is one reason why our logic is strictly more expressive than the propositional μ-calculus, as it will be shown in the remainder.

Transitive closure of a relation is a MSOL-expressible property [5]:

$$R^*(x_1, x_2) : \forall X[(\forall y, z(y \in X \wedge R(y, z) \rightarrow z \in X) \wedge \forall y(R(x_1, y) \rightarrow y \in X)) \rightarrow x_2 \in X]$$

Taking the transitive closure of the edge relation in a graph one can define $reach(X, Y)$ in MSOL, and therefore mimic closely most of the properties given in the following, with the exception of *isomorphic components*.

3.0.2 Recursive Data Structures

The DKA logic can describe the shapes of most typical recursive data structures used in programming languages with dynamic memory allocation: lists, trees, dags, etc. In reality, a heap-allocated data structure is deterministic, since a pointer variable can only have one address at the time. We can enforce this by adding the following constraint to our formulas:

$$\forall X, Y, Z \;.\; Y \nparallel X\mathcal{A} \wedge Z \nparallel X\mathcal{A} \rightarrow Y = Z \tag{1}$$

For instance, non-cyclic simply-linked lists pointed to by the $head$ variable and using the $next$ field as forward selector, are being described by the following predicate:

$$nclist(head) \;=\; \forall X \;.\; X \nparallel head \rightarrow \exists Y \;.\; Y \nparallel X next^* \wedge linear(X, Y) \wedge \neg cycle(Y, Y)$$

If, moreover, the list is not shared by other structures, we may replace the $Y \nparallel X next^*$ subformula by $Y \subseteq head \cdot next^*$. This also ensures the lack of backward pointers in the list, due to the deterministic constraint given in the previous.

Doubly-linked lists pointed to by the *head* variable and using the *next* and *prev* field pointers as forward and backward selectors, respectively, can be captured by the following:

$$\begin{aligned} dlist(head, next, prev) \;=\; & \forall X, Y \exists Z \;.\; (X \nparallel head \Rightarrow \neg Y \nparallel X prev) \wedge \\ & (Z \nparallel X next \Rightarrow X \neq Z \wedge X \nparallel Z prev) \end{aligned}$$

Some data structures, such as trees, require the absence of sharing. A sharing predicate expressing that X and Y belong to two structures that share some node can be given as: $share(X, Y) = \exists Z \;.\; reach(X, Z) \wedge reach(Y, Z)$. A tree structure pointed to by a variable *root* is described by the following formula:

$$tree(root) = \forall X . X \nparallel root \rightarrow \forall Y, Z \;.\; (reach(X, Y) \wedge reach(X, Z)) \rightarrow \neg share(Y, Z))$$

A dag structure in which every node is reachable from a *root* variable is given by the following formula. Note the absence of cycles between any two nodes in the structure.

$$dag(root) = \exists X \;.\; X \nparallel root \rightarrow \forall Y, Z \;.\; reach(X, Y) \wedge reach(X, Z) \rightarrow \neg cycle(Y, Z)$$

Since the formulas describing various recursive data structures use path predicates in conjunction with quantification over nodes, they go beyond the expressive power of μ-calculus. However using MSOL, one can express these properties in a very similar way it is done here.

3.0.3 Isomorphic Components

Defining isomorphic graphs or isomorphic components of the same graph is one of the hardest encoding problems concerning graph logics. As pointed out in [5], this can be done in Second Order Logic (SOL) but it is provably beyond the scope of MSOL. Our encoding is done only for deterministic graphs, i.e. assuming (1), and uses the following idea. We say that two nodes are *trace equivalent* if the sets of incoming traces are the same, for both nodes. Two graphs are trace equivalent if there exists a total relation between their nodes such that any two nodes that are related, are trace equivalent.

Fact 1 *Trace equivalent deterministic graphs are isomorphic.*

Trace equivalence is easily defined in our logic, since we can test equivalence of regular expressions that define sets of traces within the graph. The meaning of the following predicate is that for each node reachable from X there exists a node reachable from Y such that they are trace equivalent relatively to X and Y, respectively. The sets $X^{-1}X'$ and $Y^{-1}Y'$ denote the sets of paths from X to X' and from Y to Y', respectivelly. Note that X and Y need not be trace equivalent themselves.

$$treq(X,Y) = \forall X' \exists Y' \,.\, reach(X,X') \rightarrow X^{-1}X' = Y^{-1}Y'$$

We don't need to state $reach(Y,Y')$ explicitly, since X' is reachable from X, $X^{-1}X' = Y^{-1}Y' \neq \emptyset$, and therefore Y' is also reachable from Y. The existence of two isomorphic components within a graph is defined as follows:

$$\exists X, Y \,.\, X \neq Y \wedge treq(X,Y) \wedge treq(Y,X)$$

3.0.4 Cliques and Strongly Connected Components

A *clique* is a maximal complete subgraph. We can state the existence of a k-clique (a clique with exactly k nodes) as follows:

$$\begin{array}{lll} \exists X_1 \ldots X_k & . & \bigwedge_{1 \le i,j \le k} X_i \neq X_j \wedge \bigwedge_{1 \le i,j \le k} X_i \not\parallel X_j \mathcal{A} \wedge \\ \forall X & . & \bigwedge_{1 \le i \le k} X \neq X_i \rightarrow \bigvee_{1 \le i \le k} X_i \parallel X\mathcal{A} \vee X \parallel X_i \mathcal{A} \end{array}$$

A *strongly connected component* is a maximal subgraph in which there exists a path between any two nodes. The existence of a strongly connected component with exactly k nodes is expressed by a similar formula:

$$\begin{array}{lll} \exists X_1 \ldots X_k & . & \bigwedge_{1 \le i,j \le k} X_i \neq X_j \wedge \bigwedge_{1 \le i,j \le k} X_i \not\parallel X_j \mathcal{A}^* \wedge \\ \forall X & . & \bigwedge_{1 \le i \le k} X \neq X_i \rightarrow \bigvee_{1 \le i \le k} X_i \parallel X\mathcal{A}^* \vee X \parallel X_i \mathcal{A}^* \end{array}$$

We can also state the existence of a non-trivial clique (SCC) by letting $k = 2$ in the above formulas and omitting the last conjunct.

The existence of strongly connected components occurs implicitly in μ-calculus specifications, namely when liveness properties are expressed. In DKA we can be more explicit and pinpoint the components by referring to one of their nodes. According to [5], these properties can also be expressed in MSOL.

3.0.5 Planarity

A graph is *planar* if it can be drawn in one plane without edges crossing. Up to now we haven't found a good way to express planarity in DKA. On the other hand, planarity can be expressed in MSOL, as described in [5].

4 A Model Checking Algorithm

Let K be a Kleene algebra and L a semiring. In this section we develop a model checking algorithm supposed to work on any generalized graph $\mathcal{G} : K \to L$, even if we ignore the rooting condition G'_1 from Definition 2. The only requirement is that the equivalence of ground terms (expressions with no variables) on K be decidable.

Let Var be a set of free variables, as in Figure 1. For the purposes of this section, we redefine the semantics of a DKA term (a regular expression with variables) as follows:

$$\|\rho\| = \{(k, \nu) \mid [\![\rho]\!]_\nu = k\} \subseteq K \times (Var \mapsto K)$$

In other words, we associate the environment ν together with the value of ρ in this environment and take the set of all such pairs. A similar definition is given for logical formulas:

$$\|\varphi\| = \{\nu \mid [\![\varphi]\!]_\nu = 1\} \subseteq Var \mapsto K$$

The denotation of a formula is now the set of environments in which it holds. Notice that we can give such a denotation independently of the semantics of the formula on a given graph.

Let $\mathcal{G}$ be a finite graph. We denote by $dom(\mathcal{G})$ the set of points $k \in K$ such that $\mathcal{G}(k) \neq \mathbf{0}_L$. The model checking problem asks, for a given formula φ, whether the set $\mathcal{S} = \|\varphi\| \cap (FV(\varphi) \mapsto dom(\mathcal{G}))$ is not empty i.e., whether there exists an assignment of elements from $\mathcal{G}$ to the variables in Var such that the formula is satisfied. It is assumed in the following that all existentially quantified variables of φ are present in Var. Given that $dom(\mathcal{G})$ is finite, the set $\mathcal{S}$ can be effectively computed by the algorithm presented in Figure 4.

First we define an evaluation function for regular expressions, in a given graph:

$$regEval(\rho, \mathcal{G}) = \{(k, \nu) \mid \nu : FV(\rho) \to dom(\mathcal{G}), [\![\rho]\!]_\nu = k\}$$

It is obvious that $regEval(\rho, \mathcal{G}) = \|\rho\| \cap (K \times (FV(\rho) \mapsto dom(\mathcal{G})))$. The $regEval$ function can be implemented as in Figure 3. Notice the presence of the $\bowtie$ operator which denotes the disjoint union of mappings, and returns $\perp$ when the two mappings do not agree on one point.

For example $\{(x \mapsto a), (y \mapsto b)\} \bowtie \{(y \mapsto b), (z \mapsto c)\} = \{(x \mapsto a), (y \mapsto b)(z \mapsto c)\}$, but $\{(x \mapsto a), (y \mapsto b)\} \bowtie \{(y \mapsto d), (z \mapsto c)\} = \perp$. Notice how unmatching pairs of assignments are discarded.

$$
\begin{array}{l}
regEval(\rho, \mathcal{G}) \text{ is} \\
\text{case } \rho \text{ in} \\
\begin{array}{rcl}
\quad k & : & \{(k, [])\} \\
X & : & \{(k, [X \to k]) \mid k \in dom(\mathcal{G})\} \\
\rho_1 \cdot \rho_2 & : & \{(k_1 \odot k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G})_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \perp\} \\
\rho_1^* & : & \{(k_1^*, \nu_1) \mid (k_1, \nu_1) \in regEval(\rho_1, \mathcal{G})\} \\
\rho_1 \cup \rho_2 & : & \{(k_1 \sqcup k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G})_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \perp\} \\
\rho_1 \cap \rho_2 & : & \{(k_1 \sqcap k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G})_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \perp\} \\
\bar{\rho}_1 & : & \{(\bar{k}_1, \nu_1) \mid (k_1, \nu_1) \in regEval(\rho_1, \mathcal{G})\} \\
\rho_1^{-1}\rho_2 & : & \{(k_1^{-1}k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G})_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \perp\}
\end{array} \\
\text{end}
\end{array}
$$

Figure 3: Regular expression evaluation.

The evaluation of a formula is done recursively on its syntactical structure. To evaluate an equality proposition we find matching assignments that yield the same value for both terms, and in order to evaluate a conjunction, we need to find matching assignments that satisfy both conjuncts. Existential quantification is dealt with by projecting out the quantified variable. The algorithm in Figure 4 is just an initial attempt. The next section will present a number of improvements. An upper bound for the time complexity of this algorithm is $\mathcal{O}(|\varphi|^{2\|dom(\mathcal{G})\|+1})$, where $|\varphi|$ is the size of the formula i.e, the number of operators and $\|dom(\mathcal{G})\|$ is the size of the domain of $\mathcal{G}$. In computing this bound we assumed that the maximum size of a set computed by a call to *formEval* is $n^{\|dom(\mathcal{G})\|}$, where $n \leq |\varphi|$ is the number of existentially quantified variables in φ.

$$
\begin{array}{l}
formEval(\varphi, \mathcal{G}) \text{ is} \\
\text{case } \varphi \text{ in} \\
\begin{array}{rcl}
\rho_1 = \rho_2 & : & \{\nu_1 \bowtie \nu_2 \mid \exists k \in K.(k, \nu_i) \in regEval(\rho_i, \mathcal{G})_{i=1,2}, \nu_1 \bowtie \nu_2 \neq \perp\} \\
\varphi_1 \wedge \varphi_2 & : & \{\nu_1 \bowtie \nu_2 \mid \nu_i \in formEval(\varphi_i, \mathcal{G})_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \perp\} \\
\neg\varphi_1 & : & (FV(\varphi_1) \mapsto dom(\mathcal{G})) \setminus formEval(\varphi_1, \mathcal{G}) \\
\exists X\ \varphi_1 & : & \{\nu \mid \exists k\ [X \to k]\nu \in formEval(\varphi_1, \mathcal{G})\}
\end{array} \\
\text{end}
\end{array}
$$

Figure 4: Logic formula evaluation.

4.1 Performance Improvements

It appears that the main problem with the algorithm in Figure 4 is the exponential size of the sets of assignments calculated by *formEval*. In this section we propose two performance improvements, a first one attempting to keep the size of these sets as low as possible, and another one based on a symbolic representation of these sets. The reduction techniques are orthogonal and can therefore be used in combination.

4.1.1 Context-dependent Evaluation

When evaluating a conjunction $\varphi_1 \wedge \varphi_2$, after having obtained the set of assignments Λ_1 satisfying φ_1 we don't need to compute the same set for φ_2. It is easier to choose from Λ_1 only those assignments that satisfy φ_2. Having context information helps maintaining the sizes of the sets that are handled by the algorithm as small as possible.

A potential problem with this approach is the use of complementation, when evaluating the negation of a formula. In order to evaluate $\neg\varphi$ in the context Λ, we need to evaluate φ in the context $\overline{\Lambda} = (FV(\varphi) \mapsto dom(\mathcal{G})) \setminus \Lambda$. Hence, if Λ was small, $\overline{\Lambda}$ would turn big, which is against our purposes of improvement. To attack this problem, we chose to work with formulas in positive normal form, negation being pushed down to the atomic propositions using De Morgan's rules. In doing this, the size of the formula is only linearly increased (at most doubled).

Before describing the improved algorithm, we need to introduce some primitive operations on sets of assignments. Let $\Lambda_{1,2} \subseteq Var \mapsto K$ be such sets. For any assignments $\nu_{1,2} : Var \to K$, we define the relation $\nu_1 \preceq \nu_2$ if and only if, for all $X \in Var$, if $\nu_1(X)$ is defined, then $\nu_2(X) = \nu_1(X)$. With this notation, we have:

$$\begin{aligned}\Lambda_1 \Join \Lambda_2 &= \{\nu \mid \exists \nu_1 \in \Lambda_1 \,.\, \nu_1 \preceq \nu \text{ and } \exists \nu_2 \in \Lambda_2 \,.\, \nu_2 \preceq \nu\} \\ \Lambda_1 \uplus \Lambda_2 &= \{\nu \mid \exists \nu_1 \in \Lambda_1 \,.\, \nu_1 \preceq \nu \text{ or } \exists \nu_2 \in \Lambda_2 \,.\, \nu_2 \preceq \nu\}\end{aligned}$$

An alternative definition of $\Join$ could have been given as a pointwise extension of $\bowtie$:

$$\Lambda_1 \Join \Lambda_2 = \{\nu_1 \bowtie \nu_2 \mid \nu_1 \in \Lambda_1, \nu_2 \in \Lambda_2, \nu_1 \bowtie \nu_2 \neq \bot\}$$

The following operation projects out the variable X from all assignments in Λ:

$$\Lambda \downarrow_X = \{\nu \mid \exists k \,.\, [X \to k]\nu \in \Lambda\}$$

We extend this operation to sets of variables in the usual way. Let $Var(\Lambda)$ denote the set of variables, each of which is defined in at least one assignment from Λ. We can now define the following operation:

$$\Lambda_1 \nabla \Lambda_2 = (\Lambda_1 \Join \Lambda_2) \downarrow_{Var(\Lambda_1)} = \{\nu_1 \in \Lambda_1 \mid \exists \nu_2 \in \Lambda_2 \,.\, \nu_1 \bowtie \nu_2 \neq \bot\}$$

In other words, the ∇ operator is what allows us to filter out from Λ_1 all assignments that do not match with any assignment from Λ_2. This operator is used in the improved version of the algorithm (Figure 5) to trim out the assignments that have become unnecessary, from the sets handled by the $formEval$ function.

The main differences with respect to the initial algorithm (Figures 3 and 4) occur in the presence of the third parameter Λ, representing the evaluation context, used to trim the result on almost every branch of the case statement. The context itself is updated when evaluating a conjunction, see the definition of Λ_2 in the $\varphi_1 \wedge \varphi_2$ case. Also notice the use of $\Join$ to evaluate conjunctions and $\uplus$ to evaluate disjunctions.

$regEval(\rho, \mathcal{G}, \Lambda)$ is
case φ in

$$
\begin{array}{rcl}
k & : & \{(k, [])\} \\
X & : & \{(k, [X \to k]) \mid \exists \nu \in \Lambda \,.\, \nu(X) = k\} \\
\rho_1 \cdot \rho_2 & : & \{(k_1 \odot k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda \\
\rho_1^* & : & \{(k_1^*, \nu_1) \mid (k_1, \nu_1) \in regEval(\rho_1, \mathcal{G})\} \\
\rho_1 \cup \rho_2 & : & \{(k_1 \sqcup k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda \\
\rho_1 \cap \rho_2 & : & \{(k_1 \sqcap k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda \\
\bar{\rho}_1 & : & \{(\bar{k}_1, \nu_1) \mid (k_1, \nu_1) \in regEval(\rho_1, \mathcal{G}, \Lambda)\} \\
\rho_1^{-1} \rho_2 & : & \{(k_1^{-1} k_2, \nu_1 \bowtie \nu_2) \mid (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda
\end{array}
$$

end

$formEval(\varphi, \mathcal{G}, \Lambda)$ is
case φ in

$$
\begin{array}{rcl}
\rho_1 = \rho_2 & : & \{\nu_1 \bowtie \nu_2 \mid \exists k\ (k, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda \\
\rho_1 \neq \rho_2 & : & \{\nu_1 \bowtie \nu_2 \mid \exists (k_i, \nu_i) \in regEval(\rho_i, \mathcal{G}, \Lambda)_{i=1,2},\ k_1 \neq k_2,\ \nu_1 \bowtie \nu_2 \neq \bot\} \nabla \Lambda \\
\varphi_1 \wedge \varphi_2 & : & \text{let } \Lambda_1 = formEval(\varphi_1, \mathcal{G}, \Lambda) \text{ in} \\
 & & \quad \text{let } \Lambda_2 = formEval(\varphi_2, \mathcal{G}, \Lambda \nabla \Lambda_1) \text{ in } (\Lambda_1 \bowtie \Lambda_2) \nabla \Lambda \\
\varphi_1 \vee \varphi_2 & : & \text{let } \Lambda_1 = formEval(\varphi_1, \mathcal{G}, \Lambda), \\
 & & \quad \text{let } \Lambda_2 = formEval(\varphi_2, \mathcal{G}, \Lambda) \text{ in } (\Lambda_1 \uplus \Lambda_2) \nabla \Lambda \\
\exists X\ \varphi_1 & : & \text{let } \Lambda_X = \{[X \to k]\nu \mid k \in K,\ \nu \in \Lambda\} \text{ in} \\
 & & \quad \{\nu \mid \exists k.[X \to k]\nu \in formEval(\varphi_1, \mathcal{G}, \Lambda_X)\} \nabla \Lambda \\
\forall X\ \varphi_1 & : & \text{let } \Lambda_X = \{[X \to k]\nu \mid k \in K,\ \nu \in \Lambda\} \text{ in} \\
 & & \quad \{\nu \mid \forall k.[X \to k]\nu \in formEval(\varphi_1, \mathcal{G}, \Lambda_X)\} \nabla \Lambda
\end{array}
$$

end

Figure 5: Context-dependent evaluation.

4.1.2 Symbolic Representation of Assignment Sets

We propose as symbolic representation for assignment sets the *Kleene Decision Diagrams* (KDDs for short) that are an extension of Multivalued Decision Diagrams introduced in [15]. KDDs are rooted acyclic directed graphs where inner nodes are labeled with variables occurring in formula, edges are labeled with sets of values (see figure 7) and leaf nodes are either **0** or **1**. With the restriction that sets attached to edges outgoing from the same node are pairwise disjoint, each node in a KDD denotes a set of assignments. This set corresponds to the union of assignments obtained along each path starting at that node and leading to the leaf **1**.

In this work, we consider only *ordered* KDDs where, on each path starting from the root node variables occurs accordingly to some predefined order. Ordered KDDs admit a minimal canonical form which can be obtained by the application of the following reduction rules: (i) merge isomorphic nodes, (ii) merge edges leading to the same node by taking the union of the attached set of values, (iii) suppress edges labeled with the empty set, (iv) suppress edges leading to **0** and (v) replace inner nodes without successors with **0**.

The benefit of using KDDs for model checking is twofold. On one hand, they allow for compact representation of assignments sets. Using them it is possible to achieve an exponential reduction of the space and time required by the algorithm since there exist assignment sets of exponential size which can be represented with KDDs of polynomial size. On the other hand, they allow for an efficient manipulation i.e, all the operations used by the model-checking algorithm can be implemented directly on KDDs and do not need the explicit enumeration of the encoded elements. We present below algorithms for the main operations required by the model checking using KDDs: *join* $\bowtie$, *merge* $\uplus$, *existential quantification* $\downarrow_X$ and *restriction* ∇. Using standard BDD-like optimization techniques e.g, unique node representation, caching, etc all these operations have linear complexity with respect to the size of the arguments. The following primitives are used:

- *var(n)* returns the variable attached to the node n,
- *index(x)* the index of the variable x according to the predefined variable ordering,
- *succ(n)* returns the set of successors for the node n, that are pairs of the form (v, n') with v being the set of values and n' the corresponding target node.
- *mknode(x, l)* returns a node labeled with x and having the set of successors l.

With these notations, the algorithms for *join*, *merge* and existential quantification (variable elimination) *exist* are given below.

$\underline{\mathit{join}}(n_1, n_2)$ =
  if $n_1 = \mathbf{1}$ or $n_2 = \mathbf{0}$ then return n_2
  if $n_2 = \mathbf{1}$ or $n_1 = \mathbf{0}$ then return n_1
  let $x_k := \mathit{var}(n_k)_{k=1,2}$, $l_k := \mathit{succ}(n_k)_{k=1,2}$,
  if $\mathit{index}(x_1) < \mathit{index}(x_2)$ then
    return $\mathit{mknode}(x_1, \{(v_i, \underline{\mathit{join}}(n'_i, n_2)) \mid (v_i, n'_i) \in l_1\})$
  if $\mathit{index}(x_1) = \mathit{index}(x_2)$ then
    return $\mathit{mknode}(x_1, \{(v_i \cap v_j, \underline{\mathit{join}}(n'_i, n'_j)) \mid (v_i, n'_i) \in l_1, (v_j, n'_j) \in l_2\})$
  if $\mathit{index}(x_1) > \mathit{index}(x_2)$ then
    return $\mathit{mknode}(x_2, \{(v_j, \underline{\mathit{join}}(n_1, n'_j)) \mid (v_j, n'_j) \in l_2\})$

$\underline{\mathit{merge}}(n_1, n_2)$ =
  if $n_1 = \mathbf{1}$ or $n_2 = \mathbf{0}$ then return n_1
  if $n_2 = \mathbf{1}$ or $n_1 = \mathbf{0}$ then return n_2
  let $x_k := \mathit{var}(n_k)_{k=1,2}$, $l_k := \mathit{succ}(n_k)_{k=1,2}$, $u_k = \cup\{v|(v, n') \in l_k\}_{k=1,2}$
  if $\mathit{index}(x_1) < \mathit{index}(x_2)$ then
    return $\mathit{mknode}(x_1, \{(v_i, \underline{\mathit{merge}}(n'_i, n_2)) \mid (v_i, n'_i) \in l_1\} \cup \{(\overline{u_1}, n_2)\})$
  if $\mathit{index}(x_1) = \mathit{index}(x_2)$ then

return $mknode(x_1, \{(v_i \cap v_j, \underline{merge}(n'_i, n'_j)) \mid (v_i, n'_i) \in l_1, (v_j, n'_j) \in l_2\})$
if $index(x_1) > index(x_2)$ then
return $mknode(x_2, \{(v_j, \underline{merge}(n_1, n'_j)) \mid (v_j, n'_j) \in l_2\} \cup \{(\overline{u_2}, n_1)\})$

$\underline{exist}(n, X_e)$ =
if $n = \mathbf{1}$ or $n = \mathbf{0}$ then return n
let $x := var(n), l := succ(n)$
if $x \in X_e$ then
let $m := \mathbf{0}$
foreach $(v, n') \in l$ do
$m := \underline{merge}(m, exists(n', X_e))$
return m
else
return $mknode(x, \{(v, \underline{exist}(n', X_e)) \mid (v, n') \in l\})$

$\underline{restrict}(n_1, n_2)$ =
return $\underline{exist}(\underline{join}(n_1, n_2), vars(n_2) \setminus vars(n_1))$

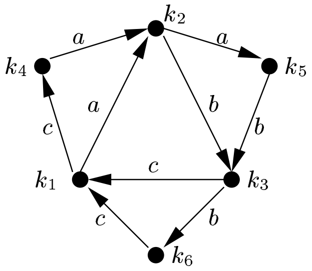
To keep the size of the KDDs manageable, we assume a function *canonicalize(n)* that returns the canonical form of the KDD at node n. Such a function is standard in most BDD-like packages.

4.1.3 An example

Consider the graph and the property from Figure 6. The formula $\exists X, Y, Z \, . \, Y \not\Vdash Xa \wedge (Z \not\Vdash Yb \ \wedge X \not\Vdash Zc)$ states the existence of a cycle labeled with abc within the graph. The graph is assumed to be deterministic over some Kleene algebra K generated by $\{a, b, c\}$ and one can easily compute the values of the nodes $k_{1,\ldots,6}$ using the operations in the algebra. For simplicity, the graph is not rooted, but this doesn't affect the correct functioning of our algorithm.

In Figure 7 (upper part) we have represented the assignment sets computed by the standard evaluation algorithm for the formula and all of its subformulas. The lower part, contains the assignment sets computed by the context-dependent evaluation algorithm. Here the sets Λ_a, Λ_b, etc. are the ones represented by the KDD's depicted in Figure 7 (a_1), (b_1), and so on. One can observe that, not only the assignment set but also the KDD representations are smaller when evaluation contexts are taken into account. For instance, the evaluation of $Z \not\Vdash Yb$ in the context satisfying $Y \not\Vdash Xa$ discards the assignment $\{Z \mapsto k_6, Y \mapsto k_3\}$. Moreover, the evaluation of $X \not\Vdash Zc$ in the context where both $Y \not\Vdash Xa$ and $Z \not\Vdash Yb$ hold discards the assignments $\{Z \mapsto k_1, X \mapsto k_4\}$ and $\{Z \mapsto k_6, Y \mapsto k_1\}$.

The picture doesn't represent the last three steps of evaluation, consisting in projecting out the existentially quantified variables X, Y and Z. The result of these operations leaves us with a KDD consisting only of the bottom $\mathbf{1}$ node, meaning that the formula is satisfied.



$$\exists X, Y, Z \,.\, Y \nparallel Xa \;\wedge\; (Z \nparallel Yb \;\wedge\; X \nparallel Zc)$$

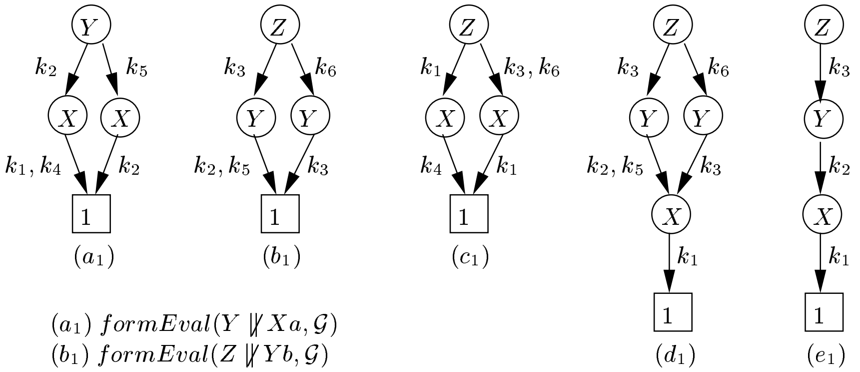
Figure 6: The graph and the property.

5 Conclusions and Applications

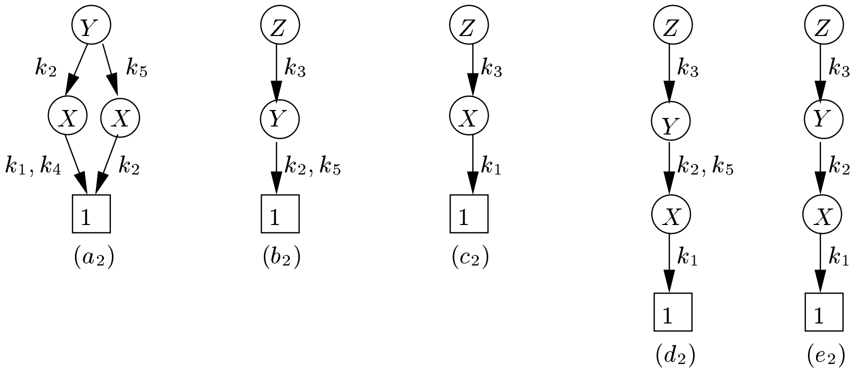
We have presented a method for checking properties of generic graphs. We introduced a representation of graphs as functions on Kleene algebras, which allows us to encompass a large classes of graphs. The structural properties are written as first-order formulas in which the atomic propositions are equivalences of regular expressions containing variables. We have studied the complexity of this logic and gave examples of interesting properties. Finally, a model checking algorithm is given. Although very expensive in theory, the performance of this algorithm can be greatly improved by caching previously computed results and using a symbolic representation of the data. The symbolic representation we use is a variant of MDDs provided with special operations that are required by our algorithm.

We envisage at least two concrete classes of applications for the DKA logic. One is in *shape analysis* [14] and consists in interpreting the outcome of an analysis with respect to a property of the program heap stated in DKA. The ability of describing recursive data structures and relations between structures comes into role now. The output of a shape analysis at a given program point is a finite set of (abstract) shape graphs, each of which represents possible heap configurations. By verifying whether some of these graphs are models of a given (annotated) formula answers general questions that can be asked by the user of the analysis which are otherwise difficultly encoded in the underlying representation used by the analysis. Also an iterative property-guided refinement of the analysis results could be designed, based on the model checking algorithm proposed in this paper.

Another direction is to check databases, websites and high-level specifications of systems such as UML diagrams against correctness requirements expressible in DKA. For instance, the existence of isomorphic components is important in a distributed database for security reasons, whereas for a website it is important to make sure that the English and the French versions have exactly the same link structure. In case one deals with safety-critical architectures, replicated components need to be organized in an isomorphic way in order to provide equal functionalities. The existence of cliques and strongly connected components is important in what concerns distributed computer architectures, in which the communication topology plays an important role. Obviously, for different applications, one can design more specific properties than the ones presented in Section 3.



(a_1) $formEval(Y \nparallel Xa, \mathcal{G})$
(b_1) $formEval(Z \nparallel Yb, \mathcal{G})$
(c_1) $formEval(X \nparallel Zc, \mathcal{G})$
(d_1) $formEval(Z \nparallel Yb \wedge X \nparallel Zc, \mathcal{G})$
(e_1) $formEval(Y \nparallel Xa \wedge (Z \nparallel Yb \wedge X \nparallel Zc), \mathcal{G})$



(a_2) $formEval(Y \nparallel Xa, \mathcal{G}, \perp)$
(b_2) $formEval(Z \nparallel Yb, \mathcal{G}, \Lambda_a) \nabla formEval(Y \nparallel Xa), \mathcal{G}, \Lambda_a)$
(c_2) $formEval(X \nparallel Zc, \mathcal{G}, \Lambda_b) \nabla (formEval(Y \nparallel Xa, \mathcal{G}, \Lambda_b) \nabla formEval(Z \nparallel Yb, \mathcal{G}, \Lambda_b)))$
(d_2) $formEval(Z \nparallel Yb \wedge X \nparallel Zc, \mathcal{G}, \Lambda_c) \nabla formEval(Y \nparallel Xa, \mathcal{G}, \Lambda_c)$
(e_2) $formEval(Y \nparallel Xa \wedge (Z \nparallel Yb \wedge X \nparallel Zc), \mathcal{G}, \Lambda_d)$

Figure 7: Evaluation using the standard and improved algorithm.

References

[1] L. de Alfaro: Model checking the world wide web. In Computer Aided Verification, Volume 2102 of Lecture Notes in Computer Science, Springer-Verlag, pp. 337–349 (2001). 1.1

[2] Benedikt, M., Reps, T., and Sagiv, M.: A decidable logic for describing linked data structures. European Symposium on Programming, (1999) LNCS, Vol. 1576, 2–19. 1.1

[3] M. Bozga, R. Iosif and Y. Lakhnech: Storeless Semantics and Alias Logic. Proc. ACM SIGPLAN 2003 Workshop on Partial Evaluation and Semantics Based Program Manipulation, 55 – 65. 1, 2.1

[4] H. Comon. Disunification: A survey. In Computational Logic: Essays in Honor of Alan Robinson. The MIT Press, Cambridge, Mass., (1991) 2

[5] B. Courcelle: The expression of graph properties and graph transformations in monadic second-order logic, Chapter 5 of the "Handbook of graph grammars and computing by graph transformations, Vol. 1 : Foundations" (1997) 313–400 1, 1.1, 3, 3.0.1, 3.0.3, 3.0.4, 3.0.5

[6] E. M. Clarke, O. Grumberg and D. Peled: Model Checking. MIT Press (2001) 1

[7] A. Deutsch: A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In Proceedings of the IEEE 1992 Conference on Computer Languages (1992) 2–13 1, 2.1, 2.1

[8] D. Harel: Recurring Dominoes: Making the Highly Undecidable Highly Understandable. Annals of Discrete Mathematics 24 (1985), pp 51 – 72 2.3

[9] D. Kozen: Results on the Propositional Mu-Calculus, Theoretical Computer Science 27 (1983), pp. 335–354. 1, 3

[10] M. Hopkins and D. Kozen: Parikh's theorem in commutative Kleene algebra. Proc. IEEE Conf. Logic in Computer Science (LICS'99), IEEE, July 1999, 394-401. 2.3, 2.3

[11] Kleene, S. C: Introduction to metamathematics, (North-Holland), 1952. 2.2.2

[12] H. B. M. Jonkers. Abstract Storage Structures. Algorithmic Languages, North-Holland (1981) 321–343 1, 2.1, 2.1

[13] D, Jungnickel: Graphs, Networks and Algorithms. Springer-Verlag (1999)

[14] M. Sagiv, M., T. Reps and R. Wilhelm: Parametric Shape Analysis via 3-Valued Logic. ACM Transactions on Programming Languages and Systems, Vol 24, No 3 (2002), pp. 217–298 1, 5

[15] A. Srinivasan and T. Kam and S. Malik and R.E. Bryant: Algorithms for discrete function manipulation. Proceedings of Int'l Conference on Computer-Aided Design, pp. 92–95 (1990) 4.1.2

[16] A. Meyer and L. Stockmeyer: The equivalence problem for regular expressions with squaring requires exponential space. In Proceedings of the 13th IEEE Symposium on Switching and Automata Theory (1972) pp. 125–129

2.3

A Proofs

Proof of Lemma 1: By reduction from (T1). Assume we are given a set T of tiles over the set C of colors. Let h and v be two distinct symbols, and $H = C \times \{h\}$, $V = C \times \{v\}$ be two disjoint alphabets used to model horizontal and vertical edges, and let $\Sigma = H \cup V$. The root alphabet is $\Pi = \{root\}$. The following formula states that all nodes in all models are only reachable from the root by combinations of symbols from Σ:

$$\forall X \,.\, X \subseteq \Pi \cdot \Sigma^* \tag{2}$$

Next, there can only be one horizontal and one vertical neighbor for each node:

$$\forall X \forall Y' \forall Y''.[(Y' \not\Vdash XH \wedge Y'' \not\Vdash XH) \vee (Y' \not\Vdash XV \wedge Y'' \not\Vdash XV)] \rightarrow Y' = Y'' \tag{3}$$

The following forbids the existence of cycles in the structure:

$$\forall X, Y \,.\, X \not\Vdash Y\Sigma^* \rightarrow Y \not\Vdash X\Sigma^* \tag{4}$$

For some $c \in C$, let $adj_H(X, Y, c) = Y \not\Vdash X \cdot (c, h) \wedge \bigwedge_{d \in C \setminus \{c\}} \neg Y \not\Vdash X \cdot (d, h)$ and define adj_V similarly, by replacing h with v. In other words, two adjacent nodes can be linked by only one edge. A weaker definition is $weak\_adj_{H,V}$ obtained from $adj_{H,V}$ respectivelly, by forgetting the second (big) conjunct. The following predicate states the existence of a tile t_k starting with X as the bottom-left node:

$$\begin{aligned} tile(X, k) = \exists X_1, X_2, X_3 \quad . \quad & adj_H(X, X_1, t_k^d) \wedge adj_V(X, X_2, t_k^l) \wedge \\ & adj_H(X_2, X_3, t_k^u) \wedge adj_V(X_1, X_3, t_k^r) \end{aligned}$$

We also define $weak\_tile$ similarly, except that we use $weak\_adj_{H,V}$ predicates instead. We can now assert that each node is the bottom-left node of a tile:

$$\forall X \,.\, \bigvee_{k=0}^{m} tile(X, k) \tag{5}$$

The conjunction of (2), (3), (4) and (5) is satisfiable if and only if there exists an infinite tiling of $\mathbb{N} \times \mathbb{N}$ with tiles from T. Notice that (3), the exclusive definition of $adj_{H,V}$ and (5) ensure that there exists only one tile placed at each node, and that two adjacent tiles respect the constraint of the tiling problem. The lack of cycles (4) ensures that X, X_1, X_2, X_3 in the definition of $tile$ are pairwise distinct. $\Box$

Proof of Lemma 2: Let T be a set of tiles over the set of colors C and H, V defined as in the proof of Lemma 1. NP-hardness is shown by reduction from (T2). Let $\mathcal{G}$ be the $n \times n$ grid in which each node $(i, j), 0 \leq i < n - 1, 0 \leq j < n$ is connected to $(i + 1, j)$ via a (c, h)-edge and each $(i, j), 0 \leq i < n, 0 \leq j < n - 1$ is connected to $(i, j + 1)$ via a (c, v)-edge for each $c \in C$.

Let $at(i, j, k) = \exists X \,.\, X \nparallel H^i V^j \wedge weak\_tile(X, k)$, where $weak\_tile$ is defined as in the proof of Lemma 1. Next, we code the constraints imposed by the tiling problem:

$$\begin{aligned} hor(i, j) &= \bigwedge_{k=0}^{m} at(i, j, k) \rightarrow \bigvee_{l=0}^{m} at(i + 1, j, l) \wedge t_k^r = t_l^l \\ ver(i, j) &= \bigwedge_{k=0}^{m} at(i, j, k) \rightarrow \bigvee_{l=0}^{m} at(i, j + 1, l) \wedge t_k^u = t_l^d \end{aligned}$$

There exists a tiling of $n \times n$ by T, if and only if $\mathcal{G}$ is a model of:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n-1} hor(i, j) \wedge ver(i, j)$$

□