Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog

René Haberland   Sergey Ivanovskiy
Department of Software Engineering and Computer Applications
Saint Petersburg Electrotechnical University "LETI"
Saint Petersburg, Russia

Abstract—**A Prolog-based framework for fully automated verification currently under development for heap-based object-oriented data is introduced. Dynamically allocated issues are discussed, recent approaches and criteria are analysed. The architecture and its components are introduced by example. Finally, propositions to further and related work are given.**

I. Introduction

The main interest of this work is dedicated to the correctness of a program according to its memory consumption behaviour. It may, however, also be extended to performance considerations based on results coming from the dynamic memory verification, particularly but not only during the alias analysis [17], [3] or during the garbage collection phase, for instance. A Prolog-based verifier is suggested.

The structure of this paper is to give a short overview on current approaches in section 2. Section 3 presents the languages being used as programming and specification languages, and introduces the overall architecture. The architecture is designed to be open. Currently the project is under progress, the final section gives an outlook on related and future work.

[9] discusses why the aliasing issue has still not been solved yet. [16] provides a more detailed mark on the issues related to aliasing issues in a commercial Unix-environment. Despite its age [16] and its partial closeness because of the commercial background, it is still often cited in very recent publications and numerous technical reports on the same topic, and in open-source projects, and surprisingly enough most of the issues found initially are still there almost with no changes. One key aspect that makes aliasing such a hard issue is that its local changes in a program listing may effect other regions unexpectedly – but at the same time this is its strength, since no additional copying is required. [16] had particularly analysed previously fixed bugs for a long-term period over last releases and found out - according to the bug distribution over time - that undesired memory behaviour is one of the most expensive bug reasons in terms of time and efforts to locate and fix.

Object-orientation [1], based on concepts such as encapsulation, polymorphism and inheritance, has been one of the most successful and widely adapted programming paradigms by now for a long time in industry, hence its combination with pointer structures remains a relevant research task up to date [14].

Prolog [27] is considered for program verification for several reasons. First, it is a logical and declarative programming language which offers a high abstraction in writing Horn-clauses as they correspond with defined axioms and rules. A proof tree occasionally insists on a back-tracking strategy, which Prolog supports for free as one of its core-language features. The hope is Prolog's *generate-and-test* goal strategy [27] may be found useful in simplifying and abstracting a proof significantly. Second, programs and internal states can be represented as terms. Terms can be easily processed in Prolog. The hope is, abduction and general symbolic term evaluation will allow generating lemmas more efficiently and make the reasoning terminate and terminate earlier. As a previous successful verification attempt, [12] shall be noted. The authors solved a fairly hard problem from mathematical numerics using Prolog elegantly and straight. Since proofs are rule-centric, a proof contradiction will eventually help generating counter-examples easily by simply matching terms from the memory and from a rule or axiom.

Example1 – memory leak

```
MyClass object1=new MyClass();
...
object1=new MyClass();
```

Example2 — unachievable memory

```
// object1 has been created
MyClass object2=new MyClass();
object2.ref=object1;
```

Example3 – invalid memory access

```
// object1.ref==null
value = (object1.ref).attribute1;
```

Example4 – data structure with cycle

```
object1.next=object1;
...
root=object1;
while(root.next!=null){
  printf(%d, object.data);
```

```
        root=root.next;
}
```

Example 1 demonstrates the case where a fresh memory region is allocated, and without freeing it, allocates it again, which may cause the previously created region becoming unachievable. The second example demonstrates where `object2` is linked to an occupied `object1`, but `object2` itself remains unused. A very common problem in practice might be the third example, when an object reference is not set, but later referenced causing either an abnormal runtime failure or continues execution, which might be even worse in realistic scenarios because the further program execution becomes totally unpredictable with invalid value settings. The forth example might not immediately be seen as a problem, but if, for whatever reason, there is a cycle in `root` the program will not terminate. Apart from direct consequences like crashes or non-termination, one more side effect is there are spontaneous allocations/deallocations taking place on runtime which may eventually become a performance bottleneck.

II. CURRENT APPROACHES

One approach to verify correctness of dynamic memory is to get the program run and to record all memory cells that will be referenced and allocated/deallocated. This is what the *valgrind* tool [29] does. This open-source tool requires on compilation guarding memory-checking code is injected to the assembly code. Not only that the enhanced program runs with huge delays, the general problem underneath this approach as well as SAFECode [25], which on runtime checks whether programmer-inserted assertions are fulfilled, is that only a small subset of all possible execution paths can be tested and that it requires additional code is inserted. For this reason, only static approaches are considered further that analyse the incoming program listing prior to running it.

In order to address the problems mentioned in the introduction part, several approaches exist: (i) Shape-based Analysis [26], [22], (ii) Separating heap. For sake of completeness a heap-free alternative proposed by (iii) Tufte and Talpin [28] and Meyer [15] shall be mentioned, who both appeal to a stack-based approach, if any possible, to avoid expensive heap allocation and deallocation operations. Automatic handling of stack-based locations is essential for both, where the stack sizes are determined during compilation. Thus, control passing which happens during a call will allow to allocate objects almost for *free*, since a stack frame needs to be created in any case, and no more expensive heap operations are needed in fact, for instance garbage collection. The disadvantage of (iii) is, however, there is often a platform-dependent restriction on stack sizes and number of entries, so in practise there are tough frame restrictions, e.g. a maximum offset, which should not be exceeded without getting a severe performance penalty on concrete target architectures at the same time. Meyer [15] asks to turn garbage collection steadily on during program execution. This implies for efficient execution runtime critical

parts will not trigger dynamic memory operations and those operations are opted out as efficiently as it can possibly be done.

Approaches (i) and (ii) are similar, both describe the memory state, although (i) describes the entire dynamic heap as an entire graph, where edges are region dependencies and vertices are locations. The problem with (i) is *locality*, because if a particular function is called, the entire graph has to be specified before, after and during the call, where approach (ii) allows to hide all non-affected heaps (*framed heaps*) this is what is meant by *locality principle* in terms of *Separation Logic* [24]. The most important concept behind Separation Logic [24], [23], [6] is the specification of two non-interleaving memory regions. Heaps might be composed, and programs may change heaps. If two heaps are connected, then the dependency has to be added explicitly to a current heap's specification. If a heap depends on some other heap data, then this is called *aliasing* (or *big brother property* as found in [15]).

The first implementation which makes use of Separation Logic is Smallfoot [5], [6]. In order to extend deductive reasoning capabilities, an abductive approach was proposed, called bi-abduction [7], for Separation Logic, which is a constructive guess of unchanged heaps by a greedy symbolic table-construction algorithm that chooses bigger rules first. The extension of Smallfoot is called *SpaceInvader*.

Hurlin extends in [10] the classic Separation Logic proposed in [23] by classes for a Java-like language. He suggests a *heap factorization*, an attempt to normalise heaps in order to remove redundant heap specification fragments which are considered as *noise*, even if the main goal of his thesis is focused on multi-threaded applications. He re-uses the same concept of *abstract predicates* as it was introduced by [20], [8], and generates unchanged parts during deduction with a parallel algorithm.

Parkinson [20] introduces a Java-like language with object-orientated features. Nevertheless, many problems are not being addressed yet: abstraction mismatch on encapsulation and inheritance, particularly, the problem of expanding specifications in subclasses seems to be a real hinder in simple and elegant specifications. The most essential contribution of [20] is the introduction of *abstract predicates*, although there are currently tough restrictions concerning expressibility. Super calls, static fields, reflection, inner classes and quantified predicates, for example, are currently missing language features.

Verifast [11] is another forward verifier based on Separation Logic. In comparison to all previously introduced verifiers which do very similar operations on the heap, all introduced conventions per tool differ strongly, and it does not automatically process loop invariants nor predicates - here it depends entirely on user-interaction or requires explicit injections within specification annotations inside the program which are used as internal reorganisation commands.

In [2] objects as class-instances are treated as records, typing and verification rules are introduced and a soundness proof is provided. Problems which neither in [2] nor [1] are addressed are that objects may have references to other objects

and that a lack in abstraction causes a dramatic increase in specification length which makes it in practice impossible to read and understand specification to a full extend. The memory state is specified by temporal predicates and a result-register for the previous computation step's result. There is no general recursive definition allowed, although [13] attempts to relax this hard restriction by an algebraic ideal-construction. Still there are hard restrictions, such as no aliasing nor late binding at all, and object-records only which even may become unsound for eager type evaluation.

III. ARCHITECTURE AND DESIGN

Before going on with more details on the architecture, the prerequisites on the architecture shall be summarised. The proposed architecture may be considered for teaching purposes in the future:

1) Automatic proof. The program and its annotations shall be sufficient in order to get the verification run. If there is an endless cycle in the proof however, there shall be no mandatory recognition, since termination is beyond the main focus of this work.

2) Openness. The provided architecture shall be open for extensions and variability, and the attached models shall be exportable, so it might eventually be passed through to another arbitrary model transformer, if needed.

3) Extensibility. The target language shall be fixed but interchangeable with an imperative programming language in the front-end. The rules and user-defined predicates shall be designed amendable, so the user may want to add rules directly to the rule set.

4) Plausibility: There shall be configurable visualisation facilities, so the incoming annotated program may be retrospected on each stage of the verification process. If, for instance, a proof fails or stops abruptly the user would perhaps like to see the proof tree and a counter-example.

In figure 1 the architecture of the Prolog-based verification system is shown. The input is a C-program with object-orientated extension that is annotated with assertions specifying the dynamic memory. The shortened syntax can be described by the Extended Backus-Naur form in figure 2. Not mentioned definitions as actual parameters, blocks, class methods, variable declarations have been skipped here for the sake of readability and follow mostly ANSI C. For readability purposes the expression sub-grammar has not been expanded according to its precedence hierarchy nor for optional assertions. `new` and `delete` reserve/free new chunks in the heap associated with previously defined locations. The access to heap memory is performed by [<location>], where <location> denotes either a field variable, another object's field or a either of those with an offset in order specify non-aligned memory regions, for instance. The rule <funcall> denotes the syntax for a method call, which may have a object specifier optionally and a method name which is required to exist with the matching total number and types of parameters being passed as expressions.
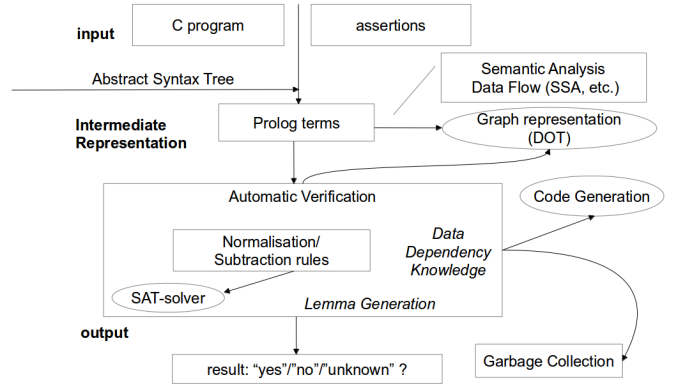

Fig. 1. Verification architecture for Prolog-based reasoning on dynamic memory

C-programs are annotated by assertions which are injected as usual Prolog terms into blocks. Blocks are encoded as lists of statement-terms. Assertions are inductively defined and can be found in figure 3. Keep in mind the expression might request object references and $\alpha(\vec{p})$ assumes predicate named α was defined prior to using it, and $\vec{p}$ contains as many actual parameters as the arity of predicate α require there are.

$\langle prog \rangle ::= \langle class \rangle \; \langle id \rangle$ '{' { $\langle field \rangle$ | $\langle method \rangle$ } '}'

$\langle location\_1 \rangle ::= \langle id \rangle$ | $\langle id \rangle$ '.' $\langle id \rangle$ | 'this' '.' $\langle id \rangle$

$\langle location \rangle ::= \langle location\_1 \rangle$ [('+' | '-') $\langle int \rangle$]

$\langle stmt \rangle ::= \langle lhs \rangle$ '=' { $\langle lhs \rangle$ '=' } $\langle expr \rangle$
  | 'if' $\langle cond \rangle$ $\langle block \rangle$ ['else' $\langle block \rangle$]
  | 'while' $\langle cond \rangle$ $\langle block \rangle$
  | 'new' '(' $\langle location\_1 \rangle$ ')'
  | 'delete' '(' $\langle location\_1 \rangle$ ')'
  | $\langle func\_call \rangle$

$\langle lhs \rangle ::= \langle location\_1 \rangle$ | '[' $\langle location \rangle$ ']'

$\langle cond \rangle ::= \langle expr \rangle$ $\langle rel \rangle$ $\langle expr \rangle$

$\langle rel \rangle ::=$ '&&' | '||' | '==' | '!=' | '$\leq$' | '$\geq$' | '>' | '<'

$\langle expr \rangle ::= \langle expr \rangle$ ('+' | '-' | '*') $\langle expr \rangle$
  | '-' $\langle expr \rangle$
  | '[' $\langle location \rangle$ ']'
  | [('this' | $\langle id \rangle$) '.'] $\langle id \rangle$ '(' $\langle act\_params \rangle$ ')'
  | $\langle location \rangle$
  | $\langle int \rangle$

$\langle func\_call \rangle ::=$ [('this' | $\langle id \rangle$) '.'] $\langle id \rangle$ '(' $\langle act\_params \rangle$ ')'

Fig. 2. Syntax definition of C-programs with object-oriented extension

For example,

```
int f(int a, int b) @ a<10 @ {
  id=2; a=1; b=6;
} @ a->5 * b->c * c->object(myClass1,15) @
```

is transformed into this Prolog-term:

```
function(f, int,
  [param(a,int), param(b,int)],
  [assert(le(a,10)),
   assign(id,2), assign(a,1), assign(b,6),
   assert(a->5 * b->c *
        c->object(myClass1,15))])
```

An important specification fragment of the intermediate Prolog-term syntax can be found in figure 4, where the remaining part is close to the syntax of figure 2. Apart from *'ite'* which represents a *if-then-else*-construct with at least one block for the if-case and one more optional block for the else-block – as long as it was provided, there are also while-loops and further constructs, like class definitions, which will not be mentioned here for simplicity purposes. All expressions, particularly with binary operators, are encoded as terms where the literal operator becomes the functor, for example add(i,7).

$H ::=$ **emp** | **true** | **false** | *atomic formulae*
$\qquad\quad x \mapsto E$ | *location map*
$\qquad\quad H * H$ | *heap separation*
$\qquad\quad H \vee H \mid H \wedge H$ | *conjunction*
$\qquad\quad \exists x.H$ | *quantification*
$\qquad\quad \alpha(\vec{p})$ *predicate unfold*
$\qquad\qquad\qquad\qquad$ where
$\qquad\quad x$ is a location
$\qquad\quad E$ is a well-defined expression (enumeration)
$\qquad\quad \vec{p}$ is a comma-separated parameter vector

Fig. 3.   Syntax definition of heap and stack assertions

Since the architecture is designed flexible, it allows the user to interchange the compiler front-end for a different language, so the user has the possibility to write own Prolog-terms directly without even having an ordinary C-program. In

⟨*stm*⟩ ::= ... | 'new' '(' ⟨*loc_1*⟩ ')'
$\quad$ | 'delete' '(' ⟨*loc_1*⟩ ')'
$\quad$ | 'funcall' '(' ⟨*id*⟩ [',' ⟨*act_params*⟩]) | ...
$\quad$ | 'ite' '(' ⟨*cond*⟩ ',' ⟨*block*⟩ [',' ⟨*block*⟩] ')'

⟨*loc_1*⟩ ::= ⟨*id*⟩ | 'oa' '(' ⟨*id*⟩ '.' ⟨*id*⟩ ')'

⟨*loc*⟩ ::= 'offset' '(' ⟨*loc_1*⟩ [',' ⟨*offset*⟩] ')'

⟨*offset*⟩ ::= ⟨*int*⟩ | 'minus' '(' '0' ',' ⟨*int*⟩ ')'

⟨*expr*⟩ ::= ('add' | 'sub' | 'mul') '(' ⟨*expr*⟩ ',' ⟨*expr*⟩ ')'
$\quad$ | 'mem' '(' ⟨*loc*⟩ ')'
$\quad$ | ⟨*loc*⟩ | ⟨*int*⟩
$\quad$ | 'funcall' '(' ⟨*id*⟩ [',' ⟨*act_params*⟩] ')'

Fig. 4.   Syntax definition of Prolog-terms

this case syntax and semantic constraints remain on full responsibility to the user. Prolog-terms are internally checked and may also be directed to a graphical output, e.g. for proof tree visualisation. Antlr 4 [21] is currently used as compilation front-end.

Once the Prolog term is constructed, it can be passed to the verification. Hereby, the term is now processed while the internal environment, which has to keep the states of the memory, needs to be updated after every statement. All locals are residing in stack, where dynamically allocated memory locations may remain in memory – even if a stack-based variable stores a dynamic address it would be freed at the end of a block.

If we decide to specify a list concatenation of two lists, we have several opportunities to describe the heap. If list(s,e) denotes a heap predicate where s is the location of the beginning root element of a list, and e denotes the last element in that list, then having two lists x and y with x->a,b,c and y->d,e,f will concatenate for instance to either (i) x->a,b,c,d,e,f * y->f or to (ii) x->a,b,c * y->d,e,f * z->a,b,c,d,e,f. Remark: The ','-operator is defined as a list constructor with variable input amount for all consecutive objects currently in memory linked together to a simply-linked list [23]. The main difference between (i) and (ii) is that (i) requires only a single assignment if the end of x is known, therefore x and y are no more as they used to be before concatenation. (ii) creates an entirely new copy of all element from x and y and does not touch neither x nor y. (ii) is safer from a general reuse perspective, but it is considerably slower and consumes more memory due to additional copies to be generated.

Finally, the SMT-solver is required whenever taking out trivial calculations, for instance in basic arithmetics. For instance, if there is an expression that might be reduced to a value, then this should in general be tried first before triggering a certain rule. Beside finding solutions to basic arithmetic and other theories, re-arrangement needs to be taken into consideration. Formal rules will usually also not deal too much about heap permutation, although a strategy must be found and is crucial in fact for the overall performance.

IV. CONCLUSION

So far the open architecture was presented providing several suggestions for further research activities. Prolog was proposed as specification and proof platform for memory-specific research, e.g. on extending the expressibility of abstract predicates or abduction. We believe, questions related to abduction in Separation Logic with objects still have not been profoundly investigated yet, as well as some object-oriented features like polymorphism in Separation Logic.

The platform might be used to incorporate with existing compiler packages in order to research improvement on code optimization during the alias analysis phase, but also garbage collection, based on knowledge obtained during the dynamic memory verification.

Further rules of normalisation and re-arrangement will be applied to cover more real world scenarios, particularly in order to resolve arithmetic equivalency by the integration of a SMT-solver ([18], [19]).

Related work includes Jacobs [11] who suggests to investigate Banerjee's Regional Logic approach [4] as substitute for the Symbolic Execution approach [6].

REFERENCES

[1] Abadi M., Cardelli L. *A. Theory of Objects*. New York: Springer, 1996, 396 p.

[2] Abadi M., Leino K. R. M. *A Logic of Object-Oriented Programs*, Proc. of the 7th Int. Joint Conf. CAAP/FASE on Theory and Practice of Software Development, Springer, 1997, pp. 682-696.

[3] Allen R., Kennedy K. *Optimizing Compilers for Modern Architectures*. 2001, 790p.

[4] Banerjee A., Naumann D. A. and Rosenberg S. *Regional logic for local reasoning about global invariants*. ECOOP, LNCS 5142, 2008, pp. 387-411

[5] Berdine J., Calcagno C. and O'Hearn P. W. *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*. FMCO, 2005, pp. 115-137.

[6] Berdine J., Calcagno C. and O'Hearn P. W. *Symbolic Execution with Separation Logic*. APLAS, 2005, pp. 52-68.

[7] Calcagno C., Distefano D., O'Hearn P. and Yang H. *Compositional shape analysis by means of bi-abduction*. Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2009, 36, pp. 289-300 .

[8] Distefano D., Parkinson M. *jStar: Towards practical verification for Java*. OOPSLA, 2008, pp. 213-226.

[9] Hind M., *Pointer Analysis: Haven't We Solved This Problem Yet?* ACM, PASTE'01, 2001, pp. 54-61.

[10] Hurlin C. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD Thesis, Universit Nice - Sophia Antipolis, 2009, 195p.

[11] Jacobs B., Piessens F. *The VeriFast Program Verifier*. Leuven University, 2008, 5p.

[12] Koch H., Schenkel A., Wittwer P. *Computer-assisted Proofs in Analysis and Programming in Logic: A Case Study*. SIAM Review, 1996, no.4(38), pp. 565-604.

[13] Leino K. R. M. *Recursive object types in a logic of object-oriented programs*. Nordic J. of Computing, (5), 1998, pp. 330-360.

[14] Meyer B. *Proving Pointer Program Properties - Part 1: Context and Overview*, Journal of Object Technology, no.2(2), March-April 2003, pp. 87-108.

[15] Meyer B. *Proving Pointer Program Properties - Part 2: The Overall Object Structure*, Journal of Object Technology, no.3(2), May-June 2003, pp. 77-100.

[16] Miller B. P., Fredriksen L. and So B. *An Empirical Study of the Reliability of UNIX Utilities*. Proc. of the Workshop of Parallel and Distributed Debugging, Digital Equipment Corporation, 1990, pp. 1-22.

[17] Muchnik S. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 2007, 856p.

[18] Nanevski A., Morrisett G., Shinnar A., Govereau P. and Birkedal L. *Ynot: Reasoning with the awkward squad*, ACM SIGPLAN Int. Conf. on Functional Programming, 2008, p. 12.

[19] OpenSMT project. http://code.google.com/p/opensmt/

[20] Parkinson M. *Local Reasoning for Java*. PhD Thesis, Cambridge University, 2005, 169 p.

[21] Parr T. *The Definitive ANTLR 4 Reference: Building Domain-Specific Languages*. O'Reilly, 2013, 328p.

[22] Pavlu V. *Shape-based Alias Analysis. Computing Alias Sets from Shape Graphs to Evaluate the Precision of Shape Analyses*. VDM Verlag Dr. Müller, 2010, 117p.

[23] Reynolds J. C. *Separation Logic: A Logic for Shared Mutable Data Structures*, Lecture Notes in Computer Science, 2002, pp.55-74.

[24] Reynolds J. C. *An Introduction to Separation Logic*. Carnegie Mellon University, 2009, 204p.

[25] SAFECode within LLVM project. http://llvm.org

[26] Sagiv M., Reps T., Wilhelm R. *Parametric shape analysis via 3-valued logic*. ACM Trans. Program. Lang. Syst., 2002, 24, pp. 217-298.

[27] Sterling L., Shapiro E. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*, MIT Press, 1994, 552 p.

[28] Tofte M., Talpin J.-P. *Implementation of the typed call-by-value λ-calculus using a stack of regions*. Proc. of the 21st ACM SIGPLAN-SIGACT. 1994, pp. 188-201.

[29] Valgrind project. http://www.valgrind.org