Let this Graph be your Witness!

An Attestor for Verifying Java Pointer Programs

Hannah Arndt*, Christina Jansen, Joost-Pieter Katoen[0000−0002−6143−1926], Christoph Matheja*[0000−0001−9151−0441], and Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University, Germany

Abstract. We present a graph-based tool for analysing Java programs operating on dynamic data structures. It involves the generation of an abstract state space employing a user-defined graph grammar. LTL model checking is then applied to this state space, supporting both structural and functional correctness properties. The analysis is fully automated, procedure-modular, and provides informative visual feedback including counterexamples in the case of property violations.

1 Introduction

Pointers constitute an essential concept in modern programming languages, and are used for implementing dynamic data structures like lists, trees etc. However, many software bugs can be traced back to the erroneous use of pointers by e.g. dereferencing null pointers or accidentally pointing to wrong parts of the heap. Due to the resulting unbounded state spaces, pointer errors are hard to detect. Automated tool support for validation of pointer programs that provides meaningful debugging information in case of violations is therefore highly desirable.

Attestor is a verification tool that attempts to achieve both of these goals. To this aim, it first constructs an abstract state space of the input program by means of symbolic execution. Each state depicts both links between heap objects and values of program variables using a graph representation. Abstraction is performed on state level by means of graph grammars. They specify the data structures maintained by the program, and describe how to summarise substructures of the heap in order to obtain a finite representation. After labelling each state with propositions that provide information about structural properties such as reachability or heap shapes, the actual verification task is performed in a second step. To this aim, the abstract state space is checked against a user-defined LTL specification. In case of violations, a counterexample is provided.

In summary, Attestor's main features can be characterized as follows:

- It employs context-free graph grammars as a formal underpinning for defining heap abstractions. These grammars enable local heap concretisation and thus naturally provide implicit abstract semantics.

* supported by Deutsche Forschungsgemeinschaft (DFG) Grant NO 401/2-1.

- The full instruction set of Java Bytecode is handled. Program actions that are outside the scope of our analysis, such as arithmetic operations or Boolean tests on payload data, are handled by (safe) over-approximation.
- Specifications are given by linear-time temporal logic (LTL) formulae which support a rich set of program properties, ranging from memory safety over shape, reachability or balancedness to properties such as full traversal or preservation of the exact heap structure.
- Except for expecting a graph grammar that specifies the data structures handled by a program, the analysis is fully automated. In particular, no program annotations are required.
- Modular reasoning is supported in the form of contracts that summarise the effect of executing a (recursive) procedure. These contracts can be automatically derived or manually specified.
- Valuable feedback is provided through a comprehensive report including (minimal) non-spurious counterexamples in case of property violations.
- The tool's functionality is made accessible through the command line as well as a graphical user and an application programming interface.

Availability. ATTESTOR's source code, benchmarks, and documentation are available online at https://moves-rwth.github.io/attestor.

2 The Attestor Tool

ATTESTOR is implemented in Java and consists of about 20.000 LOC (excluding comments and tests). An architectural overview is depicted in Fig. 1. It shows the tool inputs (left), its outputs (right), the ATTESTOR backend with its processing phases (middle), the ATTESTOR frontend (below) as well as the API connecting back- and frontend. These elements are discussed in detail below.

2.1 Input

As shown in Fig. 1 (left), a verification task is given by four inputs. First, the program to be analysed. Here, Java as well as Java Bytecode programs with possibly recursive procedures are supported, where the former is translated to the latter prior to the analysis. Second, the specification has to be given by a set of LTL formulae enriched with heap-specific propositions. See Sect. 3 for a representative list of exemplary specifications.

As a third input, ATTESTOR expects the declaration of the graph grammar that guides the abstraction. In order to obtain a finite abstract state space, this grammar is supposed to cover the data structures emerging during program execution. The user may choose from a set of grammar definitions for standard data structures such as singly- and doubly-linked lists and binary trees, the manual specification in a JSON-style graph format and combinations thereof.

Fourth, additional options can be given that e.g. define the initial heap configuration(s) (in JSON-style graph format), that control the granularity of abstraction and the garbage collection behaviour, or that allow to re-use results of previous analyses in the form of procedure contracts [11,13].

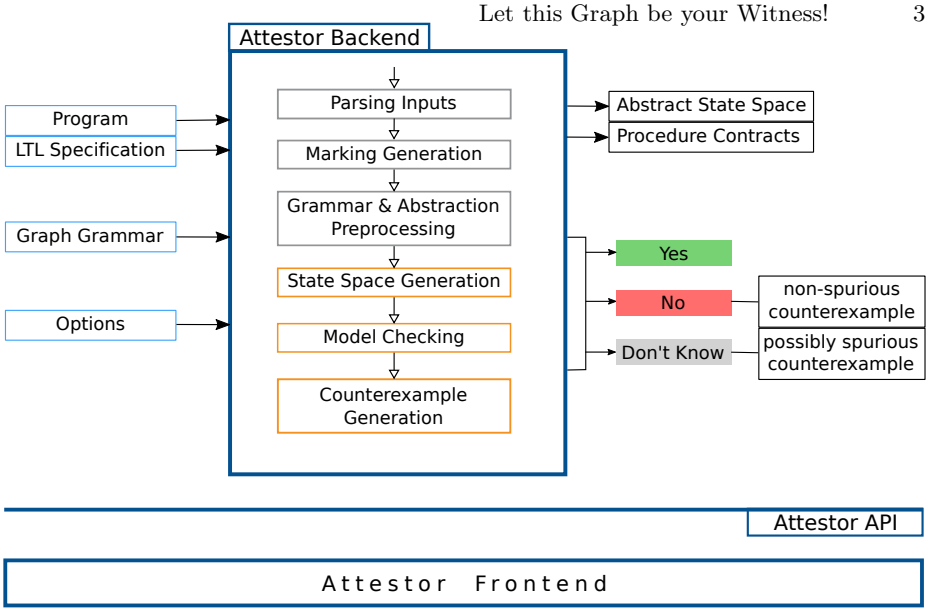

Fig. 1. The ATTESTOR Tool

2.2 Phases

ATTESTOR proceeds in six main phases, see Fig. 1 (middle). In the first and third phase, all inputs are parsed and preprocessed. The input program is read and transformed to Bytecode (if necessary), the input graphs (initial configuration, procedure contracts, and graph grammar), LTL formulae and further options are read.

Depending on the provided LTL formulae, additional markings are inserted into the initial heap (see [8] for details) in the second phase. They are used to track identities of objects during program execution, which is later required to validate visit and neighbourhood properties during the fifth phase.

In the next phase the actual program analysis is conducted. To this aim, ATTESTOR first constructs the abstract state space as described in Sect. 2.3 in detail. In the fifth phase we check whether the provided LTL specification holds on the state space resulting from the preceding step. We use an off-the-shelf tableau-based LTL model checking algorithm [2].

If desired, during all phases results are forwarded to the API to make them accessible to the frontend or the user directly. We address this output in Sect. 2.4.

2.3 Abstract State Space Generation

The core module of ATTESTOR is the abstract state space generation. It employs an abstraction approach based on hyperedge replacement grammars, whose theoretical underpinnings are described in [9] in detail. It is centred around a graph-based representation of the heap that contains concrete parts side by side with placeholders representing a set of heap fragments of a certain shape. The state space generation loop as implemented in ATTESTOR is shown in Fig. 2.

Initially it is provided with the initial program state(s), that is, the program counter corresponding to the starting statement together with the initial heap configuration(s). From these, ATTESTOR picks a state at random and applies the abstract semantics of the next statement: First, the heap configuration is locally concretised ensuring that all heap parts required for the statement to execute are accessible. This is enabled by applying rules of the input graph grammar in forward direction, which can entail branching in the state space. The resulting configurations are then manipulated according to the concrete semantics of the statement. At this stage, ATTESTOR automatically detects possible null pointer dereferencing operations as a byproduct of the state space generation. In a subsequent rectification step, the heap configuration is cleared from e.g. dead variables and garbage (if desired). Consequently, memory leaks are detected immediately. The rectified configuration is then abstracted with respect to the data structures specified by means of the input graph grammar. Complementary to concretisation, this is realised by applying grammar rules in backward direction, which involves a check for embeddings of right-hand sides. A particular strength of our approach is its robustness against local violations of data structures, as it simply leaves the corresponding heap parts concrete. Finalising the abstract execution step, the resulting state is labelled with the atomic propositions it satisfies. This check is efficiently implemented by means of heap automata (see [12,15] for details). By performing a subsumption check on the state level, ATTESTOR detects whether the newly generated state is already covered by a more abstract one that has been visited before. If not, it adds the resulting state to the state space and starts over by picking a new state. Otherwise, it checks whether further states have to be processed or whether a fixpoint in the state space generation is reached. In the latter case, this phase is terminated.

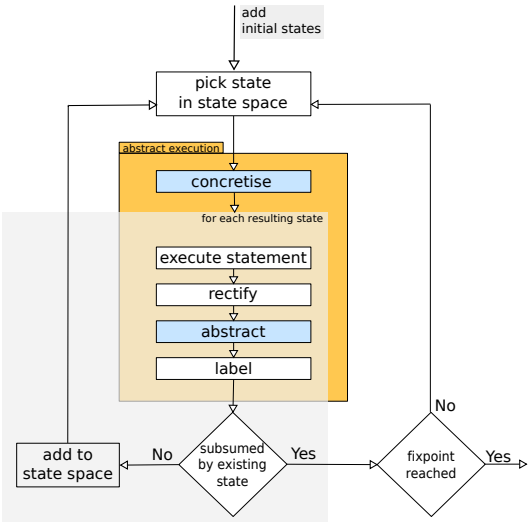

Fig. 2. State Space Generation

2.4 Output

As shown in Fig. 1 (right), we obtain three main outputs once the analysis is completed: the computed abstract state space, the derived procedure contracts, and the model checking results. For each LTL formula in the specification, results comprise the possible answers "formula satisfied", "formula (definitely) not satisfied", or "formula possibly not satisfied". In case of the latter two, ATTESTOR

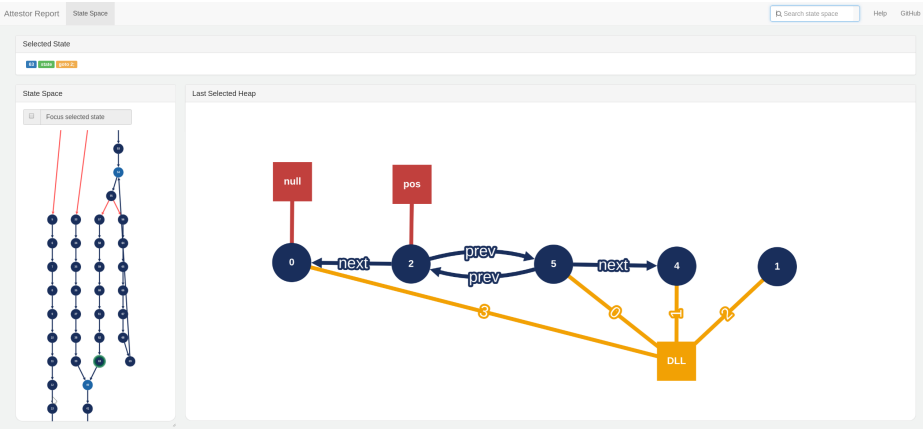

Fig. 3. Screenshot of Attestor's frontend for state space exploration.

additionally produces a counterexample, i.e. an abstract trace that violates the formula. If Attestor was able to verify the non-spuriousness of this counterexample (second case), we are additionally given a concrete initial heap that is accountable for the violation and that can be used as a test case for debugging.

Besides the main outputs, Attestor provides general information about the current analysis. These include log messages such as warnings and errors, but also details about settings and runtimes of the analyses. The API provides the interface to retrieve Attestor's outputs as JSON-formatted data.

2.5 Frontend

Attestor features a graphical frontend that visualises inputs as well as results of all benchmark runs. The frontend communicates with Attestor's backend via the API only. It especially can be used to display and navigate through the generated abstract state space and counterexample traces.

A screenshot of the frontend for state space exploration is found in Fig. 3. The left panel is an excerpt of the state space. The right panel depicts the currently selected state, where red boxes correspond to variables and constants, circles correspond to allocated objects/locations, and yellow boxes correspond to nonterminals of the employed graph grammar, respectively. Arrows between two circles represent pointers. Further information about the selected state is provided in the topmost panel. Graphs are rendered using `cytoscape.js` [6].

3 Evaluation

Tool comparison While there exists a plethora of tools for analysing pointer programs, such as, amongst others, Forester [10], Groove [7], Infer [5], Hip/Sleek [17], Korat [16], Juggrnaut [9], and Tvla [3], these tools differ in multiple dimensions:

- *Input languages* range from C code (Forester, Infer, Hip/Sleek) over Java/Java Bytecode (Juggrnaut, Korat) to assembly code (Tvla) and graph programs (Groove).
- The *degree of automation* differs heavily: Tools like Forester and Infer only require source code. Others such as Hip/Sleek and Juggrnaut additionally expect general data structure specifications in the form of e.g. graph grammars or predicate definitions to guide the abstraction. Moreover, Tvla requires additional program-dependent instrumentation predicates.
- *Verifiable properties* typically cover memory safety. Korat is an exception, because it applies test case generation instead of verification. The tools Hip/Sleek, Tvla, Groove, and Juggrnaut are additionally capable of verifying data structure invariants, so-called shape properties. Furthermore, Hip/Sleek is able to reason about shape-numeric properties, e.g. lengths of lists, if a suitable specification is provided. While these properties are not supported by Tvla, it is possible to verify reachability properties. Moreover, Juggrnaut can reason about temporal properties such as verifying that finally every element of an input data structure has been accessed.

Benchmarks Due to the above mentioned diversity there is no publicly available and representative set of standardised benchmarks to compare the aforementioned tools [1]. We thus evaluated Attestor on a collection of challenging, pointer intensive algorithms compiled from the literature [3,4,10,14]. To assess our counterexample generation, we considered invalid specifications, e.g. that a reversed list is the same list as the input list. Furthermore, we injected faults into our examples by swapping and deleting statements.

Properties During state space generation, *memory safety (M)* is checked. Moreover, we consider five classes of properties that are verified using the built-in LTL model checker:

- The *shape property (S)* establishes that the heap is of a specific shape, e.g. a doubly-linked list or a balanced tree.
- The *reachability property (R)* checks whether some variable is reachable from another one via specific pointer fields.
- The *visit property (V)* verifies whether every element of the input is accessed by a specific variable.
- The *neighbourhood property (N)* checks whether the input data structure coincides with the output data structure upon termination.
- Finally, we consider other functional *correctness properties (C)*, e.g. the return value is not null.

Setup For performance evaluation, we conducted experiments on an Intel Core i7-7500U CPU @ 2.70GHz with the Java virtual machine (OpenJDK version 1.8.0_151) limited to its default setting of 2GB of RAM. All experiments were run using the Java benchmarking harness jmh. Our experimental results are shown in Table 1. Additionally, for comparison purpose we considered Java implementations of benchmarks that have been previously analysed for memory safety by Forester [10], see Table 2.

		No. states		State Space gen.		Verification		Total Runtime	
Benchmark	**Properties**	min	max	min	max	min	max	min	max
`SLL.traverse`	M,S,R,V,N,X	13	97	0.030	0.074	0.039	0.097	0.757	0.848
`SLL.reverse`	M,S,R,V,X	46	268	0.050	0.109	0.050	0.127	0.793	0.950
`SLL.reverse` (recursive)	M,S,V,N,X	40	823	0.038	0.100	0.044	0.117	0.720	0.933
`DLL.reverse`	M,S,R,V,N,X	70	1508	0.076	0.646	0.097	0.712	0.831	1.763
`DLL.findLast`	M,C,X	44	44	0.069	0.069	0.079	0.079	0.938	0.938
`SLL.findMiddle`	M,S,R,V,N,X	75	456	0.060	0.184	0.060	0.210	0.767	0.975
`Tree.traverse` (Lindstrom)	M,S,V,N	229	67941	0.119	8.901	0.119	16.52	0.845	17.36
`Tree.traverse` (recursive)	M,S	91	21738	0.075	1.714	0.074	1.765	0.849	2.894
`AVLTree.binarySearch`	M,S	192	192	0.117	0.172	0.118	0.192	0.917	1.039
`AVLTree.searchAndBack`	M,S,C	455	455	0.193	0.229	0.205	0.289	1.081	1.335
`AVLTree.searchAndSwap`	M,S,C	3855	4104	0.955	1.590	1.004	1.677	1.928	2.521
`AVLTree.leftMostInsert`	M,S	6120	6120	1.879	1.942	1.932	1.943	2.813	2.817
`AVLTree.insert`	M,S	10388	10388	3.378	3.676	3.378	3.802	4.284	4.720
`AVLTree.sllToAVLTree`	M,S,C	7166	7166	2.412	2.728	2.440	2.759	3.383	3.762

Table 1. The experimental results. All runtimes are in seconds. Verification time includes state space generation. `SLL` (`DLL`) means singly-linked (doubly-linked) list.

Discussion The results show that both memory safety (M) and shape (S) are efficiently processed, with regard to both state space size and runtime. This is not surprising as these properties are directly handled by the state space generation engine. The most challenging tasks are the visit (V) and neighbourhood (N) properties as they require to track objects across program executions by means of markings. The latter have a similar impact as pointer variables: increasing their number impedes abstraction as larger parts of the heap have to be kept concrete. This effect can be observed for the Lindstrom tree traversal procedure where adding one marking (V) and three markings (N) both increase the verification effort by an order of magnitude.

Benchmark	**No. states**	**Verification**
`SLL.bubblesort`	287	0.134
`SLL.deleteElement`	152	0.096
`SLLHeadPtr` (traverse)	111	0.095
`SLL.insertsort`	369	0.147
`ListOfCyclicLists`	313	0.153
`DLL.insert`	379	0.207
`DLL.insertsort1`	4302	1.467
`DLL.insertsort2`	1332	0.514
`DLL.buildAndReverse`	277	0.164
`CyclicDLL` (traverse)	104	0.108
`Tree.construct`	44	0.062
`Tree.constructAndDSW`	1334	0.365
`SkipList.insert`	302	0.160
`SkipList.build`	330	0.173

Table 2. FORESTER benchmarks (memory safety only). Verification times are in seconds.

References

1. Abdulla, P.A., Gadducci, F., König, B., Vafeiadis, V.: Verification of evolving graph structures (Dagstuhl Seminar 15451). Dagstuhl Reports **5**(11) (2016) 1–28
2. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL. In: LICS 1995, IEEE (1995) 388–397
3. Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.: Revamping TVLA: Making parametric shape analysis competitive. In: CAV 2007, Springer (2007) 221–225

4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: SAS 2006, Springer (2006) 52–70
5. Calcagno, C., Distefano, D.: Infer: An automatic program verifier for memory safety of C programs. In: NFM, Springer (2011) 459–465
6. Cytoscape Consortium: Cytoscape: graph theory / network library for analysis and visualisation. `http://js.cytoscape.org/`
7. Ghamarian, A.H., de Mol, M.J., Rensink, A., Zambon, E., Zimakova, M.V.: Modelling and analysis using GROOVE. Int. Journal on Software Tools for Technology Transfer **14** (2012) 15–40
8. Heinen, J.: Verifying Java Programs – A Graph Grammar Approach. PhD thesis, RWTH Aachen University, Germany (2015)
9. Heinen, J., Jansen, C., Katoen, J.P., Noll, T.: Verifying pointer programs using graph grammars. Science of Computer Programming **97** (2015) 157–162
10. Holík, L., Lengál, O., Rogalewicz, A., Simácek, J., Vojnar, T.: Fully automated shape analysis based on forest automata. CoRR **abs/1304.5806** (2013)
11. Jansen, C.: Static Analysis of Pointer Programs – Linking Graph Grammars and Separation Logic. PhD thesis, RWTH Aachen University, Germany (2017)
12. Jansen, C., Katelaan, J., Matheja, C., Noll, T., Zuleger, F.: Unified reasoning about robustness properties of symbolic-heap separation logic. In: ESOP, Springer (2017) 611–638
13. Jansen, C., Noll, T.: Generating abstract graph-based procedure summaries for pointer programs. In: ICGT, Springer (2014) 49–64
14. Loginov, A., Reps, T.W., Sagiv, M.: Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In: SAS 2006, Springer (2006) 261–279
15. Matheja, C., Jansen, C., Noll, T.: Tree-like grammars and separation logic. In: APLAS, Springer (2015) 90–108
16. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: ICSE, Springer (2007) 771–774
17. Nguyen, H.H., David, C., Qin, S., Chin, W.N.: Automated verification of shape and size properties via separation logic. In: VMCAI, Springer (2007) 251–266