Juggrnaut – An Abstract JVM

Jonathan Heinen, Henrik Barthels, and Christina Jansen

Software Modeling and Verification Group
RWTH Aachen University, Germany
`http://moves.rwth-aachen.de/`

Abstract. We introduce a new kind of hypergraphs and hyperedge replacement grammars, where nodes are associated types. We use them to adapt the abstraction framework *Juggrnaut* presented by us in [7, 8] – for the verification of Java Bytecode programs. The framework is extended to handle additional concepts needed for the analysis of Java Bytecode like null pointers and method stacks as well as local and static variables. We define the abstract transition rules for a significant subset of opcodes and show how to compute the abstract state space. Finally we complete the paper with some experimental results.

1 Introduction

Object-oriented languages, used in most software projects these days, introduce new challenges to software verification. As objects can be created on runtime the state space is (potentially) infinite, thus making it impossible to apply standard verification techniques.

In [7, 8] we presented an abstraction framework *Juggrnaut* based on a natural representation of heaps by graphs. We employ hyperedge replacement grammars to specify data structures and their abstractions. The key idea is to use the replacements which are induced by the grammar rules in two directions. By a backward application a subgraph of the heap is condensed into a single nonterminal hyperedge (as depicted in Fig. 1), thus obtaining an abstraction of the heap. By applying rules in forward direction, parts of the heap that have been abstracted before can be concretised again.

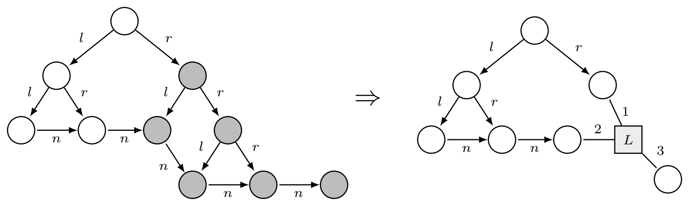


Fig. 1. Representation of abstracted heap parts by nonterminal hyperedges.

The structure in Fig. 1 describes a tree with linked leaves, i.e. the leaves form a linked list, a common data structure in data bases. This data structure is used as a running example in Section 2.

While in previous publications our focus was on pointer properties like destructive updates, sharing and dynamic allocation of heap cells, we did not consider object-oriented aspects like member variables (fields), object methods, polymorphism and other type depending instructions (e.g. **instanceof**). Furthermore (recursive) method calls and local variables, potentially resulting in an unbounded number of variables, were not considered so far. In this paper we extend the framework to handle a significant subset of Java Bytecode including the aspects listed above. In Section 2 we provide the theoretical foundations, extending hypergraphs with typed nodes to model typed objects in object-oriented languages (e.g. Java, C++, Objective-C, ...). The introduction of types involves an adaptation of hyperedge replacement grammars used for heap abstraction as well as of the requirements formulated by us before [8]. In Section 3 we extend the *Juggrnaut* framework towards an abstract Java Virtual Machine (JVM) supporting Java Bytecode specific concepts like static and dynamic method calls as well as static and local variables. Technically this is achieved by extending the abstract graph representation of the heap to cover the entire state of the JVM.

Proofs omitted due to space restrictions are found in a technical report [9].

2 Basic Concepts

Given a set S, $S^\star$ is the set of all finite sequences (strings) over S including the empty sequence ε, where $\cdot$ concatenates sequences. For $s \in S^\star$, the length of s is denoted by $|s|$, the set of all elements of s is written as $[s]$, and by $s[i]$, with $1 \leq i \leq |s|$, we refer to the i-th element of s. We denote the disjoint sum by $\uplus$. Given a tuple $t = (A, B, C, \dots)$ we write A_t, B_t etc. for the components if their names are clear from the context. Function $f{\restriction}S$ is the restriction of f to S. Function $f : A \to B$ is lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^\star \to B^\star$ by point-wise application. We denote the identity function on a set S by id_S.

2.1 Hypergraphs and Heaps

In [8] heap structures are represented by hypergraphs. Hypergraphs contain edges connecting arbitrary many nodes. They are labelled using a ranked alphabet Γ of terminals and nonterminals. A ranking function $\mathrm{rk} : \Gamma \to \mathbb{N}$ maps each label l to a rank, defining the number of nodes an l-labelled edge connects.

Example 1. Consider the right graph of Fig. 1. Nodes (depicted as circles) represent objects on the heap. Edges are labeled using an alphabet Γ. Terminal edges (labelled by terminals) connecting two nodes represent pointers, whereas nonterminal edges (depicted as shaded boxes) represent abstracted heap parts. In Fig. 1 an L-labelled hyperedge connects three nodes, i.e. $\mathrm{rk}(L) = 3$. The order on attached nodes is depicted by numbers labelling connection lines (which we call tentacles) between edges and nodes. In case of terminal edges, the direction induces the order.

Definition 1 (Tentacle). *A tuple* $(a, i), a \in \Gamma, 1 \leq i \leq \mathrm{rk}(a)]$ *is a* tentacle*.*

Although objects in common object-oriented languages are of a well-defined type, this is not reflected in this representation. We therefore extend the graph model to (labelled and typed) hypergraphs over a typed alphabet assigning a type to each node.

Definition 2 (Typed Alphabet). *A* typed alphabet Σ *is a triple* $(L, (T, \preceq), \mathit{types})$ *with set of labels* L*, type poset* $(T, \preceq)$ *and typing function* $\mathit{types} : L \to T^\star$.

Note that the ranking function is now implicitly given by $\mathrm{rk}(f) = |\mathit{types}(f)|$. The value of $\mathit{types}(X)[i]$ defines the type of node to which tentacle (X, i) connects.

We define the relation $\preceq$ also over sequences of types, where two type sequences $t_1 \preceq t_2$ are related iff they are equal in length and if for each position the elements are related correspondently, i.e. $t_1[i] \preceq t_2[i], \forall 1 \leq i \leq |t_1|$.

Definition 3 (Hypergraph). *A* (labelled and typed) hypergraph (HG) *over a typed alphabet* Σ *is a tuple* $H = (V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \mathit{ext})$*, with a set of nodes* V*, a set of edges* E*, an edge labelling function* $\mathit{lab} : E \to L_\Sigma$ *and a node labelling function* $\mathit{type} : V \to T_\Sigma$*. The attachment function* $\mathit{att} : E \to V^*$ *maps each hyperedge to a node sequence and* $\mathit{ext} \in V^\star$ *is a (possibly empty) sequence of pairwise distinct external nodes.*
For $e \in E$ *we require that* $\mathit{types}(\mathit{lab}(e)) \succeq \mathit{type}(\mathit{att}(e))$*, i.e. every tentacle is connected to a node of its corresponding type or a subtype.*

The set of all hypergraphs over Σ *is denoted by* HG_Σ.

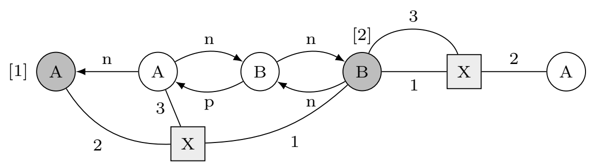


Fig. 2. A labelled and typed hypergraph.

Example 2. Fig. 2 depicts a hypergraph over the typed alphabet $\Sigma = (\{n, p, X\}, (\{A, B\}, B \preceq A), [n \mapsto AA, p \mapsto BA, X \mapsto BAA])$. Each node is annotated with its type. The order on the (grey) external nodes is given by numbers in square brackets next to them. Edge connections must respect the *types* function, e.g., tentacles $(X, 3)$ and $(X, 1)$ of the right X-edge are mapped to the same node of type B. This is correct as $B \preceq \mathit{types}(X)[1]$ and $B \preceq \mathit{types}(X)[3]$.

We use hypergraphs to model heaps where terminal edges represent pointers and nonterminal edges represent embedded heap structures of a well defined shape. Though the hypergraph depicted in Fig. 2 is correct according to the definition of a hypergraph, it does not represent a proper heap. Indeed there are two problems. The first one occurs at the second node from left, which has two outgoing n-pointers. The second is based on the fact that every pointer on the heap has to be represented either concretely as a terminal edge or abstractly within a hyperedge. Thus there should be an outgoing p-pointer at the second external node. As it is not concretely represented it has to be abstracted within

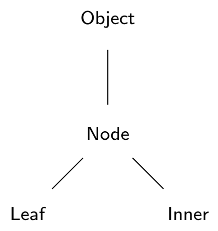
the X, either in the first or in the third tentacle. However if it was abstracted in the first tentacle there would be two p-pointers at external node two (as two $(X, 1)$-tentacles are connected), whereas if it was abstracted within the third tentacle there would be a p-pointer at the second node from left, which is of type A, thus has no p-pointer.
In order to formalise the requirements we introduce the notion of entrance- and reduction-tentacles, also named $\triangledown$- and $\blacktriangle$-tentacles, respectively. Nodes can be left via $\triangledown$-tentacles, i.e. they represent outgoing pointers, whereas $\blacktriangle$-tentacles represent incoming ones only. Given a set A we use: $A_{\blacktriangle} = A \times \{\blacktriangle\}$, $A_{\triangledown} = A \times \{\triangledown\}$ and $A_{\triangledown\blacktriangle} = A_{\blacktriangle} \cup A_{\triangledown}$. We denote $(a, \blacktriangle)$ and $(a, \triangledown)$ by $a_{\blacktriangle}$ and $a_{\triangledown}$, respectively. If we use elements of $A_{\triangledown\blacktriangle}$ where elements from A are expected, we refer to the projection on the first element.

Definition 4 (Heap Alphabet). *A* heap alphabet $\Sigma_N = (\mathbb{F} \cup N, (\mathbb{T}, \preceq), \mathit{types})$ *is a tuple with a set of field labels* $\mathbb{F}$*, a set of nonterminals* N*, a set of types* $\mathbb{T}$ *and a typing function types* : $\mathbb{F} \cup N \to {\mathbb{T}_{\triangledown\blacktriangle}}^{\star}$*, where* $\mathit{types}(\mathbb{F}) \subseteq \mathbb{T}_{\triangledown} \cdot \mathbb{T}_{\blacktriangle}$*.*

$\mathbb{T}$ corresponds to the classes with subtype relation $\preceq$, whereas $\mathbb{F}$ are the field names. The function *types* maps fields to their defining class (as $\triangledown$-tentacle) and to the class they point to (as $\blacktriangle$-tentacle), e.g. given the class definition **class** A{ B f;} $\mathit{types}(f) = A_{\triangledown} B_{\blacktriangle}$. We define $\mathit{fields}(t) = \{f \in \mathbb{F} \mid \mathit{types}(f)[1] \succeq t_{\triangledown}\}$ for $t \in \mathbb{T}$.

```
class Node{
  Inner parent;
}
class Inner extends Node{
  Node left, right;
}
class Leaf extends Node{
  Leaf next;
}
```

(a) Class definition



(b) Type poset

Fig. 3. Class definitions and resulting poset.

Example 3. Given the Java-class definitions from Fig. 3(a) we get the set of types $\mathbb{T} = \{\mathrm{Object}, \mathrm{Node}, \mathrm{Inner}, \mathrm{Leaf}\}$ and as terminal edge labels the field-names $\mathbb{F} = \{\mathit{Node.parent}, \mathit{Inner.left}, \mathit{Inner.right}, \mathit{Leaf.next}\}$. The poset $(\mathbb{T}, \preceq)$ defined by the subtype relation is given in Fig. 3(b). The type sequence for parent is $\mathit{types}(\mathit{Node.parent}) = \mathrm{Node}_{\triangledown}\mathrm{Inner}_{\blacktriangle}$, for left and right $\mathit{types}(\mathit{Inner.left}) = \mathit{types}(\mathit{Inner.right}) = \mathrm{Inner}_{\triangledown}\mathrm{Node}_{\blacktriangle}$ and $\mathit{types}(\mathit{Leaf.next}) = \mathrm{Leaf}_{\triangledown}\mathrm{Leaf}_{\blacktriangle}$.

The resulting function *fields* is $\mathit{fields}(\mathrm{Node}) = \{\mathit{Node.parent}\}$, $\mathit{fields}(\mathrm{Inner}) = \{\mathit{Inner.left}, \mathit{Inner.right}, \mathit{Node.parent}\}$, $\mathit{fields}(\mathrm{Leaf}) = \{\mathit{Leaf.next}, \mathit{Node.parent}\}$.

By $\triangledown_H(v) = \{e \in E_H \mid (\exists i \in \mathbb{N} : \mathit{types}(\mathit{lab}(e))[i] \in \mathbb{T}_{\triangledown} \wedge \mathit{att}(e)[i] = v\}$ we define the set of edges connected to the node $v \in V_H$ through an entrance tentacle.

Definition 5 (Heap Configuration). *A* heap configuration (HC) *over a heap alphabet Σ_N is a tuple $H = (V, E, lab, type, att, ext)$, where V is a set of nodes, E a set of edges. An edge labelling function $lab : E \to \mathbb{F} \cup N$ and a node labelling function $type : V \to \mathbb{T}$. The function $att : E \to V^*$ maps each hyperedge to a sequence of attached nodes and $ext \in V^{\star}_{\blacktriangle\triangledown}$ is a (possibly empty) sequence of pairwise distinct external vertices.*

For a terminal edge $e \in E, lab(e) \in \mathbb{F}$ we require that $types(lab(e)) \succeq type(att(e))$ whereas for $e \in E, lab(e) \in N$ we require that $types(lab(e)) \succeq_N type(att(e))$, where $t_\blacktriangle \succeq_N t'_\blacktriangle$ iff $t \succeq t'$ and $t_\triangledown \succeq_N t'_\triangledown$ iff $t = t'$.

For $v_\blacktriangle \in ext$ we require $\triangledown_H(v) = \emptyset$, whereas for $v \in V$ such that $v_\blacktriangle \notin ext$ we require:

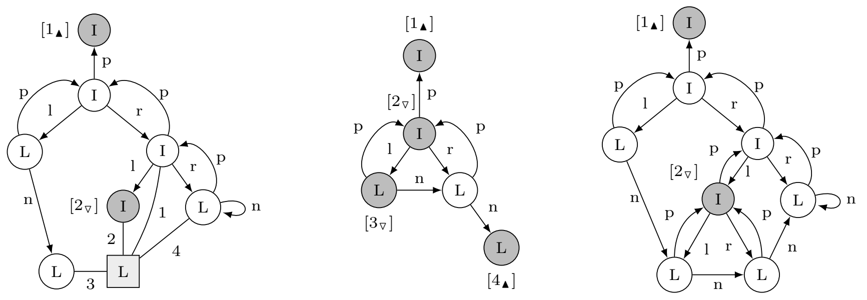
$$\begin{aligned} & lab(\triangledown_H(v)) = \mathit{fields}(type(v)) \;\wedge\; x, y \in \triangledown_H(v) \quad\Rightarrow\quad lab(x) \neq lab(y) && (1)\\ \vee\; & lab(\triangledown_H(v)) \subseteq N \;\wedge\; |\triangledown_H(v)| = 1 && (2) \end{aligned}$$

The set of all heap configurations over Σ_N is denoted by HC_{Σ_N}.

Whereas terminal $\triangledown$-tentacles represent a single outgoing pointer, non-terminal $\triangledown$-tentacles represent all outgoing pointers of a node. Therefore a node can be connected to either every non-reduction terminal tentacle defined by the type (1) or a single nonterminal $\triangledown$-tentacle (2).

External nodes can be considered to be references to nodes outside the graph and their outgoing pointers are either all outside the graph and the external node is annotated as reduction-node ($\blacktriangle$) or are all inside the graph and the external node is therefore an entrance node ($\triangledown$) as we can enter the graph from this node.

Example 4. In Fig. 4(a) a HC over the heap alphabet from Ex. 3 is given, extended by the nonterminal L with $types(L) = I_\blacktriangle I_\triangledown L_\triangledown L_\blacktriangle$. Here I is the short form for Inner, L for Leaf. Nonterminal edges labelled by L represent trees with linked leaves. The external nodes are: the root node (2) of a subtree, its parent node (1), the leftmost leaf (3) and the n-reference (4) of the rightmost leaf of the subtree. The numbering of external nodes is extended to mark them as $\triangledown$- or $\blacktriangle$-nodes. As the first external is a $\blacktriangle$-node it has no outgoing edges, whereas the second has abstracted outgoing edges represented by the $\triangledown$-tentacle $(L, 2)$.



(a) Heap configuration (b) Replacement Graph (c) Result

Fig. 4. An abstract heap configuration.

We use Σ to denote a heap alphabet Σ_N without nonterminals, i.e. $N = \emptyset$. If a HC does not contain nonterminal edges, i.e. is defined over a heap alphabet Σ we call it *concrete* ($H \in \mathrm{HC}_\Sigma$), otherwise *abstract* ($H \in \mathrm{HC}_{\Sigma_N}$).

2.2 Data Structure Grammars

Hyperedge replacement grammars can be used to describe heap structures. These grammars are defined as a set of rules each consisting of a nonterminal on the left-hand side and a hypergraph on the right-hand side.

Definition 6 (Hyperedge Replacement Grammar (HRG)). *A hyperedge replacement grammar (HRG) over a typed alphabet Σ_N is a set of production rules $p = X \to H$, with $X \in N$ and $H \in HG_{\Sigma_N}$, where $types(X) \preceq type(ext_H)$.*

We denote the set of all hyperedge replacement grammars over Σ_N by HRG_{Σ_N}.

Derivation steps of a HRG are defined by hyperedge replacement, i.e. a hyperedge e is replaced by a hypergraph by mapping the external nodes of the latter with attached notes of e. This replacement is possible only if the number and types of nodes connected by the replaced edge and of the external nodes in the replacement graph correspond to each other. This aspect is covered in the following adaption of the definition from [8] for labelled and typed hypergraphs.

Definition 7 (Hyperedge Replacement). *Given hypergraphs $H, I \in \mathrm{HG}_{\Sigma_N}$ and an edge $e \in E_H$ with $type(att_H(e)) \preceq type(ext_I)$, the replacement of the edge e in H by I is defined as $K = H[I/e] = (V_K, E_K, lab_K, type_K, att_K, ext_K)$:*

$$\begin{array}{lll}
V_K \ = V_H \uplus (V_I \setminus ext_I) & E_K \ = (E_H \setminus \{e\}) \uplus E_I & ext_K = ext_H \\
type_K = type_I{\restriction}V_K \uplus type_H & lab_K = lab_H{\restriction}E_K \uplus lab_I & \\
att_K \ = att_H \uplus att_I \circ (id_{V_I}{\restriction}V_K \cup \{ext_I(i) \mapsto att_H(e)(i) \mid i \in [1, |ext_I|]\}) & &
\end{array}$$

Example 5. Reconsider the HC H from Fig. 4(a) containing exactly one nonterminal edge e labelled with L. The rank of L is equal to the number of external nodes of the concrete HC I in Fig. 4(b), thus we can replace e by I and get $K = H[I/e]$, depicted in Fig. 4(c). Note that the result is again a HC $K \in \mathrm{HC}_\Sigma$, because $types(L) \preceq_N lab_I(ext_I)$ as stated in the following theorem.

Theorem 1 (Edge Replacement in HCs). *Given $H, I \in \mathrm{HC}_{\Sigma_N}$ and $e \in E_H$ with $types(lab_H(e)) \preceq_N type(ext_I)$ it holds that $H[I/e] \in \mathrm{HC}_{\Sigma_N}$. (Proof in [9])*

Definition 8 (Data Structure Grammar). *A* Data Structure Grammar *(or short DSG) over an abstract heap alphabet Σ_N is a set of production rules $p = X \to R$, with $X \in N$ and $R \in \mathrm{HC}_{\Sigma_N}$, where $types(X) \preceq_N lab_R(ext_R)$.*

We denote the set of all data structure grammars over Σ_N by DSG_{Σ_N}.

Given grammar $G \in \mathrm{DSG}_{\Sigma_N}$ and graph $H \in \mathrm{HC}_{\Sigma_N}$ we write $H \Rightarrow_G H'$ if there exists a production rule $X \to R \in G$ and an edge $e \in E_H$ with $lab_H(e) = X$ and $H' = H[R/e]$. We write $\Rightarrow_G^*$ for the reflexive transitive closure of $\Rightarrow_G$. We say H' is derivable from H by G iff $H \Rightarrow_G^* H'$.

Corollary 1. *Given a data structure grammar $G \in \mathrm{DSG}_{\Sigma_N}$ and $H \in \mathrm{HC}_{\Sigma_N}$ every derivable graph is a HC: $H \Rightarrow_G^* H' \Rightarrow H' \in \mathrm{HC}_{\Sigma_N}$.*

Example 6. Fig. 5 depicts a DSG for trees with linked leaves and parent pointers. The DSG consists of four rules for nonterminal L with $types(L) = I_\blacktriangle I_\triangledown L_\triangledown L_\blacktriangle$, introduced in Ex. 4. Every right hand side is a HC with $type(ext) = types(L)$.

The rules define the data structure recursively. The smallest tree represented by L is a tree where the child nodes of the root node are the two leaves. Bigger trees are defined recursively as trees where either one (second and third rule) or both (fourth rule) children of the root node are trees, with properly linked leaves.

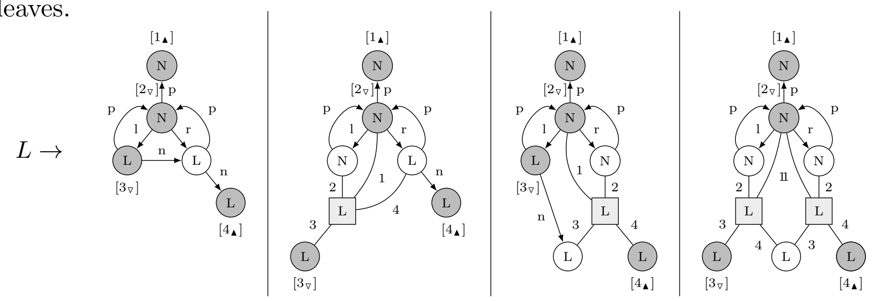


Fig. 5. DSG for trees with linked leafs.

As our grammar definition does not include a start symbol we define languages in dependence of a start configuration.

Definition 9 (Language). *For $G \in DSG_{\Sigma_N}$ we define the language $L_G(H)$ induced by a start graph $H \in \mathrm{HG}_{\Sigma_N}$ over G as the set of derivable concrete HGs: $L_G(H) = \{H' \in \mathrm{HG}_\Sigma \mid H \Rightarrow_G^* H'\}$, and for $M \subseteq \mathrm{HG}_{\Sigma_N} : L(M) = \bigcup_{H \in M} L(H)$.*

It follows from Corollary 1 that using an abstract HC as start graph and a data structure grammar the only derivable concrete graphs are again HCs. It remains to show that the restrictions in the definition of DSGs do not impair the expressiveness, i.e. that the languages representable by DSGs are exactly the HC languages ($\subseteq \mathrm{HC}_\Sigma$) representable by HRGs.

Theorem 2. *Given a HRG G over Σ_N. Then a grammar G' over Σ'_N can be constructed such that for every hypergraph S over Σ_N with $L_G(S) \subseteq \mathrm{HC}_\Sigma$ there exists a heap configuration S' with $L_{G'}(S') = L_G(S)$. (Proof in [9])*

For nonterminal X we use $X^\bullet$ to denote the X-handle, i.e. a hypergraph consisting of a single nonterminal edge labelled with X and one node for each tentacle:

$$\begin{array}{llll} V_{X^\bullet} & = \{v_i \mid i \in [1, |types(X)|]\} & E_{X^\bullet} & = \{e\} \\ type_{X^\bullet} & = \{v_i \mapsto types(X)[i] \mid i \in [1, |types(X)|]\} & lab_{X^\bullet} & = \{e \mapsto X\} \\ att_{X^\bullet} & = \{e \mapsto v_1 v_2 \dots v_{|types(X)|}\} & ext_{X^\bullet} & = \varepsilon \end{array}$$

Definition 10 (Local Greibach Normal Form [8]). *A grammar $G \in DSG_{\Sigma_N}$ is in* Local Greibach Normal Form (LGNF) *if for every non-reduction tentacle (X, i) there exists $G_{(X,i)} \subseteq G$ with:*

$$L_{G_{(X,i)}}(X^\bullet) = L_G(X^\bullet) \quad \textit{and} \quad \forall X \to R \in G_{(X,i)} : \nabla_R(ext_R(i)) \subseteq \mathbb{F}$$

Lemma 1. *Any DSG can be transformed into an equivalent DSG in LGNF [8].*

3 An Abstract Java Virtual Machine

Java programs are compiled to Java Bytecode programs that are executed by a *Java Virtual Machine* (JVM). In this section we define an abstract JVM for an significant subset of Java Bytecode, excluding threads and exceptions as well as data values, others then booleans. The introduced abstract JVM is based on HCs as defined in the previous chapter.

3.1 Java Bytecode and the JVM

Based on the formal definition of Java Bytecode and the JVM from [16], we distinguish between the static environment and the dynamic state of a JVM.

Static Environment of a JVM [16]. A JVM executes programs with respect to a static environment $cEnv : \mathsf{Class} \cup \mathsf{Interface} \to \mathit{ClassFile}$. For each class of a Java program (top-level, inner or anonymous) a separate class file is compiled.

Definition 11 (Java Class File). *In a Java Bytecode program a* class file *is a tuple $cf = (name, isInterface, modifiers, super, implements, fields, methods)$, where $name \in \mathit{Class} \cup \mathit{Interface}$ is the unique identifier of the class or interface, $isInterface \in \{true, false\}$ is true iff the file defines an interface, $modifiers \subseteq \mathit{Modifier}$ are the modifiers (*static*,* private*,* public*, ...), $super \in \mathit{Class}$ is the super class and $implements \subseteq \mathit{Interface}$ are the implemented interfaces, fields is a mapping $fields : \mathit{Field} \to \mathcal{P}(\mathit{Modifier}) \times \mathit{Type}$, with $\mathit{Type} = \mathit{Class} \cup \mathit{Interface} \cup \{\mathit{boolean}\}$ defining the fields of the class, their modifiers and types, and the mapping $methods : \mathit{MSig} \to \mathit{MDec}$ defines the methods of the class, where MSig is the set of method signatures $\mathit{MSig} = \mathit{Meth} \times \mathit{Type}^\star$ with Meth the set of method identifier, and MDec the set of method declarations as defined below.*

$\mathit{ClassFile}$ denotes the set of all class files of a given Java Bytecode program.

The sets Class and $\mathit{Interface}$ contain the identifiers of the classes/interfaces, distinguished by $isInterface$: $\mathsf{Class} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge \neg isInterface_{cf}\}$ and $\mathsf{Interface} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge isInterface_{cf}\}$. We define the sets of available fields $\mathsf{Class/Field} = \{name_{cf}.field \mid cf \in \mathit{ClassFile} \wedge field \in \mathsf{Field}_{cf}\}$, which are uniquely identified by the combination of class and field name, and the set of methods $\mathsf{Class/MSig} = \{name_{cf}.mSig \mid cf \in \mathit{ClassFile} \wedge mSig \in \mathsf{MSig}_{cf}\}$, uniquely defined by class names and method signatures.

Definition 12 (Method Declaration). *A* method declaration *is a tuple* $md = (modifiers, returnType, code, excs, maxReg, maxOpd)$*, with* modifiers $\subseteq$ $\mathsf{Modifier}$ *and* returnType $\in$ $\mathsf{Type} \cup \{\mathsf{void}\}$*,* code $\in$ $\mathsf{Instruction}^\star$ *being a sequence of instruction (*$\mathsf{Instruction}$ *is defined in 3.3); excs belongs to a set of exceptions (not considered in this paper) and* maxOpd $\in \mathbb{N}$ *being the maximum size of the operand stack, while* maxReg $\in \mathbb{N}$ *being the highest register used by the method.*

The function *method* applied as $method(c, mSig)$ returns $c'.mSig \in \mathsf{Class/MSig}$ where c' is the class in which the corresponding method is declared, i.e. returns the method of the given signature inherited from c'. Note that *method* is determined by the content of $\mathsf{ClassFile}$.

State of a JVM. Heap and method stack determine the state of a JVM.

The heap formally is a function $heap : Ref \to Heap$. $Heap = \mathsf{Class} \times \mathsf{Class/Field} \to \mathsf{Val}$ [16], is a set of objects defined by the type and evaluation of references.

The method stack consists of frames $stack \in \mathsf{Frame}^*$ with $(\, pc, reg, opd, method\,) \in \mathsf{Frame}$, where $method \in \mathsf{Class/MSig}$ is the method, $pc \in \mathbb{N}$ is the program counter defining the current position in the method, $reg : \mathbb{N} \to \mathsf{Val}$ defines the values of the registers, which are used by the JVM to store the local variable information, and $opd \in \mathsf{Val}^\star$ is the operand used to store intermediate results of calculations. The top frame of the stack defines the state of the active method.

3.2 Modelling JVM States by Heap Configurations

Our goal is to model (abstract) states of the JVM by HCs. Starting with a very basic model, representing only the heap and restricting the programs to classes and member variables. In this section we extend the representation step by step.

The Basic Model. We consider programs in the most basic case where each class file defines a class and all fields are member variables, i.e. $\mathsf{Interface} = \emptyset$, and no static fields. States are represented by HCs over the heap alphabet Σ with $\mathbb{T} = \mathsf{Class}$, $\mathbb{F} = \mathsf{Class/Field}$ and $types(c.f) = c_{\triangledown} t_{\blacktriangle}$, where $t = fields(c.f)[2]$.

Interfaces and *null*. Java differentiates between classes and interfaces. Interfaces cannot be instantiated, i.e. heap objects can not be of an interface type.

We extend the heap alphabet Σ_N to $\mathbb{T} = \mathsf{Class} \cup \mathsf{Interface} \cup \{\bot\}$, where $\bot$ represents *null*. For all $t \in \mathbb{T}$ we let $\bot \preceq t$, i.e. $\bot$ is the least element. Elements of Class and $\mathsf{Interface}$ are ordered corresponding to the subtype relation.

We model *null* as an external $\blacktriangle$-node, i.e. a node that can be referenced but is not part of the heap. We consider HCs $(V, E, lab, type, att, \mathtt{null}_{\blacktriangle})$ over Σ_N, where $\bot \in \mathbb{T}_{\Sigma_N}$ and $\{v \in V \mid type(v) = \bot\} = \{\mathtt{null}\}$, i.e. the null reference is unique. This is important for comparisons. As $\bot$ is the least element every pointer can point to $\mathtt{null}$.

Example 7. Fig. 6 represents a binary tree. Every node is of type Tree denoted by T: **class** Tree{Tree left , Tree right ;}. Leafs are Tree objects pointing to *null*. There are only two different types: Tree and $\perp$ with $\perp \preceq$ *Tree*. The external node ($ext = \texttt{null}_\blacktriangle$) is of type $\perp$ to realise pointers to *null*.

Static Variables. Beside member variables (fields) there are also static variables. These variables are not linked to an object and are accessible from any context. We make them accessible through an external node `static` of (a new) type $\mathsf{static} \in \mathbb{T}$. For every static field $f \in \mathsf{Class/Field}$ we define $\mathit{types}(c.f) = \mathsf{static}_\triangledown t_\blacktriangle$ ($t = \mathit{fields}(c.f)[2]$), i.e. type static has one outgoing pointer for each static variable. We extend the sequence of external nodes by a node `static` and get $(V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \texttt{null}_\blacktriangle\mathsf{static}_\triangledown)$. We require `static` to be the sole static-node, i.e. $\{v \in V \mid \mathit{type}(v) = \mathsf{static}\} = \{\texttt{static}\}$. The node `static` is an $\triangledown$-node.

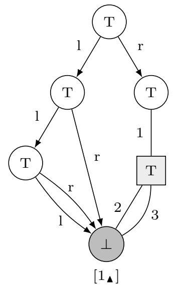


Fig. 6. A Tree

Literals and Boolean Values. Literals (constants) are a special case of static variables, whose values are explicitly given within a Java program. As we do not consider general data values the only possible literals are the boolean values false and true, represented in Java Bytecode as integer values *zero* and *one*. In order to model boolean values we add two nodes of the (newly introduced) type $\mathsf{int} \in \mathbb{T}$ representing integer value zero and one, accessible through `static` by edges labeled *int(0)* and *int(1)*, i.e. $\mathit{types}(\mathit{int(0)}) = \mathit{types}(\mathit{int(1)}) = \mathsf{static}_\triangledown\mathsf{int}_\blacktriangle$.

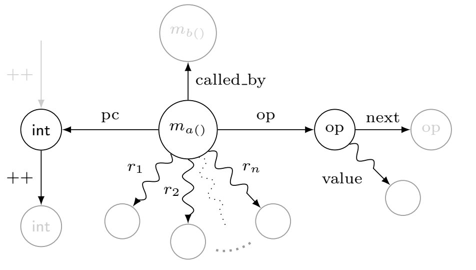
Complete Heap Representation. Given a Java Bytecode program as a set of class files *ClassFile*, we use HCs over the alphabet Σ with $\mathbb{T} = \mathsf{Class} \cup \mathsf{Interface} \cup \{\mathsf{static}, \mathsf{int}, \perp\}$, $\mathbb{F} = \mathsf{Class/Field} \cup \{\mathit{int(0)}, \mathit{int(1)}\}$ and

$$\begin{aligned}\mathit{types} = {} & \{c.f \mapsto c_\triangledown t_\blacktriangle \mid c.f \in \mathsf{Class/Field}_o \wedge \mathit{fields}(c.f)[2] = t\}\\ & \cup \{c.f \mapsto \mathsf{static}_\triangledown t_\blacktriangle \mid c.f \in \mathsf{Class/Field}_s \wedge \mathit{fields}(c.f)[2] = t\}\\ & \cup \{\mathit{int(0)} \mapsto \mathsf{static}_\triangledown\mathsf{int}_\blacktriangle, \mathit{int(1)} \mapsto \mathsf{static}_\triangledown\mathsf{int}_\blacktriangle\}\end{aligned}$$

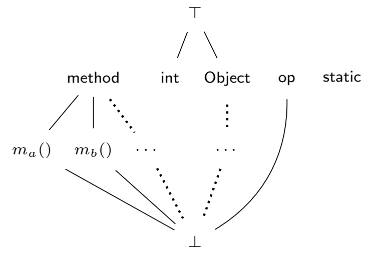
We use a HC of the form $H = (V, E, \mathit{lab}, \mathit{type}, \mathit{att}, \texttt{static}_\triangledown\texttt{null}_\blacktriangle)$, where none of the nodes is of an interface type ($\{v \in V \mid \mathit{type}(v) \in \mathsf{Interface}\} = \emptyset$), node `null` is the only node of type $\perp$ ($\{v \in V \mid \mathit{type}(v) = \perp\} = \{\texttt{null}\}$) and node `static` the only one of type static ($\{v \in V \mid \mathit{type}(v) = \mathsf{static}\} = \{\texttt{static}\}$). The only two int-nodes $\{v \in V \mid \mathit{type}(v) = \mathsf{int}\} = \{v_{\mathit{int}(0)}, v_{\mathit{int}(1)}\}$ are successors of the node `static`: $\exists e_0, e_1 \in \triangledown(\texttt{static}) : \mathit{lab}(e_i) = \mathit{int}(i) \wedge \mathit{att}(e_i)[2] = v_{\mathit{int}(i)}$.

Method Stack. So far we only considered the heap component. Now we model the method stack and its components within the same HC by representing each stack frame as a node of a special method type. This allows us to abstract the stack and handle recursive functions with unbounded method stack size. It is preferable to model both parts in one homogeneous representation as abstracting heap and stack independently would imply loosing the relation between the two.

We model each frame by one node. For each method $c.a \in \mathsf{Class/MSig}$ we define a proper type $m_{c.a}$, reflecting the *method* component of the frame. Each method type is a subtype of a general method type $\mathsf{method} \in \mathbb{T}$ (see Fig. 7(b)).



(a) Method nodes represent frames



(b) Extended type order

Fig. 7. Method nodes represent frames of the method stack.

For the program counter we add one int-node for each possible value, i.e. we add nodes $\{v_{int(i)} \mid i \in [0, max(\{|code(c.m)| \mid c.m \in \mathsf{Class/MSig}\})]\}$ and fields *int(i)* as pointers from static to int. Further we add the field ++ with $types(++) = \mathsf{int}_\triangledown \mathsf{int}_\blacktriangle$ representing the successor relation between int-nodes. The program counter is modelled as a pointer $method.pc \in \mathbb{F}$ to the corresponding int-node, i.e. $types(method.pc) = \mathsf{method}_\triangledown \mathsf{int}_\blacktriangle$. For the operand stack we add an op-type for stack elements with *next* and *value* successors, $types(op.next) = \mathsf{op}_\triangledown \mathsf{op}_\blacktriangle$ and $types(op.value) = \mathsf{op}_\triangledown \top_\blacktriangle$, where $\top \in \mathbb{T}$ with int and Object as subtypes (see Fig. 7(b)), i.e. *op.value* can reference int- and Object-nodes. We add a pointer to each method-node $op \in \mathbb{F}$ to the operand stack ($types(op) = \mathsf{method}_\triangledown \mathsf{op}_\blacktriangle$).

As registers offer random access we model each register i by a pointer r_i. The amount of registers depends on the method, therefore we define the r_i-pointer for each specification of the method-type. Given $c.m \in \mathsf{Class/MSig}$ we define for each $i \in [1, maxReg_{c.m}]$: $types(r_i) = \mathsf{c.m}_\triangledown \top_\blacktriangle$ (see Fig. 7(a)).

We model the method stack itself by an additional field $called\_by \in \mathbb{F}$ with $types(called\_by) = \mathsf{method}_\triangledown \mathsf{method}_\blacktriangle$ referencing the next node of the stack (where the least element in the stack points to `null`). The top of the stack is the active method. The corresponding node contains the currently modifiable information and therefore modelled as an external $\triangledown$-node. We get HCs of the following form: $(V, E, lab, type, att, \mathtt{method}_\triangledown \mathtt{static}_\triangledown \mathtt{null}_\blacktriangle)$.

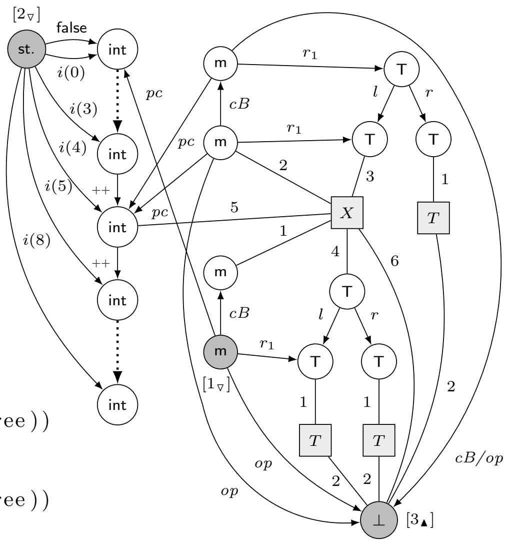
Example 8. In Fig. 8 a recursive tree traversal algorithm is given as Java (a) and Java Bytecode program (b) (details on Bytecode in Section 3.3). In Fig. 8(c) a state of the program from (b) is depicted. In this state the method *trav(Tree t)* is called. The program counter is still set to zero. The *trav* method was called various times. Three method calls are concrete, further are abstracted in the nonterminal edge X_l. Each method call was a *trav(t.left)* call as each program counter points to $i(5)$. Note that in X_l method and tree nodes are abstracted.

```
public class Tree{
  Tree left;
  Tree right;
  static void trav(Tree t){
    if(t != null){
      trav(t.left);
      trav(t.right);
} } }
```

(a) Java Class Definition

```
0  Load(Tree, 0)
1  Cond(ifNull, 8)
2  Load(Tree, 0)
3  GetField(Tree, Tree.left)
4  InvokeStatic(void, Tree.trav(Tree))
5  Load(Tree, 0)
6  GetField(Tree, Tree.right)
7  InvokeStatic(void, Tree.trav(Tree))
8  Return(void)
```

(b) Java Bytecode: trav(Tree t)



(c) State of the abstract JVM

Fig. 8. Recursive Tree Traversal.

Abstract JVM States So far we considered HCs over the concrete alphabet Σ, thus concrete HCs. To represent an abstract state we extend the alphabet by a set of nonterminals N as defined before (see Ex. 3), where we restrict *types* over N to *types* : $N \to (\mathsf{Class}_{\triangledown\blacktriangle} \cup \mathsf{Class/MSig}_{\triangledown\blacktriangle} \cup \mathsf{Interface}_{\blacktriangle} \cup \{\bot_\blacktriangle, \top_\blacktriangle\})^\star$, i.e. only class- and method-nodes can be connected to $\triangledown$-tentacles. From this restriction it follows that $type(v) \in \mathsf{Interface} \cup \{\top, \bot\} \Rightarrow v_\blacktriangle \in [ext]$.

Given $H \in \mathrm{HC}_{\Sigma_N}$ we call a node $v \in V_H$ concrete if its successors are concrete ($\triangledown(v) \subseteq \mathbb{T}$), abstract otherwise ($\triangledown(v) \subseteq N$). Concrete and abstract parts coexist on a HC. A HC without abstract nodes is called concrete, otherwise abstract.

Concretisation. Concretisations are defined through the application of grammar rules, i.e. given $H, H' \in \mathrm{HC}_{\Sigma_N}$ H' is a concretisation of H iff $H \Rightarrow H'$. $L(H)$ is the set of all concrete HCs represented by the (abstract) HC H. Given a DSG in LGNF we can systematically concretise the abstracted successors of a node v by replacing the connected $\triangledown$-tentacle by the corresponding rules from the grammar. Correspondingly we define for $H \in \mathrm{HC}_{\Sigma_N}$ and abstract node $v \in V$:

$$conc_G(H, v) = \{H[e/R] \mid \triangledown_H(v) = \{e\} \wedge att[i](e) = v \wedge (lab(e) \to R) \in G_{(lab(e),i)}\}$$

and if v is a concrete node then $conc_G(H, v) = H$, thus *conc* has no effect. Note that for every $H \in \mathrm{HC}_{\Sigma_N}$ and $G \in \mathrm{DSG}_{\Sigma_N}$: $L_G(H) = L_G(conc_G(H, v))$

We call a mapping $m : \mathrm{HC}_{\Sigma_N} \to \mathrm{HC}_{\Sigma_N}$ a concrete modifier for $H \in \mathrm{HC}_{\Sigma_N}$ iff $L_G(m(H)) = m(L_G(H))$, i.e. the modification m is safe and most precise under the abstraction. As long as a modifier uses only the information of concrete nodes and their incident edges and preserves abstracted parts it is a concrete modifier.

Abstraction. Abstraction is defined through backward application of grammar rules, i.e. given $H, H' \in \mathrm{HC}_{\Sigma_N}$, H is an abstraction of H' iff $H \Rightarrow H'$. In practice abstraction is realised by the search of embeddings of rule graphs and followed by replacement of the embedding with the corresponding nonterminals.

Definition 13 (Embedding). *Given $I, H \in \mathrm{HC}_{\Sigma_N}$ an* embedding *of I in H consists of two mappings $emb : (V_I \to V_H, E_I \to E_H)$ with following properties:*

$$\begin{array}{lcll}
emb(v) & \neq & emb(v') & \forall v \neq v' \in V_I \setminus \{v \in V_I \mid v_\blacktriangle \in ext_I\} \\
emb(e) & \neq & emb(e') & \forall e \neq e' \in E_I \\
lab_I(e) & = & lab_H(emb(e)) & \forall e \in E_I \\
type_I(v) & \preceq & type_H(emb(v)) & \forall v \in \{v \in V_I \mid v_\blacktriangle \in ext_I\} \\
type_I(v) & = & type_H(emb(v)) & \forall v \in V_I \setminus [ext_I] \\
emb(att_I(e)) & = & att_H(emb(e)) & \forall e \in E_H \\
e \notin emb(E_I) & \Rightarrow & [att_I(e)] \cap emb(V_I) = \emptyset & \forall e \in E_H
\end{array}$$

Given $I, H \in \mathrm{HC}_{\Sigma_N}$ $Emb(I, H)$ denotes the set of all embeddings of I in H.

Given $G \in \mathrm{DSG}_{\Sigma_N}$, $I, H \in \mathrm{HC}_{\Sigma_N}$, $emb \in Emb(I, H)$ and $X \in N$ replacing I in H results in $replace(I, H, emb, X) = K$, with:

$$\begin{array}{llll}
V_K & = V_H \setminus emb(V_I \setminus [ext_I]) & E_K & = (E_H \setminus emb(E_I)) \uplus \{e\} \\
lab_K & = lab_H \restriction E_K \cup \{e \mapsto N\} & type_K & = type_H \restriction V_K \\
att_K & = att_H \restriction E_K \cup \{e \mapsto emb(ext_I)\} & ext_K & = ext_H
\end{array}$$

$abstr_G(H) = \{replace(R, H, emb, X) \mid X \to R \in G, emb \in Emb(R, H)\}$ are the HCs we get via one abstraction step ($H' \in abstr_G(H)$ *iff* $H' \Rightarrow_G H$), $abstr^\star_G(H)$ is the transitive closure ($H' \in abstr^\star_G(H)$ *iff* $H' \Rightarrow^\star_G H$). We denote the set $\{H' \in abstr^\star_G(H) \mid \forall X \to R \in G : Emb(R, H') = \emptyset\}$ of maximal abstracted HCs by $maxAbstr_G(H)$. Note that $maxAbstr_G(H)$ in general is not a singleton but finite. We call G *backward confluent* iff $maxAbstr_G(H)$ is a singleton for any $H \in \mathrm{HC}_{\Sigma_N}$.

3.3 Execution of Java Bytecode

The following abstract instructions cover the whole instruction set [16]:

Prim(PrimOp)	Dupx()	Pop()
Load(Type, RegNo)	Store(Type, RegNo)	Goto(LineNumber)
Cond(PrimOp, LineNumber)		
GetStatic(Type, Class/Field)	PutStatic(Type, Class/Field)	InvokeStatic(Type, Class/MSig)
Return(Type)		
New(Class)	Return(Type)	InstanceOf(Type)
GetField(Type, Class/Field)	PutField(Type, Class/Field)	Checkcast(Type)
InvokeSpecial(Type, Class/MSig)	InvokeVirtual(Type, Class/MSig)	
Athrow	Jsr(LineNumber)	Ret(RegNo)

We defined and implemented the transition rules for the above abstract instructions up to the grey ones (used for exception handling). The Type information in the instructions can be used to check for type safeness but is ignored by the JVM. As we do not consider general data values, there is a notable cutback of primary operations. The following are supported:

if_acmpeq	if_acmpne	if_icmpeq	if_icmpne
iconst_0	iconst_1	iand	ior

The primary *if*-operations, used by the Cond instruction, are realised by comparing the corresponding nodes referred by the stack, iconst_0 and iconst_1 push the corresponding int-nodes on the stack. iand and ior can be defined explicitly for the four possible inputs. Note that the set $\{0, 1\}$ is closed under both operations. We present a selection of instructions and their transition rules. Most of the instructions are realised as concrete modifiers expecting external nodes and the actual operand stack to be concrete.

Graph Manipulations We define some basic actions, shared by several Bytecode instruction (as push, pop, etc.) by means of direct graph manipulations.

$\mathit{new}(H, t) = (H', v_{new})$ adds a new node to a HC. Given $H \in \mathrm{HC}_{\Sigma_N}$ and $t \in \mathbb{T}$ we get $H' = (V_H \uplus \{v_{new}\}, E_H, \mathit{lab}_H, \mathit{type}_H \cup \{v_{new} \mapsto t\}, \mathit{att}_H, \mathit{ext}_H)$.

suc (H, v, f) returns for $H \in \mathrm{HC}_{\Sigma_N}$, $v \in V_H$ and $f \in \mathbb{F}$ the f-successor of v: $\mathit{suc}(H, v, f) = v'$, if $\{v'\} = \mathit{att}_H(\{e \in \nabla_H(v) \mid \mathit{lab}_H(e) = f\})[2]$.

setSuc (H, v, f, v') alters for $H \in \mathrm{HC}_{\Sigma_N}$, $v, v' \in V_H$ and $f \in \mathbb{F}$ the edge representing the f-pointer of v: $\mathit{setSuc}(H, v, f, v') = (V_H, E_H, \mathit{lab}_H, \mathit{type}_h, \mathit{att}_h[e \mapsto v\ v'], \mathit{ext}_h)$, where $\{e\} = \{e \in \nabla_H(v) \mid \mathit{lab}(e) = f\}$.

$\mathit{pushOp}(H, v)$ pushes a reference to $v \in V_H$ onto the operand stack by adding a node of type op $(H_{new}, v_{op}) = \mathit{new}(H, \mathsf{op})$ and connecting the *next*-edge to the operand stack $v_{top} = \mathit{suc}(H, \mathit{ext}_H[1], \mathit{op})$ and the *value*-edge to node v: $H' = \mathit{setSuc}(\mathit{setSuc}(H_{new}, v_{op}, \mathit{value}, v), v_{op}, \mathit{next}, v_{top})$. The reference to the operand stack is updated for the top method: $\mathit{pushOp}(H, v) = \mathit{setSuc}(H', \mathit{ext}'_H[1], \mathit{op}, v_{op})$.

$\mathit{popOp}(H) = (H', v)$ pops the top element $v_{top} = \mathit{suc}(H, \mathit{ext}_H[1], \mathit{op})$ by altering the *op*-edge to the next operand $H' = \mathit{setSuc}(H, \mathit{ext}'_H[1], \mathit{op}, \mathit{suc}(H, v_{top}, \mathit{op}))$. The value of the removed stack element $v = \mathit{suc}(H, v_{top}, \mathit{value})$ is returned.

$\mathit{peekOp}(H, n) = \mathit{suc}^n(H, \mathit{ext}_H[1], \mathit{op})$ returns for $H \in \mathrm{HC}_{\Sigma_N}, n \in \mathbb{N}$ the n_{th} element of the operand stack, where suc^n is defined recursively as $\mathit{suc}^n(H, v, f) = \mathit{suc}^{n-1}(H, \mathit{suc}(H, v, f), f)$ for $n > 0$ and $\mathit{suc}^0(H, v, f) = H$.

$\mathit{incPc}(H)$ increments the program counter $v_{pc} = \mathit{suc}(H, \mathit{ext}_H[1], \mathit{pc})$, by altering it to the successor node: $\mathit{incPc}(H) = \mathit{setSuc}(H, \mathit{ext}_H[1], \mathit{pc}, \mathit{suc}(H, v_{pc}, {+}{+}))$.

$\mathit{inst}(H)$ returns the current instruction: $\mathit{inst}(H) = c[\mathit{pc}]$, where $c = \mathsf{code}(\mathit{type}_H(\mathit{ext}_H[1]))$ and $\mathit{pc} = \mathit{intValue}(\mathit{suc}(H, \mathtt{method}_H, \mathit{pc}))$.

Transition Rules In our tool (see Sec. 4) we implemented transition rules for all of the instructions given at the beginning of this subsection. Here we exemplarily present the transition rules of some of the instructions. Further rules can be found in the extended version [9]:

Load(RegNo) reads a reference from register RegNo and pushes it to the operand stack. We determine the node corresponding to the value of the register and push it to the stack.

$$\frac{\mathit{inst}(H) = \mathsf{Load(t,i)}}{H \to \mathit{incPc}(\mathit{pushOp}(H, \mathit{suc}(H, \mathtt{method}_H, r_i)))}$$

GetStatic(Class/Field) reads a static variable and pushes the result to the operand stack.

$$\frac{\mathit{inst}(H) = \mathsf{GetStatic(c.f)}}{H \to \mathit{incPc}(\ \mathit{pushOp}(H, \mathit{suc}(H, \mathit{ext}_H[2], c.f))}$$

PutField(Class/Field) writes a value to a field of an object. As the node that represents the object could be abstract we concretise it before we set the field. The update could be destructive and could yield garbage. Therefore we perform a garbage collection on the result and afterwards try to abstract. Note that this instruction is not deterministic.

$$\frac{\mathit{inst}(H) = \mathsf{PutField(f)} \qquad \mathit{popOp}(H) = (H', v') \qquad \mathit{popOp}(H') = (H'', v'')}{H \to \mathit{maxAbstr}(\mathrm{gc}(\ \mathit{incPc}(\ \mathit{setSuc}(K, v'', f, v)))), K \in \mathit{conc}(H'', v'')}$$

InvokeVirtual(Class/MSig) is the call of an object method. We add a new method node and set the registers to the parameters given in msig $= \mathit{Name}(p)$, with $p \in \mathsf{Type}^\star$. This instruction uses information from the static environment cEnv of the program.

$$\frac{\mathit{inst}(H) = \mathsf{InvokeVirtual(c.msig)} \qquad \mathit{popOp}(H) = (H', v)}{H \to \mathit{call}(H', \mathit{method}_{\mathit{cEnv}}(\mathit{type}(v), \mathit{c.msig})))}$$

where $\mathit{call}(H, c.m(p)) = \mathit{setSuc}(K, \mathit{op}, \mathit{peekOp}(H, |p|+1))$ with:

$$\begin{aligned}
V_K &= V_H \uplus \{v_m\} \\
E_K &= E_H \uplus \{e_{\mathit{calledBy}}, e_{\mathit{op}}, e_{\mathit{pc}}\} \uplus \{e_{r_i} \mid i \in [1, \mathit{maxReg}_{c.m(p)}]\} \\
\mathit{lab}_K &= \mathit{lab}_H \uplus \{e_x \mapsto x \mid e_x \in E_K \setminus E_H\} \\
\mathit{type}_K &= \mathit{type}_H \cup \{v_m \mapsto c.m(p)\} \\
\mathit{att}_K &= \mathit{att}_K \\
&\quad \cup \{e_{\mathit{calledBy}} \mapsto \mathit{ext}_H[1], e_{\mathit{op}} \mapsto \mathit{ext}_H[3], e_{\mathit{pc}} \mapsto \mathit{suc}(H, \mathit{ext}_H[2], \mathit{int}(0))\} \\
&\quad \cup \{e_{r_i} \mapsto \mathit{peekOp}(i) \mid i \in [1, |p|]\} \\
&\quad \cup \{e_{r_i} \mapsto \mathit{ext}_H[3] \mid i \in [|p|+1, \mathit{maxReg}_{c.m(p)}]\} \\
\mathit{ext}_K &= v_{m\triangledown}\ \mathit{ext}_H[2]\ \mathit{ext}_H[3]
\end{aligned}$$

4 Experimental Results

We implemented the above concepts in a prototype tool which, for a Java Bytecode program, a hyperedege replacement grammar, and a start heap generates the abstracted state space. The following table gives some experimental results:

Method	Rules	States	Parsing	Generation
ReverseList (singly linked)	3	113	0:010 s	0:006 s
TraverseTree (recursive)	49	574	0:472 s	0:264 s
Lindstrom (no marking)	14	4,297	0:245 s	0:198 s
Lindstrom (single marking)	14	224,113	0:245 s	2:360 s
Lindstrom (extended marking)	14	937,510	0:245 s	9:074 s

TraverseTree is the Java program from Fig. 8. The *Lindstrom Traversal Algorithm* [10] traverses a tree with constant additional memory by altering the pointers of the elements. This algorithm (Fig. 9) was analysed by us before [7].

The column *rules* gives the size the provided grammar, *states* the size of the generated abstracted state space, *parsing* the time for parsing Bytecode, grammar, and start heap and *generation* the time needed to generate the state space. The examples where calculated on a 2 GHz Intel Core i7 Laptop.

In none of the given examples, null pointer dereferencing occurs. To describe complex properties we use LTL with pointer equations (e.g. $x.l = y$) and a flag *terminal* as atomic propositions. For *Lindstrom* we proved *termination*, *completeness* (each node is visited) and *correctness* (at the end the input tree is not altered) [7].
We need quantification over objects as in [13] to verify these properties. We realise quantification by adding markings, i.e. static variables not visible to the program. Markings are determined by exhaustive object exploration where objects are concretised and abstracted as needed. For *Lindstrom* we get 41 different abstract markings, in each of them one object is marked as x. We can prove that for each of these the LTL formulas $\mathbf{FG}(cur \neq x)$ and $\neg(cur \neq x \,\mathbf{U}\, terminal)$ hold. The former states that the variable *cur* points only finitely many times to the marked object and as this could be any object (or *null*) the calculation has to terminate eventually. The latter states that before terminating, *cur* points at least once to the marked node, thus the algorithm is complete. For correctness we also mark the left and right successor of x by x_l and x_r respectively and check that the successors are the same at the end of the traversal: $(x = root \rightarrow \mathbf{G}(x = root)) \wedge (\mathbf{G}(terminal \rightarrow (x_l = x.l \wedge x_r = x.r)))$. Markings increase the state space (see row single and extended marking). The above only works for quantification over objects in the start heap. The termination check is only correct if no objects are generated at runtime.

```
static void trav(Tree root){
  if(root == null) return;
  Tree sen = new Tree();
  Tree prev = sen;
  Tree cur  = root;
  while(cur != sen){
    Tree next = cur.left;
    cur.left  = cur.right;
    cur.right = prev;
    prev = cur;
    cur  = next;
    if (cur == null){
      cur  = prev;
      prev = null;
} } }
```

Fig. 9. Lindstrom Traversal

5 Related Work

The basic idea of using hyperedge replacement grammars for abstraction of heap structures was proposed in [7, 8]. However it was not suitable for the analysis of Java Bytecode as typed objects were not reflected. Various other techniques for the analysis and verification of heap manipulating programs where supposed. The most popular ones are *shape analysis via three-valued logic* [15] and *separation logic* [14]. The latter is an extension of Hoare logic and uses recursive predicates to describe the shape of heaps. There is a one-to-one correspondence between recursive predicates and nonterminals of our representation as stated by [5]. Separation logic is classically used in Hoare Triple style verification where decidability of entailment is essential. Therefore entailment, not decidable in general, has to be proven decidable for any recursive predicate. There are decidable logics for lists and trees [1, 2]. In [12] separation logic is extended for Bytecode by adding type information. There are several separation logic tools as SpaceInvador [17], for linear data structures, or Smallfoot [2], for (doubly) linked lists and trees. The advantage of tools based on deductive methods is scalability [17]. However, their applicability is restricted to predefined data structures.

Another abstraction technique is the *shape analysis via three-valued logic* [14]. Nodes are summarised by properties expressed as predicates in a three-valued logic. Predicates are typically shape properties such as reachability, cycle membership, etc. Most of these are implicitly given by our representation and are considered during state space exploration. Given an abstract state the satisfied predicates should be extractable [4]. Whereas in shape analysis all nodes reflecting the same set of predicates are summarised, in our approach nodes are summarised if they form a well defined substructure, resulting in additional structural information. Unfortunately, structures expressible by HRG and the commonly used fragment of separation logic are restricted to those with bounded tree width [6], e.g. the set of all graphs is not expressible by HRGs.

In [11] TVLA is used to verify the *Lindstrom Algorithm.* The given proof depends on 24 predicates encoding deep knowledge of the algorithm, resulting in a less automatic proof than the one provided in Sect. 4.To reduce the number of predicates and to keep the example manageable, the input code is modified in [11] so that the heap is always a tree. This is not necessary in our approach as it is robust against local violations of the data structure. Our abstraction results in slightly larger state space but also in shorter running time (TVLA: 183,564 states in over 36 minutes on a 2.4GHz Core 2 Duo with 4GB of RAM [3]).

6 Conclusion

We introduced labeled and typed hypergraphs and corresponding HRGs, where modes are associated with a type from a type hierarchy. We showed how they can be used to model abstracted JVM states and how to compute an abstract state space for Java Bytecode programs. Experimental results attest that the approach has a practical value. In the future we will consider automatic inference of DSGs during state space generation as well as extended verification techniques.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A Decidable Fragment of Separation Logic. In Lodaya, K., Mahajan, M., eds.: FSTTCS. Volume 3328 of LNCS, Springer (2004) 97–109
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS, Springer (2005) 115–137
3. Bogudlov, I., Lev-Ami, T., Reps, T.W., Sagiv, M.: Revamping TVLA: Making Parametric Shape Analysis Competitive. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS, Springer (2007) 221–225
4. Courcelle, B.: The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. In Rozenberg, G., ed.: Handbook of Graph Grammars, World Scientific (1997) 313–400
5. Dodds, M., Plump, D.: From Hyperedge Replacement to Separation Logic and Back. ECEASST **16** (2008)
6. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations. World Scientific Publishing (1997) 95–162
7. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS **266** (2010) 93 – 107
8. Jansen, C., Heinen, J., Katoen, J.P., Noll, T.: A Local Greibach Normal Form for Hyperedge Replacement Grammars. In Dediu, A.H., Inenaga, S., Martín-Vide, C., eds.: LATA. Volume 6638 of LNCS, Springer (2011) 323–335
9. Jonathan Heinen, H.B., Jansen, C.: Juggrnaut - An Abstract JVM. Technical Report AIB-2011-21, RWTH Aachen (2011)
10. Lindstrom, G.: Scanning List Structures Without Stacks or Tag Bits. Inf. Process. Lett. **2**(2) (1973) 47–51
11. Loginov, A., Reps, T.W., Sagiv, M.: Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal algorithm. In Yi, K., ed.: SAS. Volume 4134 of LNCS, Springer (2006) 261–279
12. Luo, C., He, G., Qin, S.: A Heap Model for Java Bytecode to Support Separation Logic. In: APSEC, IEEE (2008) 127–134
13. Rensink, A.: Model Checking Quantified Computation Tree Logic. In Baier, C., Hermanns, H., eds.: CONCUR. Volume 4137 of LNCS, Springer (2006) 110–125
14. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS, IEEE Computer Society (2002) 55–74
15. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298
16. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer (2001)
17. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In Gupta, A., Malik, S., eds.: CAV. Volume 5123 of LNCS, Springer (2008) 385–398