Abstract Contract Synthesis and Verification in the Symbolic $\mathbb{K}$ Framework*

María Alpuente[1], Daniel Pardo[1], and Alicia Villanueva[1]

[1]DSIC, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
{alpuente,daparpon,villanue}@dsic.upv.es

Abstract

In this article, we propose a symbolic technique that can be used for automatically inferring software contracts from programs that are written in a non-trivial fragment of C, called KernelC, that supports pointer-based structures and heap manipulation. Starting from the semantic definition of KernelC in the $\mathbb{K}$ semantic framework, we enrich the symbolic execution facilities recently provided by $\mathbb{K}$ with novel capabilities for contract synthesis that are based on abstract subsumption. Roughly speaking, we define an abstract symbolic technique that axiomatically explains the execution of any (modifier) C function by using other (observer) routines in the same program. We implemented our technique in the automated tool KindSpec 2.1, which generates logical axioms that express pre- and post-condition assertions which define the precise input/output behavior of the C routines. Thanks to the integrated support for symbolic execution and deductive verification provided by $\mathbb{K}$, some synthesized axioms that cannot be guaranteed to be correct by construction due to abstraction can finally be verified in our framework with little effort.

Keywords contract inference, symbolic execution, abstract subsumption, deductive verification

1 Introduction

Checking software contracts is one of the most promising techniques for achieving software reliability [29]. Contracts typically consist of essential requirements that are imposed on the arguments and result values when functions are invoked.

*This work has been partially supported by the EU (FEDER) and Spanish MINECO under grant TIN2015-69175-C4-1-R, and and TIN2013-45732-C4-1-P, and by Generalitat Valenciana PROMETEOII/2015/013. Daniel Pardo was supported by FPU-ME grant FPU14/01830.

Given its interest, considerable effort has recently been invested in achieving automatic support for equipping programs with extensive contracts, yet the current contract inference tools are still often unsatisfactory in practice [13].

Given a program P, the contract synthesis problem for P is generally described as the problem of inferring a likely specification for every function m in P (hereinafter referred to as *method*) that uses I/O primitives and/or modifies the state. By automating the tedious and time-consuming process of generating contract assertions, programmers can reap the benefits of assertion–based debugging and verification methods with reasonable effort.

In this article, we develop a contract synthesis methodology that is based on symbolic computation and that applies to heap-manipulating programs that are written in a non-trivial fragment of C, called KernelC [16], which includes functions, I/O primitives, dynamically allocated structures, and pointer manipulation. The contracts that we synthesize essentially consist of logical assertions that characterize the function behavior and that are expressed as method preconditions (imposed on the arguments) and post-conditions (relating the arguments and the result for a method). As a by-product of the synthesis process, for each method we also infer the list of references to memory locations whose value might be affected by the execution of the method, which can be key for sound usage of contracts as, for instance, in compositional verification [11].

In [1, 2], a preliminary technique for inferring method contracts was proposed that is based on the classification scheme for data abstractions developed in [26], where a function may be either a *constructor*, a *modifier*, or an *observer*. The intended behavioral specification of any *modifier* function m of P is expressed as a set of logical assertions that characterize the pre- and post-states of the execution of m by using the *observer* functions in P. The inference technique of [1, 2] relies on symbolic execution (SE), a well-known program analysis technique that allows programs to be executed using *symbolic* input values instead of actual (concrete) data so that the program execution manipulates symbolic expressions involving the symbolic values [25, 33]. More precisely, for each pair (s, s') of initial and final states in the symbolic execution of m, an implicative axiom $p \Rightarrow q$ is synthesized where both the antecedent p and the consequent q are expressed in terms of the (sub-)set of program observers that *explain* (i.e., characterize) s and s'. For instance, for the case of a modifier method `push` that adds the element `x` at the top of a given bounded stack `t` (and assuming the traditional meaning for the observer functions `top`, `isfull`, and `size`), a typically expected logical axiom describing the effect of running the call `push(x,t)` could be: `isfull(t)=0` $\wedge$ `size(t)=n` $\Rightarrow$ `top(t)=x` $\wedge$ `size(t)=n+1` (i.e. starting from an initial computation state where the input stack `t` is not full, at the final state the top stack element is `x` and the stack size has been increased by 1), where the value 0 is used to represent in C the Boolean *false* value. In the methodology of [1, 2], this axiom is generated by analyzing the results of symbolically executing each observer method o (such as `top`, `isfull`, `size`, ...) from initial configurations that symbolically represent the pre-state s and the post-state s' of the execution of method m.

The symbolic infrastructure developed in [1] was built on top of the rewriting-

based, programming language definitional framework $\mathbb{K}$, which facilitates the development of executable semantics of programming languages and related formal analysis techniques and tools [36]. However, it was developed by reusing spare features of a hand–crafted, formal verifier called MatchC [37] that was formerly provided within $\mathbb{K}$ but is currently unsupported since the current deductive verification infrastructure of $\mathbb{K}$, which is language-independent, can be instantiated with the operational semantics of any programming language to automatically generate a program verifier for that language [42]. On the other hand, the underlying methodology in [1] was rather limited since a fixed threshold for loop unrolling was imposed on the symbolic computations in order to avoid non-termination risks. The methodology in [2] was already based on the current, language-independent native extension of $\mathbb{K}$ that supports symbolic execution [5] but inherited the loop unrolling strategy based on depth bounds from [1].

In this work, we improve the inference power of [1, 2] by endowing $\mathbb{K}$'s symbolic execution with modern subsumption techniques based on abstraction [4] and lazy initialization [24]. The fact that this symbolic infrastructure is much more flexible and (potentially) language-independent allows us to define a generic, more accurate, easily maintainable and robust framework for the inference of program contracts that could be adapted to other languages defined within the $\mathbb{K}$ framework with little effort. A preliminary version of the abstract contract synthesis algorithm developed in this article first appeared in [3].

1.1 Contributions

The main contributions of this paper are as follows.

1. We define a symbolic technique that synthesizes contracts for heap-manipulating code while coping with infinite computations. This is done by

   (a) augmenting $\mathbb{K}$'s symbolic execution with lazy initialization [24] and a widening operator based on abstract subsumption [4], and

   (b) synthesizing method pre- and post-conditions by means of a contract synthesis algorithm that explains the (initial and final) abstract symbolic execution states by using the program observers.

   In comparison to [3], in this article we provide a detailed formalization of the symbolic heap subsumption and abstract heap subsumption techniques that are essential for developing (a). Regarding (b), because of the abstraction, some inferred axioms for method m cannot be guaranteed to be correct and are kept apart as *candidate* (or overly general) axioms. A refinement post-processing is then formalized that first removes every candidate axiom for which an instance is refuted and then filters out any redundant elements from the surviving axioms.

2. As a complement to the symbolic contract synthesis technique, we provide a novel methodology that distills a set of formal *verification rules*

(written in $\mathbb{K}$'s verification syntax) from the deployed symbolic execution trees. This allows some correct axioms that were roughly categorized as candidates because of the abstraction to be automatically verified within the very same deductive verification infrastructure provided by $\mathbb{K}$ [42]. This may yield more concise specifications since other (more specialized axioms) might be subsumed by the new discovered correct axioms.

3. The proposed techniques are implemented in the KindSpec 2.1[1] system, which builds on the capabilities of the SMT solver Z3 [31] for the inference. The synthesized contracts are further simplified by applying duplicate elimination and subsumption checking, to be given a compact representation that abstracts the user from any implementation details.

1.2 Plan of the paper

This paper is organized as follows. In Section 2, we introduce a KernelC program that is used throughout the paper as the leading example to discuss the effectiveness and adequacy of our contract synthesis methodology. In Section 3, we summarize the key concepts of the $\mathbb{K}$ framework that are crucial for this work. Section 4 explains how we augmented the symbolic machinery of $\mathbb{K}$ with lazy initialization and abstract subsumption to support the execution of programs with symbolic data structures and unbounded loops. Section 5 formalizes our contract synthesis technique together with a key postprocessing refinement that improves the quality of the contracts that we infer. Section 6 outlines an extension of our technique that provides for (semi-)automated axiom verification, which may further better the learned contracts. Section 7 presents the prototype tool that we implemented to evaluate the proposed techniques together with some experimental results on a set of benchmark programs. Finally, in Section 8, we discuss the related work and we conclude.

2 Method Specification: A Running Example

By abuse, we use the standard terminology for contracts of object-oriented programming and speak of *methods* when we refer to KernelC functions. Like many state-of-the-art formal specification approaches, we assume to work in a contract-based setting [29], where the granularity of specification units is at the level of one method. Our inference technique relies on the classification scheme developed for data abstractions in [26], where a function (method) may be either a *constructor*, a *modifier*, or an *observer*. A constructor returns a new data object from scratch (i.e., without taking any object as an input parameter); a modifier alters an existing object (i.e., it changes the state of one or more of its attributes); and an observer inspects the object and returns a value characterizing one or more of its state attributes. Since the C language does not enforce data encapsulation, we cannot presume purity of any function; thus, we do not

[1] `http://safe-tools.dsic.upv.es/kindspec2_1/`

assume the traditional premise which states that observer functions do not cause side effects. In other words, any function can potentially be a modifier, and we simply define an observer as any function whose return type is different from `void`.

Let us introduce the leading example that we use to describe our inference methodology: a KernelC implementation of an abstract data type for representing sets by using linked lists. The example is composed of 7 methods: one constructor (`new`), one modifier (`insert`), and five observers (`isnull`, `isempty`, `isfull`, `contains`, and `length`). Note that the program observers do not modify any program objects, even if purity of observers is not required in our framework. As is usual in C, logical observers return value 1 (resp. 0) to represent the traditional boolean value *true* (resp. *false*).

Example 1 *Consider the program fragment*[2] *given in Figure 1, where we define set operations over a data structure (*`struct set`*) that records the number of elements contained in the set (field* `size`*), the maximum number of elements that can be held (field* `capacity`*), and a pointer to a list that stores the set elements (field* `elems`*). Each node of the list is a record data structure (*`struct lnode`*) that contains an integer value (field* `value`*) and a pointer to the subsequent list element (field* `next`*).*

A call `insert(s,x)` *to the* `insert` *function proceeds as follows. It first checks that the pointer* `s` *to the set structure is different from* `NULL`*, that the set is not full, and that* `x` *is not in the set yet. Then, a new list node* `*new_node` *is allocated, filled with the value* `x`*, and inserted as the first element of the list; also, the size of the set is increased by 1 and the call returns 1. Otherwise, 0 is returned and* `s` *is not modified.*

The following observers return 0 unless explicitly stated otherwise. `isnull(s)` *returns 1 only if the pointer* `s` *references to* `NULL` *memory;* `isempty` *returns 1 if* `s` *is initialized but its* `elems` *field is* `NULL`*;* `isfull(s)` *returns 1 if the* `size` *of* `s` *is greater than or equal to its* `capacity`*; and* `contains(s,x)` *returns 1 if the value* `x` *is found in* `s`*. The function* `length(s)` *incrementally counts up the number of elements (nodes) in the set* `s` *by traversing the list* `s->elems` *and returns this number; it returns 0 in the case when the pointer to the set* `s` *is* `NULL`*.*

From the source code of the program, for each modifier function m, we aim to synthesize a contract of the form $< P, Q, \mathcal{L} >$ where P is the method precondition, Q is the method postcondition, and $\mathcal{L}$ is the set of program locations (local variables, data-structure pointers and fields, and method parameters) that are (potentially) affected by the method execution. We first compute a set of implication formulae of the form $p \Rightarrow q$, where p and q are conjunctions of equations $l = r$. The left-hand side l of each equation can be either: 1) a call to an observer function, and then r represents the return value of that call; or 2) the keyword `ret`, and then r represents the value returned by the modifier function m being observed. Then, given the set of implication formulae

[2] For the sake of completeness, the full program code is given in Appendix A.

```
1  #include <stdlib.h>
2
3  struct lnode{
4   int value;
5   struct lnode *next;
6  };
7
8  struct set {
9   int capacity;
10  int size;
11  struct lnode *elems;
12 };
13
14 struct set* new(int capacity)
15 {...}
16
17 int insert(struct set *s, int x)
18  struct lnode *new_node;
19  struct lnode *n;
20  int found;
21
22  if(s==NULL)
23   return 0; /* NULL set */
24  if(s->size >= s->capacity)
25   return 0; /* no space left */
26
27  n = s->elems;
28  found = 0;
29  while(n != NULL) {
30   if(n->value == x)
31    found = 1;
32   n = n->next;
33  }
34
35  if(found)
36   return 0; /* element already in
          the set */
37
38  new_node = (struct lnode*) malloc(
        sizeof(struct lnode));
39  if(new_node == NULL)
40   return 0; /* no memory left */
41  new_node->value = x;
42  new_node->next = s->elems;
43
44  s->elems = new_node;
45  s->size += 1;
46  return 1; /* element added */
47 }
48
49 int isnull(struct set *s) {...}
50 int isempty(struct set *s) {...}
51 int isfull(struct set *s) {...}
52 int contains(struct set *s, int x)
        {...}
53 int length(struct set *s) {...}
```

Figure 1: Fragment of the KernelC implementation of a set data type.

$\{p_1 \Rightarrow q_1, \ldots, p_n \Rightarrow q_n\}$, P is defined as $p_1 \vee \ldots \vee p_n$, the postcondition Q is the formula[3] $(p_1 \Rightarrow q_1) \wedge \ldots \wedge (p_n \Rightarrow q_n)$, and the elements of $\mathcal{L}$ refer to locations whose value might be affected by the execution of m, that is, all memory locations of the pre-state that do not belong to the set $\mathcal{L}$ remain allocated and are left unchanged in the post-state.

Example 2 *The intended postcondition Q for the modifier function* `insert(s,x)` *of Example 1 contains five axioms (each one given by an implication), which are shown in Figure 2. We adopt the standard primed notation to distinguish variable values after the execution of the method from their value before the execution.*

The first axiom can be read as follows: if the outcome of `isnull(s)` *is 1 before the call to* `insert(s,x)`*, then, after execution, the set is still null and the value returned by* `insert(s,x)` *is 0, which means that the element* `x` *was not*

[3]This is similar to the idea of contracts with *named behaviors* as provided in the ACSL contract specification language for C [7].

$$\left(\ \texttt{isnull(s)} = 1\ \right) \Rightarrow \left(\ \texttt{isnull}(\texttt{s}') = 1\ \wedge\ \texttt{ret} = 0\ \right)$$

$$\left(\ \texttt{isfull(s)} = 1\ \right) \Rightarrow \left(\begin{array}{l} \texttt{contains}(\texttt{s}', \texttt{x}) = \texttt{contains}(\texttt{s}, \texttt{x}) \wedge \\ \texttt{length}(\texttt{s}') = \texttt{length}(\texttt{s}) \wedge \\ \texttt{isfull}(\texttt{s}') = 1\ \wedge\ \texttt{ret} = 0 \end{array}\right)$$

$$\left(\ \texttt{contains}(\texttt{s}, \texttt{x}) = 1\ \right) \Rightarrow \left(\begin{array}{l} \texttt{contains}(\texttt{s}', \texttt{x}) = 1 \wedge \\ \texttt{length}(\texttt{s}') = \texttt{length}(\texttt{s})\ \wedge\ \texttt{ret} = 0 \end{array}\right)$$

$$\left(\ \texttt{isempty(s)} = 1\ \wedge \texttt{isfull(s)} = 0\ \right) \Rightarrow \left(\begin{array}{l} \texttt{isempty}(\texttt{s}') = 0\ \wedge\ \texttt{contains}(\texttt{s}', \texttt{x}) = 1 \wedge \\ \texttt{length}(\texttt{s}') = \texttt{length}(\texttt{s}) + 1\ \wedge\ \texttt{ret} = 1 \end{array}\right)$$

$$\left(\begin{array}{l} \texttt{isnull(s)} = 0\ \wedge \texttt{isempty(s)} = 0 \wedge \\ \texttt{isfull(s)} = 0\ \wedge \texttt{contains(s,x)} = 0 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull}(\texttt{s}') = 0\ \wedge\ \texttt{isempty}(\texttt{s}') = 0 \wedge \\ \texttt{contains}(\texttt{s}', \texttt{x}) = 1 \wedge \\ \texttt{length}(\texttt{s}') = \texttt{length}(\texttt{s}) + 1\ \wedge \texttt{ret} = 1 \end{array}\right)$$

Figure 2: Expected postcondition axioms for the `insert` method

inserted. As for the last axiom, it can be read as follows: if the set is not null, not full, and not empty, and there is no node in the list with value `x`*, then, after execution, the set remains non-null and non-empty, the value* `x` *is now in the set, the length is increased by 1, and the call to* `insert(s,x)` *returns 1, which denotes a successful insertion.*

3 The (symbolic) $\mathbb{K}$ Framework

$\mathbb{K}$ is a rewriting-based framework for engineering language semantics [36]. Provided that the syntax and semantics of a programming language are formalized in the internal language of $\mathbb{K}$, the system automatically generates a parser, an interpreter, and formal analysis tools such as model checkers and deductive theorem provers and verifiers. Complete formal program semantics are currently available in $\mathbb{K}$ for Scheme, Java 1.4, JavaScript, Python, Verilog, and C among others [36].

A language definition in $\mathbb{K}$ consists of three parts: the BNF language syntax, the structure of program configurations, and the semantic rules. Program configurations are represented in $\mathbb{K}$ as potentially nested structures of labeled *cells* (or containers) that represent the program state. Similarly to the classic operational semantics, program configuration cells include a computation stack or continuation (named k), one or more environments (env, heap), and a call stack (stack) among others, and are represented as algebraic datatypes in $\mathbb{K}$. That is, $\mathbb{K}$ cells can be lists, maps, (multi)sets of computations, or a multiset of other cells.

The part of the $\mathbb{K}$ program configuration structure for the KernelC semantics that is relevant to this work is

$$\left\langle\ \langle \mathrm{K} \rangle_{\mathsf{k}} \langle \mathrm{Map} \rangle_{\mathsf{env}} \langle \mathrm{Map} \rangle_{\mathsf{heap}}\ \right\rangle_{\mathsf{cfg}}$$

where the env cell is a mapping of variable names to their memory positions, the heap cell binds the active memory positions to the actual values (i.e., it stores information about pointers and data structures), and the k cell represents a stack of computations waiting to be run, with the left-most element of the stack being the next computation to be undertaken. For example, the configuration

$$\langle\ \langle \texttt{int y = x + 2; } \cdots \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto \&\mathsf{x} \rangle_{\mathsf{env}} \langle \&\mathsf{x} \mapsto\ \mathrm{tv}(\mathit{int}, 5) \rangle_{\mathsf{heap}}\ \rangle_{\mathsf{cfg}} \tag{1}$$

models a computation state where the subsequent program instruction to be interpreted (by the KernelC semantics defined in $\mathbb{K}$) is the assignment `int y = x + 2`. At the same time, program variable `x` (stored in the env cell) has the integer value 5 (stored in the memory address given by &x in the heap cell). The symbol tv is a semantic construct aimed to encapsulate typed values. Variables representing symbolic memory addresses are written in sans-serif font preceded by the & symbol.

The semantic rules in $\mathbb{K}$ state how configurations (terms) evolve throughout the computation. Similarly to configurations, rules can also be graphically represented and are split into two levels. Changes in the current configuration (which is shown in the upper level) are explicitly represented by underlining the part of the configuration that changes. The new value that substitutes the one that changes is written below the underlined part.

As an example, consider the KernelC rule for assigning a value V of type T to the variable X. This rule uses three cells: k, env, and heap.

$$\langle\ \underline{X = \mathrm{tv}(T, V)}\ \cdots\ \rangle_{\mathsf{k}} \langle\ \cdots\ X \mapsto \&\mathsf{X}\ \cdots\ \rangle_{\mathsf{env}} \langle\ \cdots\ \&\mathsf{X} \mapsto\ \underline{\phantom{\mathrm{tv}(T, V)}}\ \cdots\ \rangle_{\mathsf{heap}}$$
$$\mathrm{tv}(T, V) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \_ \quad \mathrm{tv}(T, V)$$

The rule states that, if the next pending computation consists of an assignment $X = \mathrm{tv}(T, V)$, then we look for X in the environment $(X \mapsto \&\mathsf{X})$ and we update the associated mapping in the memory with the new value V of type T $(\mathrm{tv}(T, V))$. The underscore _ represents that the actual value that is mapped to the address &X is irrelevant. The value $\mathrm{tv}(T, V)$ is kept at the top of the stack since it might be used in the evaluation of a bigger expression. The rest of the cell's content in the rule does not undergo any modification (this is represented by the ⋯ card). This reveals a useful feature of $\mathbb{K}$, known as *configuration abstraction*: «rules only need to mention the minimum part of the configuration that is relevant for their operation».

For symbolic reasoning, $\mathbb{K}$ uses a particular class of first-order formulae with equality (encoded as boolean non-ground terms with constraints over them). These formulae, called *patterns*, specify those configurations that match the pattern algebraic structure and that satisfy its constraints. For instance, the pattern

$$\left\langle \begin{array}{c} \langle \mathrm{tv}(int, 0) \rangle_{\mathsf{k}} \\ \langle \cdots \ \mathtt{x} \mapsto \&\mathsf{x}, \mathtt{s} \mapsto \&\mathsf{s} \ \cdots \rangle_{\mathsf{env}} \\ \langle \cdots \ \&\mathsf{s} \mapsto (\mathtt{size} \mapsto ?\mathsf{s.size}, \mathtt{capacity} \mapsto ?\mathsf{s.capacity}) \ \cdots \rangle_{\mathsf{heap}} \end{array} \right\rangle_{\mathsf{cfg}}$$
$$\left\langle \ \&\mathsf{s} \neq \mathsf{NULL} \wedge ?\mathsf{s.size} \geq ?\mathsf{s.capacity} \ \right\rangle_{\mathsf{path\text{-}condition}}$$

specifies the configurations satisfying that: 1) the k cell only contains the integer value 0 (i.e., it corresponds to a final state with return value 0); 2) in the env cell, program variable `x` (in typographic font) is associated to the memory address &x, while `s` is bound to the pointer &s; and 3) in the heap cell, the field `size` of (the data structure pointed by) &s (resp. its `capacity` field) contains the symbolic value[4] ?s.size (resp. ?s.capacity). Additionally, &s is not null and the value of its `size` field is greater than or equal to its `capacity` field value.

Since patterns allow logical variables and constraints over them, by using patterns, the $\mathbb{K}$ execution principle (which is based on term rewriting) becomes *symbolic execution*. Unlike concrete execution, where the path taken is determined by the input, in symbolic execution the program can take any feasible path and each possible path is associated to a *path condition*, which represents the conditions that input values have to satisfy in order to follow that path. The path condition is formed by constraints that are gathered along the path taken by the execution to reach the current program point, so each symbolic execution path stands for many actual program runs (in fact, for exactly the set of runs whose concrete values satisfy the logical constraints).

Symbolic execution in $\mathbb{K}$ relies on an automated transformation of $\mathbb{K}$ configurations and $\mathbb{K}$ rules into corresponding symbolic $\mathbb{K}$ configurations (i.e., patterns) and symbolic $\mathbb{K}$ rules that capture all required symbolic ingredients: symbolic values for data structure fields and program variables; path conditions that constrain the variables in cells; multiple branches when a condition is reached during execution, etc [5]. The transformed, symbolic rules define how symbolic configurations are rewritten during computation. Roughly speaking, by symbolically executing a program statement, the configuration cells are updated by mapping fields and variables to new symbolic values that are represented as symbolic expressions, while the path conditions (stored in a new path-condition cell) are correspondingly updated at each branching point.

In [2], an inference procedure for KernelC programs was defined using the $\mathbb{K}$ symbolic execution infraestructure described above. In order to avoid the exponential blow-up that is inherent to path enumeration, the symbolic procedure of [2] follows the standard approach of exploring loops up to a specified number of unfoldings. This ensures that symbolic execution ends for all explored paths, thus delivering a finite (partial) representation of the program behavior. Given a method call $m(args)$ and an initial path condition ϕ, and assuming an unspecified unrolling bound for loops, we denote by $\mathrm{SE}(m(args)\{\phi\})$ the symbolic execution of method m with input arguments $args$ as described in [2], which returns the set of leaves (patterns) of the symbolic execution tree

[4]Symbolic values are in sans-serif font preceded by a question mark.

for m under the constraints given by ϕ. For any function f, by $f(args)\{\phi\}$ we represent the $\mathbb{K}$ pattern $\langle\langle f(args)\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}\langle\phi\rangle_{\mathsf{path\text{-}condition}}$ that is built by inserting the call $f(args)$ at the top of the k cell and by initializing the path condition cell with ϕ.

4 Improving Symbolic Execution in $\mathbb{K}$

In this section, we extend $\mathbb{K}$'s symbolic execution machinery with lazy initialization techniques and abstract subsumption checking in order to support the synthesis of contracts for methods that require refined loop finitization together with C pointer dereference and initialization.

4.1 Lazy initialization

Structured data types (`struct`) in C are aggregate types that define non-empty sets of sequentially allocated *member objects*[5], called fields, each of which has a name and a type. In our symbolic setting, pointer arithmetics and memory layout of C programs are abstracted by: 1) operating with *symbolic addresses* instead of concrete addresses; and 2) mapping each structure object into a single element of the heap cell that groups all object fields (and associated values). A specific field is then accessed by combining the identifier of the structure object with the name of the field, mimicking how the concrete access would be done in C (i.e., `s->elems`).

Symbolic execution for complex data uses lazy initialization to avoid requiring any a priori bound on the size of the input structures [4]. This is because the symbolic execution of a method that takes structurally complex inputs starts with inputs that have uninitialized fields of reference types, and these fields need to be (lazily) initialized when they are (first) accessed during the method's symbolic execution. We adapt the lazy initialization approach of [24] to our setting as follows: when a symbolic address (or address expression) is accessed for the first time, two cases are considered: the case in which the memory stores a null pointer, and the case in which the memory is initialized and it stores an object of its respective type. This implies that the mapping in the heap cell is updated by assigning a new symbolic value (given by the very name of the symbolic address of the accessed field) that symbolically represents the assumptions made on the dynamic data structure. In order to deal with cyclic data structures, like circular lists and lassos, a third possibility needs to be considered: the case in which the symbolic memory references an already existing object in the heap (*aliasing*). Since this generates a new path for *every* single object of the same type existing in the memory heap, in order to avoid state blow-up we enable lazy initialization to consider aliasing only on demand.

Example 3 *The main idea of lazy initialization is graphically represented in Figure 3. Consider the execution state that is depicted on the left-hand side of*

[5] An object in C is a region of data storage in the execution environment.

the figure, where n *points to an uninitialized memory address (represented by a cloud), and assume that the program instruction to be executed next accesses* n *for reading (e.g.,* `if(n != NULL) { ... }`*). When the expression involving uninitialized memory is evaluated, two branches are generated in the symbolic execution tree as depicted on the right-hand side of the figure: in the first branch (upper picture),* n *points to null, and, in the the second branch (lower picture),* n *points to a full-fledged object of the corresponding type (*`struct lnode` *in this example). Note that only the primitive type fields (*`value`*) of the instantiated object are initialized, whereas the reference type fields (*`next`*) point to uninitialized memory.*

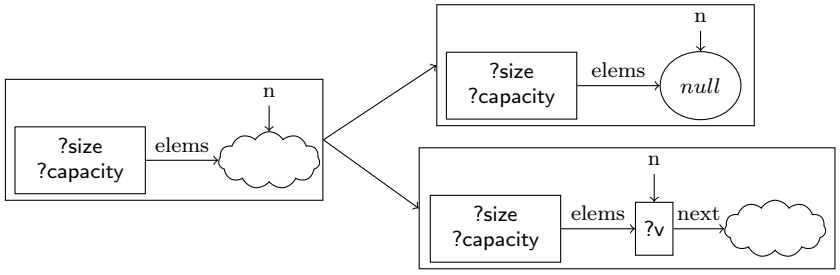

Figure 3: Lazy initialization example.

To keep track of the constraints that are introduced by the lazy initialization, a new cell $\langle\rangle_{\text{init-heap}}$ is added to the configuration that represents the initialization assumptions on the heap memory at a given program point. In other words, at every leaf of the symbolic execution tree, the init-heap cell records the constraints that define the set of initial heaps that lead the execution to reach the given final symbolic configuration.

4.2 Abstract subsumption

Symbolic execution traditionally undergoes non-termination problems in the presence of loops or recursion: the exhaustive exploration of all program paths is generally unaffordable because the search space may be infinite and, consequently, the number of symbolic execution paths may be unbounded. This situation arises when the end of a loop or a recursive call depends on unconstrained properties over the symbolic data.

Different approaches have been proposed in the literature to overcome this problem or mitigate its effects. A classical solution (used in [1, 2]) is to establish a bound to the depth of the symbolic execution tree by specifying the maximum number of unfoldings for each loop and recursive function. As a better approach, the abstract subsumption approach of [4] determines the length of the symbolic execution paths in a dynamic way. Intuitively, symbolic execution with abstract subsumption checking proceeds as standard symbolic execution, except that, before entering a loop, it is checked that the current (abstract) state has not

already been explored; otherwise, the execution of the loop stops. Supporting this check does not require whole execution paths to be recorded; only symbolic states that correspond to the evaluation of loop guards need to be recorded and checked for subsumption.

4.2.1 A constrained representation of symbolic states

In order to formalize the notion of state subsumption and abstract state subsumption, we first introduce a simple notion of symbolic heap and symbolic state, similarly to [4].

Definition 4 (Symbolic heap) *A symbolic heap is a graph $H = (N, E)$ where N is a set of nodes and E is a set of edges. The set of nodes N is then split into $N = O \uplus R \uplus U$ with $U = \{null, \text{uninit}\}$ where:*

- *O is the set of initialized heap objects, with associated type $\tau(o)$, and a corresponding set F of field names;*
- *R is the set of reference variables in the program; and*
- *null and* uninit *are distinguished nodes used to represent null objects and uninitialized memory positions, respectively.*

Furthermore, the set E of edges is itself split into $E = E_R \uplus E_F$, where

- *$E_R :: R \times (O \cup U)$, i.e., edges that connect reference variables with memory objects.*
- *$E_F :: O \times F \times (O \cup U)$, i.e., edges that connect two objects through one of the reference fields of the* origin *node.*

For example, the edge $(r, o) \in E_R$ denotes that program variable r of reference type points to the object o, while the edge $(o_1, f, o_2) \in E_F$ means that pointer field f of object o_1 references to object o_2.

Now we are ready to introduce the symbolic states.

Definition 5 (Symbolic state) *A symbolic program state is defined as a tuple $S = \langle Cf, i \rangle$, where: the first component $Cf = \{H, \sigma, \phi\}$ is a symbolic program configuration with symbolic heap H, symbolic valuation σ, and extended* path-condition *ϕ; and the second component i is a simplistic program counter that corresponds to the line number in the source code of the subsequent instruction to be executed, or the* `return` *statement if the configuration Cf is final.*

The valuation of a symbolic state σ is split into the valuation of structured objects in the heap (denoted σ^h) and the valuation of primitive variables (denoted σ^x).

Definition 6 (Symbolic valuation) *A symbolic valuation is a mapping of the form $\sigma = \sigma^h \uplus \sigma^x$ where σ^h is the evaluation mapping for the non-reference fields of nodes and σ^x is the evaluation mapping for the non-reference program variables (stored in the* env *cell). That is, $\sigma^h = \{e^n_f \mapsto v \mid f$ is a primitive type field of node n and v is a value of the corresponding type$\}$ and $\sigma^x = \{e_x \mapsto v \mid x$ is a program variable of primitive type T and v is a value of type $T\}$.*

Note that the reference fields, reference variables, and objects are part of the *shape* of the heap and are represented in the H component of the state.

Moreover, each symbolic state $S = \langle Cf, i\rangle$ has an associate *state constraint* $SC(S)$, which is given by the conjunction of all constraints over the symbolic values of primitive-type variables and structure fields expressed in the env cell, the heap cell (represented by the valuation σ), and the extended path condition ϕ in Cf that conjuncts the constraints on primitive input variables (given by the path-condition cell) and the the heap constraints imposed by the lazy initialization (given by init-heap cell).

In the following, we adopt a logical constraint representation $\widehat{\sigma}$ for the valuation σ, with $\widehat{\sigma} = \{x = x\sigma | x \in Dom(\sigma)\}$. We also denote by $\mathcal{V}$ the set of all variables in the domain of the valuations σ and $\widehat{\sigma}$, by $\mathcal{V}\restriction_o$ the restriction of $\mathcal{V}$ to the set of variables e^o_f associated to the fields of o, and by $\mathcal{V}\restriction_O = \bigcup_{o\in O} \mathcal{V}\restriction_o$ the natural extension to object sets. Similarly, we also represent the restriction of the mappings in σ to the fields of o as $\sigma\restriction_o$.

We let *State* denote the set of all symbolic program states. Intuitively, a symbolic state S represents the set of all program states s whose concrete values satisfy the symbolic constraints in S. We also say that the state s *matches* the symbolic state S. Similarly, we say that the program heap h *matches* the symbolic heap H if its concrete shape and values satisfy the symbolic constraints in H.

4.2.2 Symbolic subsumption

Let us formalize the notion of symbolic state subsumption by leaning on a simpler notion of symbolic heap subsumption.

Definition 7 (Symbolic Heap subsumption) *Given two symbolic heaps H_1 and H_2, we say that H_1 subsumes H_2, written $H_2 \sqsubseteq H_1$, if the set of program heaps that match H_1 includes the set of program heaps that match H_2.*

Formally, given two program heaps $H_1 = (N_1, E_1)$, with $N_1 = \{O_1 \uplus R_1 \uplus U_1\}$, and $H_2 = (N_2, E_2)$, with $N_2 = \{O_2 \uplus R_2 \uplus U_2\}$, the subsumption relationship $H_2 \sqsubseteq H_1$ is given by: $H_2 \sqsubseteq H_1 \leftrightarrow \forall n_2 \in (N_2 \backslash R_2), \forall r \in R_1, \exists n_1 \in (N_1 \backslash R_1)$ such that

$$corresponding\_root(r, n_1, n_2) \wedge compatible\_shape(n_1, n_2)$$

where

$$corresponding\_root(r, n_1, n_2) = \begin{cases} true & \text{if } (r, n_1) \in E_1^+ \text{ implies } (r, n_2) \in E_2^+ \\ false & \text{otherwise} \end{cases}$$

$$compatible\_shape(n_1, n_2) = \begin{cases} true & \text{if } \begin{cases} n_1 = uninit \vee \\ n_1 = null \wedge n_2 = null \vee \\ n_1 \in O_1 \wedge n_2 \in O_2 \wedge \tau(n_1) = \tau(n_2) \end{cases} \\ false & \text{otherwise} \end{cases}$$

Roughly speaking, for every node n_2 in the less general heap H_2, there must exist a corresponding node n_1 in H_1 such that, for all program variables r (also called *roots*) for which the pair (r, n_1) is contained in the transitive closure of the set of edges E_1, the pair (r, n_2) also belongs to the transitive closure of E_2. This means that, in both heap graphs H_1 and H_2, there is a path starting from the root r that eventually reaches the nodes n_1 and n_2, respectively. This is represented in our formalization by the predicate *corresponding_root*. In addition, the shapes of n_1 and n_2 must be *compatible*, which is represented by the predicate *compatible_shape*. Two nodes n_1 and n_2 are considered to be compatible whenever they are not roots (i.e., $n_1 \in (N_1 \backslash R_1) \wedge n_2 \in (N_2 \backslash R_2)$), and one of the following conditions holds:

- $n_1 =$ uninit, which means that any node in n_2 is subsumed by the most general type of node uninit in the more general heap H_1. Note that, since the subsumption relation is not commutative, a uninit node in H_2 could only be compatible with a uninit node in H_1.
- $n_1 = null \wedge n_2 = null$; that is, null nodes are only compatible with other null nodes. Hence, shapes are not compatible if only one of the nodes is *null*.
- n_1 and n_2 are non-null, non-uninit objects of the same structured type $\tau(n_1) = \tau(n_2)$.

In the case when the requirements above are not fulfilled, we conclude $H_2 \not\sqsubseteq H_1$. This notion was first formulated in [4] in an algorithmic style. However, our formalization gets rid of complex data structures and any implementation details, so it is much simpler and closer to the related, stronger notion of graph simulation [30].

For subsumption checking, a symbolic heap H is represented as a directed graph whose nodes are either objects (subset O of H in the formalization above) or reference variables (subset R), also called pointers. The graph edges (subset E) connect either a reference with an object (meaning that the program variable points to that memory object) or two object nodes (meaning that a reference field of the origin object points to the destination object). To make the visualization of symbolic heaps easier, we adapt to our symbolic setting a classical graphical representation for heaps based on UML object diagrams [21], where

null nodes are rendered as ellipses, uninitialized nodes are drawn as clouds, and references are depicted as arrows.

Now let us formalize symbolic state subsumption as follows.

Definition 8 (Symbolic State subsumption) *Given two symbolic states* $S_1 = \langle\{H_1, \sigma_1, \phi_1\}, i_1\rangle$ *and* $S_2 = \langle\{H_2, \sigma_2, \phi_2\}, i_2\rangle$*, we say that* S_1 *subsumes* S_2*, written* $S_2 \sqsubseteq S_1$*, if: 1)* $i_1 = i_2$*, 2)* $H_2 \sqsubseteq H_1$*, and 3)* $SC_2 \Rightarrow SC_1$*, i.e.,* $(\widehat{\sigma}_2 \wedge \phi_2) \Rightarrow (\widehat{\sigma}_1 \wedge \phi_1)$*.*

Example 9 *Consider the following symbolic states* S_1 *and* S_2 *(depicted in Figure 4) with program counter* 29*, such that* $SC_2 \Rightarrow SC_1$*:*

$S_1 = \langle\{H_1, \{e^1_{siz} \mapsto \mathsf{?size}, e^1_{cap} \mapsto \mathsf{?capacity}, e^2_{val} \mapsto \mathsf{?v_0}, e_x \mapsto \mathsf{?x}, e_{fd} \mapsto 0\}, \mathsf{?size} < \mathsf{?capacity}\}, 29\rangle$

$S_2 = \langle\{H_2, \{e^1_{siz} \mapsto \mathsf{?size}, e^1_{cap} \mapsto \mathsf{?capacity}, e^2_{val} \mapsto \mathsf{?v_0}, e^3_{val} \mapsto \mathsf{?v_1}, e_x \mapsto \mathsf{?x}, e_{fd} \mapsto 0\},$
$\mathsf{?size} < \mathsf{?capacity} \wedge \mathsf{?v_0} \neq \mathsf{?x}\}, 29\rangle$

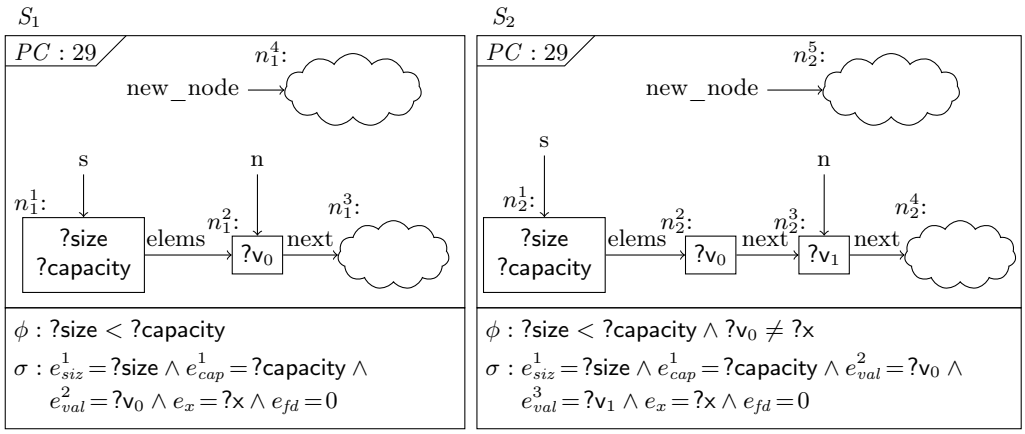

Figure 4: Symbolic state subsumption example.

Let us describe how the heap subsumption checking is performed for this execution scenario. For each of the non-root nodes in H_2 *(which we labeled to be easily identified), we check the existence of at least one corresponding node in* H_1 *that can be reached from the exact same roots and have compatible shapes. This way, we can find that node* n^2_2 *from* H_2 *(the object of type* `struct lnode` *that holds the symbolic value* $\mathsf{?v_0}$*) has a compatible shape with and only with the node* n^2_1 *in* H_1*. However, while* n^2_1 *can be reached from the roots* s *and* n*,* n^2_2 *cannot be accessed through* n *in any number of steps. This means that a correspondence cannot be established for node* n^2_2*, and since there is a node for which the relation corresponding_root does not hold, we conclude that* $H_2 \not\sqsubseteq H_1$*. Consequently,* $S_2 \not\sqsubseteq S_1$ *for this example.*

4.2.3 Abstract subsumption

In order to ensure termination and improve scalability of symbolic execution with subsumption, we enhance symbolic state subsumption by means of abstract interpretation [12]. We abstract both primitive domains and heaps by using a source-to-source abstraction function $\alpha : State \rightarrow State$ that essentially consists of a heap shape transformation that collapses two or more nodes into a *summary node* (i.e., a single node which corresponds to one or more individual nodes in a concrete state represented by the abstract state [28]). The valuation for each primitive field in the summary node is given by the disjunction of the corresponding valuations for this field in the original nodes.

Definition 10 (Summary node) *A summary node $n^\sharp$ represents a finite set of uninterruptedly linked nodes in the heap graph such that, for each node n in the set,*

- $n \neq null$
- $n \neq$ uninit
- $n \notin \{m \,|\, (r, m) \in E_R\}$*, i.e., n is not pointed to by any reference variable, and*
- $\nexists n_1, n_2, n_1 \neq n_2 \, s.t. \, \{(n_1, f_1, n), (n_2, f_2, n)\} \in E_F$*, i.e., n is not pointed to by two or more different reference fields.*

Roughly speaking, nodes can be collapsed when they are in a sequence and can only be accessed by traversing all of their predecessors. The resulting summary node is then consistently re-connected to the rest of nodes in the heap.

Example 11 *Figure 5 illustrates shape abstraction for the given state. The circled nodes are abstracted into a summary node. Then, the first node of the list points to this new summary node and, in turn, the summary node points to the node referenced by* n*. Moreover, the valuation for the field* `value` *of the summary node (identified by e^2_{val}) is $e^2_{val} = \mathsf{?v}_0 \vee e^2_{val} = \mathsf{?v}_1$.*

Definition 12 (Abstract state) *Given the symbolic state $S = \langle \{H, \sigma, \phi\}, i \rangle$, with $H = (N, E)$, $N = O \uplus R \uplus U$ and $E = E_R \uplus E_F$, we define the corresponding abstract state $S^\sharp = \alpha(S)$, where the abstraction function $\alpha(S)$ is formalized as follows.*

Consider the set $MS(O)$ of maximal ordered subsets $\{o_1, \ldots, o_n\}$ of O whose elements o_i are uninterrupted linked nodes that form a summarizable segment (i.e., none of them is pointed to by a program variable nor by more than one object). Formally:

$$MS(O) = \{M = \{o_1, \ldots, o_n\} \subseteq O \,|\, \forall o \in M \; not\_rooted(o) \wedge not\_heap\_shared(o) \wedge o_1 \preceq o \preceq o_n\}$$

where the predicate $not\_rooted(o)$ states that o is not pointed to by any root node; the predicate $not\_heap\_shared(o)$ assures that there are not two different

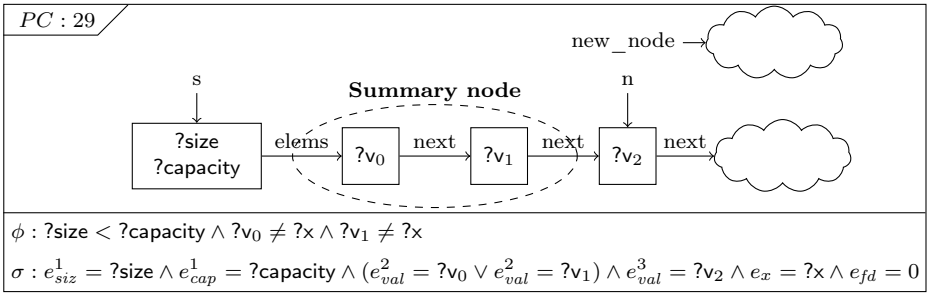

Figure 5: Shape abstraction example.

nodes o_x and o_y pointing to o through one of their reference fields; and the ordering relation $\preceq$ is given by the precedence of connected nodes in the heap graph, that is, $o_1 \preceq o$ means that either $o_1 = o$, or there is an edge from o_1 to o. Whenever all of the three conditions hold, the nodes in M are summarizable into a single abstract node. Formally:

$$not\_rooted(o) = \begin{cases} true & \text{if } \nexists r \in R \text{ such that } (r, o) \in E_R \\ false & \text{otherwise} \end{cases}$$

$$not\_heap\_shared(o) = \begin{cases} true & \text{if } \forall o_x, o_y \in O\, \{(o_x, \_, o), (o_y, \_, o)\} \subseteq E_F \text{ implies } o_x = o_y \\ false & \text{otherwise} \end{cases}$$

Then, we define the abstract state $S^\sharp$ by $\alpha(S) = \langle \{H^\sharp, \sigma^\sharp, \phi\}, i \rangle$, where:

$$\begin{array}{l} H^\sharp = (\{O^\sharp \uplus R \uplus U\}, \{E_R \uplus E^\sharp_F\}) \\ O^\sharp = (O \setminus \bigcup_{M \in MS(O)} M) \cup \bigcup_{\{o_1,\ldots,o_n\} \in MS(O)} (o_1, \ldots, o_n) \\ E^\sharp_F = (E_F \setminus \bigcup_{M \in MS(O)} E^M_F) \cup \\ \qquad \bigcup_{\{o_1,\ldots,o_n\} \in MS(O)} \exists n_i, n_e \in N, f_i \in F_{n_i}, f_n \in F_{o_n} \{(n_i, f_i, (o_1, \ldots, o_n)), ((o_1, \ldots, o_n), f_n, n_e)\} \\ \qquad \text{where } E^M_F = \bigcup_{o \in M} \{(o, \_, \_) \in E_F\} \cup \{(\_, \_, o) \in E_F\} \\ \sigma^\sharp = (\widehat{\sigma \restriction_{\mathcal{V}^-_{MS(O)}}}) \bigwedge_{M \in MS(O)} \bigvee_{o \in M} (e = v) \in \widehat{(\sigma \restriction_o)}, \text{ where } \mathcal{V}^-_{MS(O)} = \mathcal{V} \setminus \mathcal{V} \restriction_{\bigcup_{M \in MS(O)} M} \end{array}$$

In other words, the new abstract state $S^\sharp$ is constructed in the following way. First, for all $M = \{o_1, \ldots, o_n\} \in MS(O)$, we take out from O the individual nodes $o_1, \ldots, o_n$ of M and add a single abstract node $(o_1, \ldots, o_n)$ to the set $O^\sharp$ of abstract memory objects.

With regard to the edges, since we are transforming list segments into individual nodes, we need to reformulate the incoming edge to the first node of each segment, o_1, as well as the outgoing edge from the last node of the segment, o_n, for consistency. The identifiers n_i and n_e represent the arbitrary nodes that may be respectively pointing to o_1 and pointed from o_n through the corresponding

reference fields f_i and f_n. Furthermore, in order to fully consider the summary nodes as individuals, we also need to dispense with the inner references between nodes that have been abstracted (represented by E_F^M). This way, we substitute all the original references in E_F to the concrete nodes $o_1, \ldots, o_n$ with just two new ones incoming to and outgoing from each abstract node $(o_1, \ldots, o_n)$. Note that the sets of roots and null and uninitialized objects, R and U, as well as the program variable references E_R from the original heap H of S remain unaltered.

As for the heap valuation σ, a new abstract valuation $\sigma^\sharp$ is built that preserves the equations regarding the heap nodes that are not affected by the summary node abstraction (i.e., the restriction of the set of variables in the original valuation $\mathcal{V}$ to the ones that are not related to the objects in the subsets of $MS(O)$, given by $\mathcal{V}^-_{MS(O)}$). For each subset M of $MS(O)$ (i.e., for each summary node), it also contains a disjunction of all the equations that correspond to the objects of M in the original valuation σ. These disjuncts represent the individual valuations of the summary nodes since they are now treated as single objects.

The abstract state $S^\sharp$ is ultimately built by joining together the abstract ingredients $H^\sharp$ and $\sigma^\sharp$ with the path condition ϕ of S. With this notion of state abstraction, let us finally define the abstract subsumption relation between symbolic states as follows.

Definition 13 (Abstract subsumption) *Given two symbolic states S_1 and S_2, the abstract symbolic subsumption relation $S_2 \sqsubseteq^\sharp S_1$ is given by $S_2^\sharp \sqsubseteq S_1^\sharp$.*

The abstract subsumption approach improves the classical solution (applying thresholds to limit the unrolling of loops and the number of recursive calls) in several ways. Since the completeness of the symbolic analysis is highly dependent on the chosen threshold, it is not generally possible to ascertain the optimal number of iterations that *subsume* all possible behaviors by inspecting the source code. Applying abstract subsumption alleviates the user from the burden of determining the best threshold for each loop in each problem, which often becomes unaffordable. Moreover, it usually covers much more significant behaviors and avoids exploring unnecessary program paths due to excessive unrolling of loops.

4.2.4 Symbolic execution with abstract subsumption

The symbolic execution with abstract subsumption and lazy initialization (which we call *abstract symbolic execution*, for short) of a given method m with arguments *args* and initial path condition ϕ, written $\mathrm{SE}^\sharp(m(args)\{\phi\})$, is defined as an approximation of the SE mechanism of [2] where, each time a symbolic state S_2 is visited that corresponds to a recursive call or loop guard evaluation with the same program counter as a previously visited state S_1, the abstract subsumption $S_2 \sqsubseteq^\sharp S_1$ is checked. If there is more than one state with the same program counter as S_2, the most recent one (in the same branch) is checked first. If the test succeeds, a final call or loop iteration is then performed such

that the heap is transformed (by means of lazy initialization) into an instance that makes the recursion or iteration end. This is essential to ensure that every final state of the symbolic execution tree corresponds to a real execution.

Example 14 *The uncontrolled symbolic execution* SE *of the function* `insert` *from Example 1 generates an infinite state space. In contrast, its* $\mathrm{SE}^\sharp$ *terminates after three iterations of the loop. Figure 6 illustrates a fragment of the symbolic execution tree for* `insert(s,x)` *where the subsumption between two abstract states occurs. The state* S_{10} *corresponds to the execution point where the loop guard* `n != NULL` *(with program counter 29) is to be checked for the second time. This requires evaluating* n*, which points to an uninitialized node; hence, lazy initialization is applied. This results in two children nodes,* S_{11} *and* S_{13}*, with the same program counter because the guard has not been evaluated yet. The left child* S_{11} *corresponds to the case when the loop guard is not satisfied (the object pointed to by* n *is null); thus the loop is exited. On the contrary, the right child* S_{13} *represents entering the loop iteration since the guard evaluates to true (the object pointed to by* n *is an initialized object of the corresponding type). Program counter 29 is reached again at state* S_{18} *in the right branch after lazy initialization, and then the abstract subsumption check* $S_{18} \sqsubseteq^\sharp S_{13}$ *succeeds.*

Assuming that appropriate abstractions are defined to ensure termination of $\mathrm{SE}^\sharp$, the call $\mathrm{SE}^\sharp(f(args)\{\phi\})$ returns the set of final patterns obtained from the abstract symbolic execution of the pattern $f(args)\{\phi\}$ (i.e., the leaves of the deployed abstract symbolic execution tree). A new, (abstract subsumption) cell $\langle\rangle_{\mathsf{aSubFlag}}$ identifies with a *true* value those final abstract configurations ending any branch that was folded (at some intermediate configuration) by the application of abstract subsumption. This is used for the inference process to distinguish the inferred axioms that are ensured to hold (because no approximation was done to extract them) from the *plausible*, candidate axioms that are not demonstrably correct because of the potential precision loss caused by the abstraction. Furthermore, in order to obtain the set of locations that may be affected by the execution of f (the component $\mathcal{L}$ of the contract), those locations have to be harvested during symbolic execution. To this end, we add a new cell $\langle\rangle_{\mathsf{locations}}$ to the symbolic engine of $\mathbb{K}$. Then, whenever a program location is overwritten, it is recorded in the new cell locations. At the end of the symbolic execution, the program locations recorded for each final configuration in their respective locations cells are all joined by union to obtain a global set with every program location that is potentially modifiable by a call to the function f. Thus, assignable locations are obtained as a by-product of the symbolic execution.

5 The Synthesis Algorithm

Let us introduce the basic notions that we use in our formulation. Given an input program P, we distinguish the set of observers $\mathcal{O}$ and the set of modifiers $\mathcal{M}$ in P. A function can be considered to be an *observer* if it explicitly returns

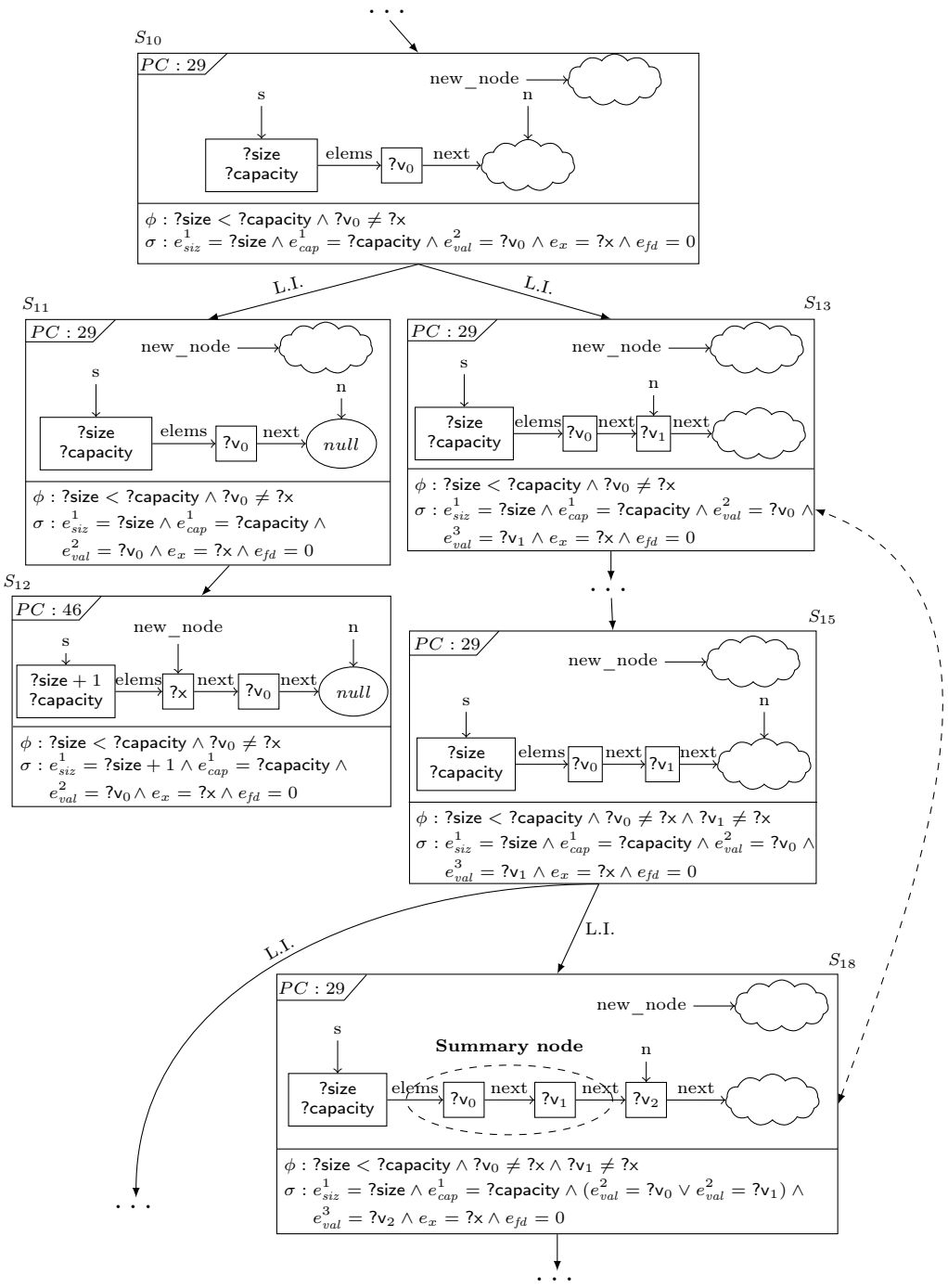

Figure 6: Fragment of the abstract symbolic execution of `insert(s,x)`.

a value, whereas any method can be considered to be a *modifier*. Thus, the set $\mathcal{O} \cap \mathcal{M}$ is generally non-empty.

5.1 The basic inference algorithm

Our contract inference methodology is formalized in Algorithm 1. Let $\overline{a_n}$ denote the list of fresh symbolic variables $a_1, \ldots, a_n$. First, the *modifier* method of interest m is symbolically executed with argument list $\overline{a_n}$ and empty path constraint `true`, and the set $\mathcal{F}$ of final configurations is retrieved from the leaves of the abstract symbolic execution tree. For each final configuration, the corresponding path condition ϕ is simplified by calling the automated theorem prover Z3 to avoid redundancies and simplify the analysis.

Algorithm 1 Contract Synthesis

Input: $m \in \mathcal{M}$: a modifier function with arity n
Output: C : a specification contract for m
Output: $Q^{\sharp\prime}$: a set of candidate contract axioms

1: $root := m(\overline{a_n})$
2: $\mathcal{F} :=$SE$^{\sharp}(root\{\texttt{true}\})$
3: $P := false\ \ Q := true\ \ Q^{\sharp} := true\ \ \mathcal{L} := \emptyset$
4: **for all** $F \in \mathcal{F}$, with $F = \langle\langle v \rangle_{\mathsf{k}} \langle \varphi \rangle_{\mathsf{init\text{-}heap}} \cdots \rangle_{\mathsf{cfg}} \langle \phi \rangle_{\mathsf{path\text{-}condition}} \langle \sharp \rangle_{\mathsf{aSubFlag}} \langle L \rangle_{\mathsf{locations}}$ **do**
5: $\quad \phi' := Z3\_simplify(\phi)$
6: $\quad p := explain(I, \overline{a_n})$, where $I = \langle\langle root \rangle_{\mathsf{k}} \langle \varphi \rangle_{\mathsf{heap}} \cdots \rangle_{\mathsf{cfg}} \langle \phi' \rangle_{\mathsf{path\text{-}condition}}$
7: $\quad q := explain(F', \overline{a_n}) \wedge (ret = v)$, where $F' = F[\langle \phi' \rangle_{\mathsf{path\text{-}condition}}]$
8: $\quad ax_{(I-F)} := (p \Rightarrow q)$
9: $\quad$ **if** $\sharp$ **then** $Q^{\sharp} := Q^{\sharp} \cup \{ax_{(I-F)}\}$ **else** $Q := Q \cup \{ax_{(I-F)}\}$
10: $\quad \mathcal{L} := \mathcal{L} \cup \{L\}$
11: **end for**
12: $(Q', Q^{\sharp\prime}) := refine(Q, Q^{\sharp})$
13: $P := \bigvee_{(p \Rightarrow q) \in Q'} p$
14: $C :=< P, Q', \mathcal{L} >$
15: **return** $C, Q^{\sharp\prime}$

After initializing the contract components (line 3), we proceed to compute one axiom for each (abstract) symbolic configuration F in $\mathcal{F}$. First, the path condition ϕ is simplified into ϕ' by using the SMT Solver Z3 to reduce the complexity of the symbolic execution. Next, the premise p of the axiom $p \Rightarrow q$ is computed (line 6) by means of the function $explain(I, as)$ given in Algorithm 2. This function receives as argument the pattern I, which expresses the initial symbolic configuration leading to F in the execution tree for m (i.e., an instance of the initial configuration for $m(\overline{a_n})$ that is obtained by assuming the constraints φ and ϕ' in the corresponding init-heap and path-condition cells of I). The consequent q of the axiom is then computed (line 7) as the conjunction

of $ret = v$, which specifies the return value v of the method m as recorded in the k cell of F, and the equations given by $explain(F', \overline{a_n})$, where the configuration F' is obtained by replacing ϕ with ϕ' in F. At this point, the initial and final configurations have been characterized in terms of the observers by invoking the function *explain* and we can build the axiom (line 8). By using the abused notation $ax_{(I-F)}$, we implicitly record the initial and final configurations I and F from which a given axiom is derived.

It is important to note that, in the axioms, the different function calls in the antecedent (resp. consequent) of every implication formula are run independently of each other under the same initial configuration. This is achieved by the *explain* algorithm by using the same initial state when considering the different observer functions to explain I and F. This avoids making any assumptions about function purity or side-effects.

Depending on the boolean value of the abstract subsumption flag $\sharp$ in F (line 9), the synthesized axiom $ax_{(I-F)}$ is directly added to the postcondition Q (when $\sharp$ is *false*) or to the conjunction $Q^\sharp$ (when $\sharp$ is *true*) that collects all candidate axioms extracted from branches that contain at least one node that was folded by abstract subsumption.

Note that, due to the under-approximation introduced by abstract subsumption [4], there may be some behaviors (real trace fragments) beyond the abstract folded states that are not captured by the deployed symbolic abstract traces. Therefore, axioms in $Q^\sharp$ could have spurious instances and must be double-checked. We apply a post-processing refinement $refine(Q, Q^\sharp)$ which essentially tries to find those spurious instances and discards the axioms that present them, while keeping in $Q^\sharp$ any axioms that remain overly general (i.e., that might have both true and false instances). We postpone to Section 6 a description of how those remaining candidate axioms can be eventually verified. A further subsumption checking over the resulting sets of axioms is included in the refinement post-processing that purges Q and $Q^\sharp$ from less general axioms in order to obtain a minimal contract that summarizes all the input/output behavior of the modifier method.

When Algorithm 1 terminates, the generated contract C is $< P, Q', \mathcal{L} >$ where: 1) the method precondition P is the disjunction of all axiom premises; 2) the method postcondition is the final, simplified set of correct axioms given by $refine(Q, Q^\sharp)$; and 3) $\mathcal{L}$ records all program locations that are (potentially) modifiable by m. This global set of locations $\mathcal{L}$ is built by joining all the local sets of modified program locations that are recorded in the locations cell of each final configuration.

Let us compute a specification for the `insert` modifier function of Example 1 by applying Algorithm 1.

Example 15 *We first compute* $\mathrm{SE}^\sharp$(`insert(&s,?x){true}`) *with* `&s` *being a symbolic address with initial value* uninit *and with* `?x` *being a symbolic integer value. Since there are no constraints in the initial symbolic configuration, the execution covers all possible initial concrete configurations. Then, the abstract symbolic execution computes 17 final configurations. Figure 7 shows the*

$$\langle \mathrm{tv}(int, 1) \rangle_{\mathsf{k}}$$
$$\langle\ \mathtt{s} \mapsto \&\mathsf{s}, \mathtt{new\_node} \mapsto \&\mathsf{new\_node}, \mathtt{n} \mapsto null, \mathtt{x} \mapsto \&\mathsf{x}, \mathtt{found} \mapsto 1\ \rangle_{\mathsf{env}}$$
$$\left\langle \begin{array}{c} \&\mathsf{s} \mapsto (\mathtt{capacity} \mapsto ?\mathsf{s.capacity}, \mathtt{size} \mapsto ?\mathsf{s.size}, \mathtt{elems} \mapsto \&\mathsf{s.elems}), \\ \&\mathsf{s.elems} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_0, \mathtt{next} \mapsto \&\mathsf{s.elems.next}), \\ \&\mathsf{s.elems.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_1, \mathtt{next} \mapsto \&\mathsf{s.elems.next.next}), \\ \&\mathsf{s.elems.next.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_2, \mathtt{next} \mapsto null), \end{array} \right\rangle_{\mathsf{init\text{-}heap}}$$
$$\left\langle \begin{array}{c} \&\mathsf{s} \mapsto (\mathtt{capacity} \mapsto ?\mathsf{s.capacity}, \mathtt{size} \mapsto ?\mathsf{s.size} + 1, \mathtt{elems} \mapsto \&\mathsf{new\_node}), \\ \&\mathsf{new\_node} \mapsto (\mathtt{value} \mapsto ?\mathsf{x}, \mathtt{next} \mapsto \&\mathsf{s.elems}), \\ \&\mathsf{s.elems} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_0, \mathtt{next} \mapsto \&\mathsf{s.elems.next}), \\ \&\mathsf{s.elems.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_1, \mathtt{next} \mapsto \&\mathsf{s.elems.next.next}), \\ \&\mathsf{s.elems.next.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_2, \mathtt{next} \mapsto null), \\ \&\mathsf{x} \mapsto ?\mathsf{x} \end{array} \right\rangle_{\mathsf{heap}}$$
$$\langle ?\mathsf{s.size} < ?\mathsf{s.capacity} \wedge ?\mathsf{v}_0 \neq ?\mathsf{x} \wedge ?\mathsf{v}_1 \neq ?\mathsf{x} \wedge ?\mathsf{v}_2 \neq ?\mathsf{x} \rangle_{\mathsf{path\text{-}condition}}$$

Figure 7: Final configuration corresponding to the branch where the element is inserted, in Example 15.

final configuration for the case when the element is actually inserted and the `while` *loop stops due to abstract subsumption between the states associated to two consecutive iterations (nodes S_{13} and S_{18} of Example 14). The graphical representation for the init and final heaps in such a final configuration is depicted in Figures 8 and 9, respectively.*

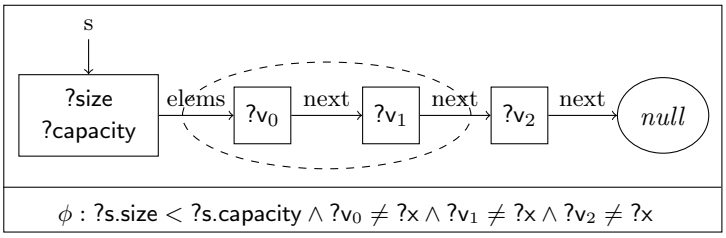

Figure 8: Graphical representation of the initial heap of the configuration in Figure 7.

The circled nodes are the nodes that collapse, as a summary node, in the corresponding abstract state. Roughly speaking, the execution of this path corresponds to the case when the element `x` *(with symbolic value* ?x*) is effectively inserted at the beginning of a list that contains three elements. Thus, the return value (*k *cell) of the call* `insert`(&s, ?x) *is the integer 1 (standing for success); the symbolic (initial) value* ?s.size *of the field* `size` *of* `s` *is increased by 1 and now the field* `elems` *of* `s` *points to an object* &new_node *with* `value` ?x *as the first node of the set. For the sake of simplicity, we omit any configuration components that are irrelevant for the discussion.*

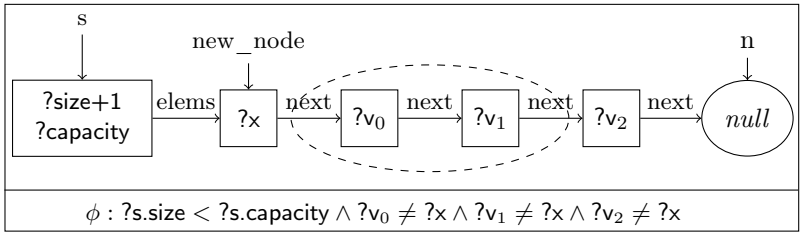

Figure 9: Graphical representation of the final heap of of the configuration in Figure 7.

Let us now describe Algorithm 2, which defines the function $explain(p, as)$. Given a $\mathbb{K}$ configuration p and a list of symbolic variables as, this function describes p by means of an equational conjunction that is obtained by executing the observer functions $o \in \mathcal{O}$ with the constraints in p. Each equation relates the call to an observer function with the symbolic value that the call returns. Intuitively, to express a partial observational abstraction or explanation for (the constraints in) a given state in terms of the observer o, our criterion is that o computes the same symbolic values at the end of all its symbolic execution branches. In the algorithm, $As \sqsubseteq as$ means that the list of elements As is a permutation of a sublist of as.

Algorithm 2 Computing explanations: $explain(p, as)$

Input: p : the $\mathbb{K}$ pattern to be explained (with path condition ϕ)
Input: as : a list of symbolic variables
Output: $eqConj$: conjunction of equations that explain p

1: $eqConj := true$
2: **for all** $o(As) \in \mathcal{C}$, with $As \sqsubseteq as$ **do**
3: $F_o := \mathrm{SE}^\sharp(o(As)\{\phi\})$
4: **if** $\forall(f = \langle\langle v\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \cdots, f' = \langle\langle v'\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \cdots) \in F_o : v \neq v' \rightarrow (v = \text{uninit} \vee v' = \text{uninit})$ **then**
5: **if** $\nexists f_u = \langle\langle \text{uninit}\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \cdots \in F_o$ **then**
6: $eqConj := eqConj \wedge (o(As) = v)$
7: **else**
8: $eqConj := eqConj \wedge (o(As) = fresh\_symbolic\_value())$
9: **end if**
10: **end if**
11: **end for**
12: **return** $eqConj$

Roughly speaking, $explain(p, as)$ explores the universe of observer execution calls $\mathcal{C}$, which consists of all the function calls $o(As)$ that satisfy: 1) $o \in \mathcal{O}$; and 2) the argument list $As \sqsubseteq as$ respects the type and arity of o. Then, for each call $o(As) \in \mathcal{C}$, Algorithm 2 checks whether the symbolic execution of $o(As)$

stemming from an initial configuration that is constrained by the path condition ϕ returns the same value in all the final symbolic configurations. When the call satisfies this requirement, an equation is generated (line 6). Line 8 represents the case when uninit return values are involved and is explained later. In any other case, the observation is inconclusive and no explanation is delivered in terms of the executed observer function. This algorithm runs symbolic execution with abstract subsumption but without lazy initialization as we discuss later. The algorithm finally returns the conjunction of all the explanatory equations previously inferred.

Let us show how we compute the explanation for the final configuration shown in Example 15.

Example 16 *Given the observer functions* `isnull`*,* `isempty`*,* `isfull`*,* `length`*, and* `contains`*, and the symbolic arguments* `&s` *and* `?x`*, The universe of observer calls is* $\mathcal{C} = \{$`isnull(&s)`, `isempty(&s)`, `isfull(&s)`, `length(&s)`, `contains(&s, ?x)`$\}$*. Let us consider, for instance, the call* `contains(&s,?x)` *in detail.*

When we symbolically execute `contains(&s,?x)` *on the final configuration shown in Example 15, we obtain a single final configuration that is shown in Figure 10, whose heap is graphically depicted in Figure 11.*

$$
\begin{array}{c}
\langle\, \mathrm{tv}(int, 1)\rangle_{\mathsf{k}} \\
\langle \mathtt{s} \mapsto \&\mathsf{s}, \mathtt{x} \mapsto ?\mathsf{x}, \mathtt{n} \mapsto \&\mathsf{new\_node}\rangle_{\mathsf{env}} \\
\left\langle \begin{array}{c}
\&\mathsf{s} \mapsto (\mathtt{capacity} \mapsto ?\mathsf{s.capacity}, \mathtt{size} \mapsto ?\mathsf{s.size} + 1, \mathtt{elems} \mapsto \&\mathsf{new\_node}), \\
\&\mathsf{new\_node} \mapsto (\mathtt{value} \mapsto ?\mathsf{x}, \mathtt{next} \mapsto \&\mathsf{s.elems}), \\
\&\mathsf{s.elems} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_0, \mathtt{next} \mapsto \&\mathsf{s.elems.next}), \\
\&\mathsf{s.elems.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_1, \mathtt{next} \mapsto \&\mathsf{s.elems.next.next}), \\
\&\mathsf{s.elems.next.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_2, \mathtt{next} \mapsto \mathit{null}), \\
\&\mathsf{x} \mapsto ?\mathsf{x}
\end{array} \right\rangle_{\text{init-heap}} \\
\left\langle \begin{array}{c}
\&\mathsf{s} \mapsto (\mathtt{capacity} \mapsto ?\mathsf{s.capacity}, \mathtt{size} \mapsto ?\mathsf{s.size} + 1, \mathtt{elems} \mapsto \&\mathsf{new\_node}), \\
\&\mathsf{new\_node} \mapsto (\mathtt{value} \mapsto ?\mathsf{x}, \mathtt{next} \mapsto \&\mathsf{s.elems}), \\
\&\mathsf{s.elems} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_0, \mathtt{next} \mapsto \&\mathsf{s.elems.next}), \\
\&\mathsf{s.elems.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_1, \mathtt{next} \mapsto \&\mathsf{s.elems.next.next}), \\
\&\mathsf{s.elems.next.next} \mapsto (\mathtt{value} \mapsto ?\mathsf{v}_2, \mathtt{next} \mapsto \mathit{null}), \\
\&\mathsf{x} \mapsto ?\mathsf{x}
\end{array} \right\rangle_{\text{heap}} \\
\langle ?\mathsf{s.size} < ?\mathsf{s.capacity} \wedge ?\mathsf{v}_0 \neq ?\mathsf{x} \wedge ?\mathsf{v}_1 \neq ?\mathsf{x} \wedge ?\mathsf{v}_2 \neq ?\mathsf{x}\rangle_{\text{path-condition}}
\end{array}
$$

Figure 10: Final after the execution of `contains(&s,?x)` in Example 16.

Note that the initial heap of the call to the observer `contains(&s,?x)` *corresponds to the final heap of the resulting configuration after the execution of* `insert(&s,?x)` *(the configuration shown in Example 15). Since no observer execution path returns different values and the only return value is the integer 1, then the equation* `contains(s,x)=1` *is generated as a part of the explanation of* p*.*

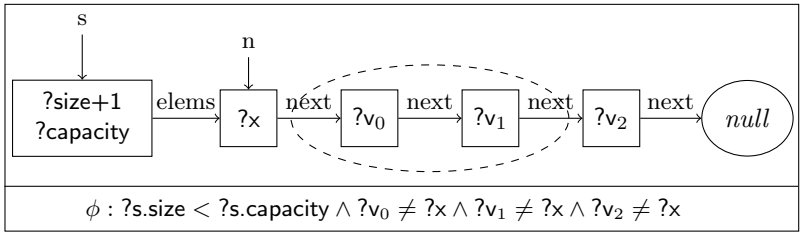

Figure 11: Graphical representation of the heap of the configuration in Figure 10.

As we noted before, lazy initialization is not applied when symbolically executing the observer functions. This is because we want to start from an initial configuration whose dynamic memory satisfies (or is given by) ϕ, and if any uninitialized addresses are expanded by lazy initialization, such an initial configuration (and thus the target of the observation) would be altered. This implies that some final patterns in the symbolic execution trees for the given observer may contain uninit return values, meaning that we know nothing regarding the dynamic memory from that point on. When this occurs (and the rest of the branches either return other uninit values or the same concrete value), the *explain* algorithm generates a conjunct where the observer call is equated to a fresh symbolic value (line 8 in Algorithm 2).

Another particular case regarding state abstraction must be considered when running the observer calls for explanation. Whenever an abstraction is applied to end a loop, the final (abstract) state for the loop somehow *generalizes* the set of symbolic traces that have not been unrolled; i.e., it can be thought of as a representative of the set of program states that have been collapsed. Since the observer calls that are symbolically executed by algorithm *explain* start from such an abstract state when the abstract subsumption flag is *true*, the observer function may try to access summary nodes that represent sets of nodes of unknown length (for instance, while traversing a list). Then, an abstract (symbolic) value is assigned as the partially computed result for the observer and the observer execution proceeds.

Example 17 *Let us compute the result of the observer function* `length(&s)` *on the final state of the symbolic execution trace for* `insert(&s,?x)` *that was pruned by abstract subsumption, shown in Example 15. When* `length` *starts traversing the list at the initial state, it immediately finds a summary node (which includes the nodes with values* $?v_0$ *and* $?v_1$*). Since a summary node actually represents a set of nodes with arbitrary length, the execution computes a symbolic value, say* ?l*, as (the partially accumulated) length. The list traversal then goes on until the execution of the loop ends, producing as a result the equations* `length(s) = ?l + 1` *for the initial state and* `length(s') = ?l + 2` *for the final state.*

5.2 The postprocessing refinement algorithm

The refinement postprocessing $refine(Q, Q^\sharp)$ for method m is formalized in Algorithm 3.

Algorithm 3 Contract refinement algorithm: $refine(Q, Q^\sharp)$

Input: Q : set of demonstrably correct axioms
Input: $Q^\sharp$: set of overly general, candidate axioms
Output: a refined, final set Q' of correct contract axioms, and a refined set $Q^{\sharp\prime}$of candidate axioms

1: $Q_f^\sharp = drop\_duplicates(Q^\sharp)$
2: $Q_s^\sharp = testing\_based\_filtering(Q_f^\sharp)$
3: $(Q', Q^{\sharp\prime}) = subsumption\_filtering(Q, Q_s^\sharp)$
4: **return** $(Q', Q^{\sharp\prime})$

Roughly speaking, we first apply a *drop_duplicates* function that gets rid of any duplicates from the axiom set $Q^\sharp$ and returns the repetition-free set $Q_f^\sharp$. Next, for each candidate axiom $p \Rightarrow q$ in $Q_f^\sharp$, the *testing_based_filtering* function works as follows: 1) a number of test cases (initial configurations) that satisfy the axiom antecedent p are randomly generated; 2) the modifier method m is run on those initial configurations; 3) satisfiability of the (correspondingly instantiated) axiom consequent q is checked; and 4) the candidate axiom set $Q_s^\sharp$ is returned that contains those survival candidate axioms for which no refuted instances were obtained. In order to obtain a compact, minimal, and easily readable set of axioms, we apply a further *subsumption_filtering* postprocessing that independently tests the axiom sets Q and $Q_s^\sharp$ for subsumption (modulo associativity and commutativity of the logical conjunction $\wedge$) so that the pair of sets $(Q, Q^{\sharp\prime})$ that only contains the more general elements from each set is returned.

Let us illustrate the refinement postprocessing on our running example.

Example 18 *After the* `for` *loop of Algorithm 1, one axiom for each of the (17) final patterns is synthesized. After removing duplicates, 7 axioms are kept (see Figure 12), together with three candidate axioms (labelled as* `C1`*,* `C2` *and* `C3`*), where* `C3` *derives from the final configuration discussed in Example 15.*

Axioms `A3` and `A4` are simple instances of the more general candidate axiom `C1`, while axioms `A6` and `A7` are instances of the more general candidate axiom `C3`. However, because candidate axioms can be spurious, we cannot get rid of {`A1`, `A2`, `A3`, `A7`} yet. Moreover, `C1` is a correct specialized version of the (incorrect) candidate axiom `C2`: if `contains(s,x)` is 1 before the execution of `insert` (meaning that the element `x` is already in the original set `s`), the length of the resulting list is the same as before. However, due to the summary node abstraction that happens when not all the summarized nodes contain the element but at least one does, the explanation of the initial pattern through

$$
\begin{array}{ll}
\texttt{A1} & \left(\begin{array}{l} \texttt{isnull(s)} = 1 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 0 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 1 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{isfull(s')} = 0 \ \wedge \texttt{contains(s',x)} = 0 \ \wedge \\ \texttt{length(s')} = 0 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A2} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = \_v1 \ \wedge \\ \texttt{isfull(s)} = 1 \ \wedge \texttt{contains(s,x)} = \_v2 \ \wedge \\ \texttt{length(s)} = \_v3 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = \_v1 \ \wedge \\ \texttt{isfull(s')} = 1 \ \wedge \texttt{contains(s',x)} = \_v2 \ \wedge \\ \texttt{length(s')} = \_v3 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A3} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 1 \ \wedge \\ \texttt{length(s)} = 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{isfull(s')} = 0 \ \wedge \texttt{contains(s',x)} = 1 \ \wedge \\ \texttt{length(s')} = 1 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A4} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 1 \ \wedge \\ \texttt{length(s)} = 2 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{isfull(s')} = 0 \ \wedge \texttt{contains(s',x)} = 1 \ \wedge \\ \texttt{length(s')} = 2 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A5} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 1 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 0 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{contains(s',x)} = 1 \ \wedge \texttt{length(s')} = 1 \ \wedge \\ \texttt{ret} = 1 \end{array}\right) \\
\texttt{A6} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{contains(s',x)} = 1 \ \wedge \texttt{length(s')} = 2 \ \wedge \\ \texttt{ret} = 1 \end{array}\right) \\
\texttt{A7} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 2 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{contains(s',x)} = 1 \ \wedge \texttt{length(s')} = 3 \ \wedge \\ \texttt{ret} = 1 \end{array}\right) \\
\texttt{C1} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 1 \ \wedge \\ \texttt{length(s)} = \texttt{?l} + 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s',x)} = 1 \ \wedge \\ \texttt{length(s')} = \texttt{?l} + 1 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{C2} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = \_v1 \ \wedge \\ \texttt{length(s)} = \texttt{?l} + 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s',x)} = \_v1 \ \wedge \\ \texttt{length(s')} = \texttt{?l} + 1 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{C3} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = \texttt{?l} + 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s')} = 0 \ \wedge \texttt{isempty(s')} = 0 \ \wedge \\ \texttt{contains(s',x)} = 1 \ \wedge \\ \texttt{length(s')} = \texttt{?l} + 2 \ \wedge \texttt{ret} = 1 \end{array}\right)
\end{array}
$$

Figure 12: Set of axioms and candidate axioms of Example 15.

the observer `contains(s,x)` in `C2` cannot conclude that the element `x` is or is not yet in `s`. Consequently, the return value equated to `contains(s,x)` in the precondition of the axiom is the symbolic value $\_v1$, which stands for any possible value that the function may return (in this example program, either 0 or 1).

The refinement process is then triggered over the candidate axioms to check whether they can be falsified. We find that this is actually the case for `C2`; given the binary domain 0/1 of the `contains(s,x)` function, the axiom is straightforwardly falsified (e.g., by the test case where, for instance, `s` is a non-full set containing a single element with value 5, and `x` is 3). The final state does not satisfy the postcondition of axiom `C2` because, since the set `s` did not initially

contain the desired element, the modifier `insert(s,x)` does not return 0 and the length does not remain unaltered (since it is increased) after the execution. Since the axiom has been falsified (for the instance with `contains(s,x)=0`), it is automatically ruled out.

Having falsified the candidate `C2`, we proceed to repeat the same process with axioms `C1` and `C3`. Despite the fact that these candidate axioms do not suffer from any information loss and are indeed not falsifiable, we cannot remove the candidate mark from them, which prevents us from comparing any candidates with their corresponding instances while looking for subsumption. Therefore, the specification cannot get any more compact for now. In the next section, we show how this drawback can be overcome by deductively verifying candidate axioms, which finally yields an optimal synthesized contract.

As for the last element of the contract, the set of assignable program locations $\mathcal{L}$ is obtained as the union of the location sets that are recorded in the $\langle\rangle_{\mathsf{locations}}$ cells of the final symbolic execution states, which for our running example is $\mathcal{L} = \{\mathtt{s}, \mathtt{n}, \mathtt{new\_node}, \mathtt{new\_node} \mapsto \mathtt{value}, \mathtt{new\_node} \mapsto \mathtt{next}, \mathtt{s} \mapsto \mathtt{elems}, \mathtt{s} \mapsto \mathtt{size}\}$.

In the following section, we introduce a semi-automated axiom verification technique that can be used to further improve the inferred contracts: if a candidate axiom can be proved to be correct, we can move it to the set Q of demonstrably correct axioms, and this usually leads to a more concise specification since other (more specialized axioms) might be subsumed by this newly added correct axiom.

6 Axiom Verification

The key idea of this section is to reuse the $\mathbb{K}$ verification infrastructure to formally prove correctness of some candidate axioms so that our synthesized contracts can eventually be improved. The proposed method is based on the deductive verification infrastructure recently added to the $\mathbb{K}$ framework. In [35, 42], a language-independent, coinductive and deductive verification infrastructure is proposed that allows coinductive, properties (that are expressed in terms of operatons that *observe* (rather than construct) the state to be proved. Coinductive (or coalgebraic) specifications [34] are typically used to describe state-based dynamical systems in general, and object-oriented programming languages in particular, where the state space of the system is considered as a black box and where nothing is assumed about the way that the observable behavior is realized.

The verification infrastructure of $\mathbb{K}$ is built on top of *Reachability Logic* (RL) [41, 38, 40], whose formulae (called reachability rules) can be seen as a generalization of both rewrite rules and Hoare triples. Hence, Reachability Logic unifies operational and axiomatic semantics by using reachability rules to express dynamic properties of programs. To improve readability, in the following we distinguish the reachability rules that represent the program semantics (called *operational reachability rules*) from the reachability rules that express program

properties *à la Hoare* (that we simply call *reachability logic formulae*).

Roughly speaking, reachability rules of RL have the form $\pi \wedge \psi \Rightarrow \pi' \wedge \psi'$, where π and π' are *configuration patterns* (i.e., non-ground configuration terms), and ψ (resp. ψ') is a first-order logic formula on the variables in π (resp. π'). The semantics of reachability rules captures the intuition of *partial correctness* in axiomatic semantics, i.e., whenever the execution of the program reaches a concrete configuration c that is an instance of π and provided ψ holds in c, then, upon termination, the execution reaches a configuration which is an instance of π' and where ψ' holds, either in one of the possible computation paths stemming from c (*one-path reachability* [38]) or in all of them (*all-path reachability* [40]). In tool-supported $\mathbb{K}$ verification syntax, $\overline{\psi}$ is expressed as a *requires* clause, whereas $\overline{\psi'}$ is expressed as an *ensures* clause.

The $\mathbb{K}$ verification infrastructure is based on a proof system that has been proven to be sound (partially correct) and relatively complete in [42]. Given the rewriting-based (operational) semantics of a programming language written in $\mathbb{K}$, the semantics is used (through the $\mathbb{K}$ proof system) to derive correct program properties, without giving the language any other (axiomatic) semantics.

6.1 From (candidate) contract axioms to RL formulae

Let us now describe how we can generate the *to-be-proven* reachability logic formulae that specify dynamic program properties that conform to the candidate axioms we would like to verify in $\mathbb{K}$. One might naïvely think that this can be straightforwardly achieved by a direct translation of the (candidate) axioms into RL rules. However, such an immediate translation would not work because the observer methods appearing in the axioms that specify the behavior of method m are defined in the original program P, but the proof system cannot evaluate the program observers when verifying an RL formula on a fragment of P (namely, the piece of C source code where the modifier m is defined). This is why we do not depart from the candidate axiom itself but from the initial and final (abstract) symbolic configurations from which it derives.

Roughly speaking, given the (candidate) axiom $ax_{(I-F)}$, we generate a pair of corresponding patterns π and π' that respectively represent the information in the configurations I and F that is needed for verification. The configuration cells of I and F that are relevant for the verification process are:

1. I_{k}: the k cell of I, which contains the initial call to the modifier m with arguments *args*;
2. F_{k}: the k cell of F, which represents the returned value of the execution of m;
3. I_{heap}: the heap cell of I, which characterizes the precondition on the heap (note that it coincides with the init-heap cell of F);
4. F_{heap}: the heap cell of F, which represents the final heap; and

5. $F_{\mathsf{path\text{-}condition}}$: the path-condition cell of F, which characterizes the precondition on the input (primitive) arguments that, together with the init-heap cell, guarantees that the final configuration F is reached.

Hence, given the modifier call $m(args)$, the reachability rule $\pi \wedge \psi \Rightarrow \pi'$ is defined as:[6]

$$\langle \cdots \langle I_{\mathsf{k}} \rangle_{\mathsf{k}} \langle I_{\mathsf{heap}} \rangle_{\mathsf{heap}} \cdots \rangle_{\mathsf{cfg}} \wedge \overline{\psi} \Rightarrow \langle \cdots \langle F_{\mathsf{k}} \rangle_{\mathsf{k}} \langle F_{\mathsf{heap}} \restriction_{I_{\mathsf{heap}}} \rangle_{\mathsf{heap}} \cdots \rangle_{\mathsf{cfg}}$$

where the k cell of π contains the call $m(args)$ (in I_{k}) whereas the k cell of π' contains the final return value given by F_{k}, and the heap cell of π contains the initial heap I_{heap} whereas the heap cell of π' contains the restriction of F_{heap} to the entries in I_{heap}. Finally, the formula $\overline{\psi}$ corresponds to the path-condition cell of F (with some straightforward adaptations that we describe below).

6.2 Proving the candidate axiom correct

Consider again the final configuration F that we analyzed in Example 15. In the following, we show how we can characterize as a reachability formula the execution of `insert(s,x)` along the path leading from I to F. Remember that this path corresponds to the general case when the element x is indeed inserted in the list but the loop execution was approximated by abstract subsumption. Therefore, we could not derive a demonstrably correct axiom from it but only a candidate axiom (`C3` in Example 18).

Following the method described above, a corresponding reachability logic formula is obtained that we represent in Figure 13. The identifiers in upper-

$$\left\langle \begin{array}{c} \langle \texttt{insert(S,X)} \rangle_{\mathsf{k}} \\ \cdots \\ \texttt{S} \mapsto \left(\begin{array}{l} \texttt{capacity} \mapsto \texttt{CAPACITY}, \\ \texttt{size} \mapsto \texttt{SIZE}, \\ \texttt{elems} \mapsto \texttt{object(SE)} \end{array} \right), \\ \\ \mathit{lseg}(\texttt{object(SE)}, \mathrm{null})(\texttt{L} : \mathit{List}) \\ \cdots \end{array} \right\rangle_{\mathsf{heap}} \Rightarrow \left\langle \begin{array}{c} \langle \, \mathrm{tv}(\mathit{int}, 1) \rangle_{\mathsf{k}} \\ \cdots \\ \texttt{S} \mapsto \left(\begin{array}{l} \texttt{capacity} \mapsto \texttt{CAPACITY}, \\ \texttt{size} \mapsto \texttt{SIZE} + 1, \\ \texttt{elems} \mapsto \texttt{object(?NN)} \end{array} \right), \\ \\ \mathit{lseg}(\texttt{object(?NN)}, \mathrm{null})(\texttt{ListItem(X) L}) \\ \cdots \end{array} \right\rangle_{\mathsf{heap}}$$

requires $\texttt{SIZE} < \texttt{CAPACITY} \wedge \texttt{X}$ *not in* L

Figure 13: Reachability logic formula associated to the candidate axiom `C3`.

case and typographic font (e.g., `SIZE`) are pattern variables that are used to represent in RL syntax the symbolic values appearing in the given configuration F at the corresponding positions (e.g., `?s.size`). Pattern variables preceded by question marks, such as ?NN, are existentially quantified and represent memory

[6] Note that the reachability logic formulae we construct do not include a formula postcondition ψ'.

addresses and values that do not appear in the initial pattern π but are relevant for reasoning about the final pattern π'. We note that pattern variables are truly variables that are used to match actual, defined elements in program configurations that satisfy the pattern π during the verification; hence, pattern variables are different from symbolic addresses and values.

The *lseg* construct in Figure 13 is a built-in $\mathbb{K}$ abstract pattern that approximates a set of memory addresses that conform a list segment. Its profile is $lseg(P1, P2)(L)$, where $P1$ is the pointer to the first object of the segment, $P2$ is the pointer to the last object of the segment, and L is a list containing the data values of the nodes in the given list segment. $\mathbb{K}$ abstractions of this kind are intended to represent commonly-used data structures (and fragments of them) in order to make verification independent from low-level details of any program implementation. Note that our summary node abstraction in Section 4.2.3 transfers almost literally to the *lseg* $\mathbb{K}$ abstraction.

As is common in deductive program verification, in order to successfully prove a given property, it is generally necessary to first prove some auxiliary properties (e.g., loop invariants). An algorithm is presented in [27] that infers loop invariants for imperative list-processing programs, together with a prototype implementation for a C-like language that successfully generates loop invariants for a variety of programs. For the `while` loop involved in the execution of `insert(s,x)`, the invariant we need to prove (in tool-supported $\mathbb{K}$ syntax) is:

```
rule [loop-inv]:
     <struct> ...  STRUCT:Map ...  </struct>
     <k>
         while(n != NULL) {
              if(n->value == x) {
                found = 1;
              }
              n = n->next;
         }
     =>
         .K
         ...
     </k>
     <env>
           ...
           s |-> tv(struct set **, object(VS:Int))
           x |-> tv(int *, object(PX:Int))
           n |-> tv(struct listNode **, object(PN:Int))
           found |-> tv(int *, object(PF:Int))
           ...
     </env>
     <heap>
            ...
            object(VS) |-> tv(struct set *, object(S:Int))
            object(S) |-> (capacity |-> tv(int, SC:Int)
```

```
                        size |-> tv(int, SS:Int)
                        elems |-> tv(struct listNode *, object(SE:Int)))
          object(PN) |-> tv(struct listNode *, (object(P1:Int) => null))
          (lseg(object(SE), object(P1))(A:List)
          lseg(object(P1), null)(B:List)
     =>
          lseg(object(SE), null)(?C:List)
          object(PX) |-> tv(int, X:Int)
          object(PF) |-> tv(int, (lessOneVariable(PFV:Id) => ?PFV2:Int))
          ...
     </heap>
ensures A B ==K ?C andBool ((ListItem(X) not in ?C andBool ?PFV2 ==Int 0) orBool
          (ListItem(X) in ?C andBool ?PFV2 ==Int 1))
```

All we have to do now is to put the two rules above in a file `insert_spec.k`, and then verify them both with the command `krun insert.c --prove insert_spec.k`. The `--prove` option uses the all-path version of the reachability proof system to check that, for each rule in the file, on all terminating execution paths beginning with a configuration satisfying the left-hand side of the rule there exists some configuration satisfying the right-hand side of the rule [35].

6.3 Contract simplification by candidate axiom verification

Let us now consider a version of Algorithm 1 where, by following the verification methodology outlined above, a subset S of $Q^\sharp$ can be verified before the call $\mathit{refine}(Q, Q^\sharp)$ at line 12 of Algorithm 1 is undertaken. Since the proof system of reachability logic is sound, by replacing such a line 12 with $(Q', Q^{\sharp\prime}) := \mathit{refine}(Q \cup S, Q^\sharp - S)$, in the case when the newly added correct axioms in S subsume other (more specialized axioms) in Q, we achieve a more precise contract $C =< P, Q', \mathcal{L} >$.

For instance, by applying this version of Algorithm 1, since the RL formula of Figure 13 (that corresponds to the candidate axiom `C3`) can be verified, `C3` can be removed from the set of candidate axioms $Q^\sharp$ and then added as a new axiom (`A8`) to the set of correct axioms Q. Similarly, candidate `C1` can be also proven correct and then can be reclassified as axiom `A9`. Now, axioms `A6` and `A7` are removed by subsumption checking since they are instances of `A8`, and `A3` and `A4` are also removed because they are instances of `A9`. Therefore, the final contract that is synthesized for the running example (shown in Figure 14) contains just five axioms, as expected according to the intended specification provided in Figure 2.

We think the verification methodology improves our technique in several ways. On the one hand, we get an alternative specification of the program behavior as RL rules. On the other hand, it can be also used to prove non-candidate axioms, serving as a mechanism to (re-)assure correctness. Finally,

$$
\begin{array}{ll}
\texttt{A1} & \left(\begin{array}{l} \texttt{isnull(s)} = 1 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 0 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s}') = 1 \ \wedge \texttt{isempty(s}') = 0 \ \wedge \\ \texttt{isfull(s}') = 0 \ \wedge \texttt{contains(s}', \texttt{x}) = 0 \ \wedge \\ \texttt{length(s}') = 0 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A2} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = \_v1 \ \wedge \\ \texttt{isfull(s)} = 1 \ \wedge \texttt{contains(s,x)} = \_v2 \ \wedge \\ \texttt{length(s)} = \_v3 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s}') = 0 \ \wedge \texttt{isempty(s}') = \_v1 \ \wedge \\ \texttt{isfull(s}') = 1 \ \wedge \texttt{contains(s}', \texttt{x}) = \_v2 \ \wedge \\ \texttt{length(s}') = \_v3 \ \wedge \texttt{ret} = 0 \end{array}\right) \\
\texttt{A5} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 1 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = 0 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s}') = 0 \ \wedge \texttt{isempty(s}') = 0 \ \wedge \\ \texttt{contains(s}', \texttt{x}) = 1 \ \wedge \texttt{length(s}') = 1 \ \wedge \\ \texttt{ret} = 1 \end{array}\right) \\
\texttt{A8} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 0 \ \wedge \\ \texttt{length(s)} = \texttt{?l} + 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s}') = 0 \ \wedge \texttt{isempty(s}') = 0 \ \wedge \\ \texttt{contains(s}', \texttt{x}) = 1 \ \wedge \\ \texttt{length(s}') = \texttt{?l} + 2 \ \wedge \texttt{ret} = 1 \end{array}\right) \\
\texttt{A9} & \left(\begin{array}{l} \texttt{isnull(s)} = 0 \ \wedge \texttt{isempty(s)} = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s,x)} = 1 \ \wedge \\ \texttt{length(s)} = \texttt{?l} + 1 \end{array}\right) \Rightarrow \left(\begin{array}{l} \texttt{isnull(s}') = 0 \ \wedge \texttt{isempty(s}') = 0 \ \wedge \\ \texttt{isfull(s)} = 0 \ \wedge \texttt{contains(s}', \texttt{x}) = 1 \ \wedge \\ \texttt{length(s}') = \texttt{?l} + 1 \ \wedge \texttt{ret} = 0 \end{array}\right)
\end{array}
$$

Figure 14: Final contract for `insert(s,x)`.

we could even think of verifying specialized versions of the overly general axioms in $Q^{\sharp'}$ (which remained candidates after the post-processing refinement), which may lead to more complete specifications.

7 Implementation

We have developed a prototype implementation of the extended $\mathbb{K}$ symbolic machinery and contract inference algorithm described in the previous sections, which we used to mechanize our running example. The KindSpec 2.1 tool has been developed in Java and contains about 3000 lines of Java source code, and 600 lines of HTML5 and JavaScript code. The tool is publicly available together with more detailed experiments at `http://safe-tools.dsic.upv.es/kindspec2_1`.

We evaluated our prototype in classical contract inference and verification benchmark programs. Our main objective was to probe the quality of the discovered contracts as well as the precision and viability of the method on a variety of programs with loops and recursion that are commonly used in the literature on shape analysis and program verification with automatic inference of contracts [9, 20, 42]. Specification inference is notoriously expensive for accurate and strong properties; hence, research in the field usually uses test programs that size in the hundreds or tens of lines of code. Other techniques sacrifice precision or coverage in order to gain scalability, which is an important matter for future research.

Currently, $\mathbb{K}$ 4.0 does not include all the symbolic execution facilities of $\mathbb{K}$ 3.4 (for instance, instrumentation). Therefore, we run the axiom inference on

$\mathbb{K}$ 3.4 and then feed the verification infrastructure of $\mathbb{K}$ 4.0 with the computed output.

Our test platform was an Intel Core2 Quad CPU Q9300(2.50GHz) with 6 Gigabytes of RAM running $\mathbb{K}$ v3.4 on Maude v2.6, for inference tasks, and $\mathbb{K}$ v4.0 for verification. Table 1 summarizes the figures that we obtained for benchmark programs that contain (both cyclic and acyclic) data structures. In the LOC column, we describe the program size (in lines of code), while the Modifiers and Observers columns list the names of the symbolically executed functions in each category. The Paths explored column shows the number of final symbolic execution configurations that are automatically generated for each benchmark program (i.e., the total number of root-to-leave paths in the deployed symbolic execution trees), while the Extracted axioms column reflects how many *different* axioms are retrieved from those final configurations. The Overly general axioms column indicates the number of candidate axioms (i.e., potentially spurious); the Discarded falsified axioms column states how many of these candidate axioms were falsified and could not be specialized during the refinement process, so they were discarded; and Verifiable candidate axioms are the number of candidate axioms that are actually correct. The Final contract column indicates the total number of axioms that are distilled as a result of the whole process, including subsumption checking and verification. Assuming an appropriate set of observer functions, we are able to infer accurate (even complete) contracts for all our benchmarks.

With respect to the time cost of the inference, we distinguish between the amount of time taken for the symbolic execution of methods performed by $\mathbb{K}$ (which can be heavy considering both the modifier and the observers) and the elapsed time of the processing applied by our inference algorithm. The time spent in $\mathbb{K}$'s symbolic execution ranges from 1 min. to 5 min., depending on the quantity and complexity of the method definitions. On the other hand, the time taken for actual inference of contracts (once the symbolic execution trees have been deployed) ranges from approximately 150 ms to 300 ms. Our preliminary results are very encouraging since they show that general correct axioms can be inferred, leading to quite compact, clear, and correct specifications.

8 Related work and Conclusion

The wide interest in formal specifications as helpers for other analysis, validation, and verification tools has resulted in numerous approaches for (semi-)automatically computing different kinds of specifications that can take the form of contracts, snippets, summaries, assumptions, invariants, properties, process models, rules, graphs, automata, interfaces, or component abstractions. In this work, we focus on input-output relations; given a precondition for the state, we infer which modifications in the state are implied, and we express the relations as logical implications that reuse the program functions themselves.

Let us briefly discuss those strands of research that have influenced our work the most. A detailed description of the related literature can be found

Program	LOC	Modifiers	Observers	Paths explored	Extracted axioms	Overly general axioms	Discarded falsified axioms	Verifiable candidate axioms	Final contract
insert.c (running example)	112	insert	isnull isempty isfull contains length	17	10	3	1	2	5
delete.c (sequence of loops)	131	delete-lseg	isnull isempty contains_node interval_length	28	11	3	0	3	6
deallocate.c (reduction of heap size)	56	deallocate	isnull length	5	5	0	0	0	5
delete-all-circ.c (cyclic lists)	69	delete_all	isnull isempty length-circular	12	10	4	3	1	4
append.c (two symbolic lists, 1 loop)	60	append	isnull isempty length sum_sizes	12	12	2	1	1	4
merge.c (two symbolic lists, 2 loops)	129	merge	isnull isempty length sum_sizes	124	18	1	0	1	6
treeinsert.c (trees and recursion)	80	insert	isnull isempty find depth	32	6	2	2	0	4

Table 1: Experimental results for KindSpec 2.1 on programs manipulating lists and trees

in [1, 13, 20, 32, 45]. Our axiomatic representation is inspired by [44], which relies on a model checker for (bounded) symbolic execution of .NET programs and generates either Spec# specifications or parameterized unit tests. Similarly to [44], we aim to infer high-level (rich) information that is easily understandable by the programmer; however, we take advantage of $\mathbb{K}$ symbolic capabilities to generate simpler and more accurate formulas that avoid reasoning with the global heap because the different pieces of the heap that are reachable from the function argument addresses are kept separate. Unlike our symbolic approach, Daikon [17] and DIDUCE [22] detect program assertions by extensive testing. QuickSpec [39] relies on the automated testing tool QuickCheck to distill general laws that a Haskell program satisfies. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms (laws) that are constructed by using an API. This is similar to the approach of Henkel and Diwan [23], which generalizes the results of running tests on Java class interfaces as an algebraic specification. A combination of symbolic execution with dynamic testing is used in DySy [14]. By combining the concrete execution of actual test cases with a simultaneous symbolic execution of the same tests, DySy determines program properties that generalize the observed behaviors. Starting from simple, partial contracts previously written by the programmer, rich post-conditions involving quantification are defined in [45] by using random testing. By relying on symbolic execution and abstraction, our approach is able to guarantee completeness/correctness under some conditions. This is different from testing-based approaches such as [39, 23], which are limited to delivering properties that *have not been previously falsified* by a (finite) number of tests. An alternative approach to software specification discovery is based on inductive machine learning: rather than using test cases to validate a tentative specification, they are used as examples to *induce* the specification (e.g., [46, 19]). AbsSpec [6] is a semantic-based inference method that relies on abstract interpretation and generates laws for Curry programs in the style of QuickSpec. A different abstract interpretation approach to infer approximate specifications is that of Taghdiri and Jackson [43]. Finally, Ghezzi *et. al.* [18] infer specifications for container-like classes and express them as finite state automata that are supplemented with graph transformation rules.

Another related thread of research concerns the inference of Hoare triples (and invariants) that summarize a heap manipulating program. Existing approaches usually infer low-level specifications that are intended to be used later by automated verification or optimization tools. For instance, given a program procedure P that manipulates dynamic lists, [9, 32] combine abstract interpretation and weakest precondition computation to infer Hoare-like summaries of the form $[\Phi]\ \mathsf{P}\ [\hat{\Phi}]$ that are aimed to verify low-level properties such as memory access safety (e.g., not to reference a dangling pointer). The triple $[\Phi]\ \mathsf{P}\ [\hat{\Phi}]$ specifies that when P runs in a state satisfying $[\Phi]$ (the discovered precondition), it terminates without any memory error (such as null dereference) in a state satisfying $[\hat{\Phi}]$ (the discovered postcondition). Also, [27] combines symbolic execution with predicate abstraction in order to infer invariants that are almost limited to describing the mutation of the dynamic data structures. This is in

contrast to our approach, which infers richer (human-readable) assertions.

This work improves existing approaches in the literature in several ways. Thanks to the handling of $\mathbb{K}$'s algebraic laws by means of equational attributes [10], algebraic axioms such as associativity, commutativity, or identity are naturally supported in our approach. This leads to simpler and more efficient specifications and also makes it easy to reason about typed data structures such as lists (list concatenation is associative with identity element *nil*), multisets (bag insertion is associative-commutative with identity $\emptyset$), and sets (set insertion is associative-commutative-idempotent with identity $\emptyset$). In addition, we do not need to fix the size of arrays and dynamic structures or limit the number of iterations to ensure termination in the presence of loops; instead, we handle unbounded structures by means of lazy initialization and ensure termination by using abstraction. Our experiments show that our method can infer rich summaries that advance the state of the art of competing tools. For example, our tool infers contracts for challenging programs having recursive predicates, tree-like structures, and cyclic lists, which are not handled by competing tools, e.g., [9, 20].

Enhanced generality is achieved by inference tools by dealing with intermediate code (bytecode, CIL, ...). However, this is achieved at the expense of some additional precision loss [13]. We achieve generality thanks to the $\mathbb{K}$ Framework. Note that since our technique is not tied to the $\mathbb{K}$ semantics specification of KernelC, we expect that the methodology developed in this work can be easily extended to other languages for which a $\mathbb{K}$ semantics is given. Moreover, the correctness of the delivered specifications can be ensured by using the existing $\mathbb{K}$ formal tools. The contracts generated by our tool may be easily translated to richer (but also heavier) notations for behavioral interface C specifications such as ACSL or to the native syntax of some SMT solvers, which is planned as future work. We also plan to explore other existing abstract domains for structured data [8, 15] and integrate them in our tool to improve accuracy and applicability. Finally, in order to improve the level of automation of the verification, we are working towards providing the system with some facility to establish a correspondence between: 1) abstractions defined for contract synthesis that may appear in the (abstract) symbolic configurations and $\mathbb{K}$ abstraction patterns; and 2) program observers and (user-defined) $\mathbb{K}$ operators.

References

[1] María Alpuente, Marco A. Feliú, and Alicia Villanueva. Automatic Inference of Specifications Using Matching Logic. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM'13)*, pages 127–136, New York, NY, USA, 2013. ACM.

[2] María Alpuente, Daniel Pardo, and Alicia Villanueva. Automatic Inference of Specifications in the K Framework. *Electronic Proceedings in Theoretical Computer Science*, 200:1–17, 2015. arXiv: 1512.06941.

[3] María Alpuente, Daniel Pardo, and Alicia Villanueva. Symbolic Abstract Contract Synthesis in a Rewriting Framework. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'16)*, volume 10184 of *Lecture Notes in Computer Science*, pages 187–202. Springer, Cham, 2016.

[4] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1):53–67, 2009.

[5] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. Symbolic execution based on language transformation. *Computer Languages, Systems & Structures*, 44(A):48–71, 2015.

[6] Giovanni Bacci, Marco Comini, Marco Antonio Feliú, and Alicia Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP'12)*, pages 25–34, New York, NY, USA, 2012. ACM.

[7] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language, version 1.4, 2010.

[8] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, Berlin, Heidelberg, 2007.

[9] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint Analysis: A Shape Analysis That Discovers Preconditions. In *Proceedings of the 14th International Static Analysis Symposium (SAS'07)*, volume 4634 of *Lecture Notes in Computer Science*, pages 402–418. Springer, Berlin, Heidelberg, 2007.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[11] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, Berlin, Heidelberg, 2003.

[12] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, New York, NY, USA, 1977. ACM.

[13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic Inference of Necessary Preconditions. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, volume 7737 of *Lecture Notes in Computer Science*, pages 128–148. Springer, Berlin, Heidelberg, 2013.

[14] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 281–290, New York, NY, USA, 2008. ACM.

[15] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A Local Shape Analysis Based on Separation Logic. In *Proceedings of the 12th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, Berlin, Heidelberg, 2006.

[16] Chucky Ellison and Grigore Roşu. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544, New York, NY, USA, 2012. ACM.

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[18] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE'09)*, pages 430–440, 2009.

[19] Dimitra Giannakopoulou and Corina S. Păsăreanu. Interface Generation and Compositional Verification in JavaPathfinder. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 94–108. Springer Berlin Heidelberg, 2009.

[20] Bhargav S. Gulavani, Supratik Chakraborty, Ganesan Ramalingam, and Aditya V. Nori. Bottom-up Shape Analysis Using LISF. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(5):17:1–17:41, 2011.

[21] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A Visual Interactive Debugger Based on Symbolic Execution. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, pages 143–146, New York, NY, USA, 2010. ACM.

[22] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 291–301, New York, NY, USA, 2002. ACM.

[23] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering Documentation for Java Container Classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[24] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 553–568, New York, NY, USA, 2003. ACM.

[25] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, USA, 1986.

[27] S. Magill, A. Nanevski, A. Clarke, and P Lee. Inferring invariants in Separation Logic for imperative list-processing programs. In *3rd SPACE Workshop*, 2006.

[28] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 181–198. Springer, Berlin, Heidelberg, 2005.

[29] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[30] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[31] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[32] Yannick Moy and Claude Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 45(11):1184–1211, 2010.

[33] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[34] Horst Reichel. An approach to object semantics based on terminal coalgebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.

[35] Grigore Roşu. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *Logic, Rewriting, and Concurrency: Essays dedicated to José Meseguer on the occasion of his 65th Birthday*, Lecture Notes in Computer Science, pages 598–616. Springer, Cham, 2015.

[36] Grigore Roşu and Traian-Florin Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[37] Grigore Roşu and Andrei Ştefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 868–871, New York, NY, USA, 2011. ACM.

[38] Grigore Roşu, Andrei Ştefănescu, Stefan Ciobâcă, and Brandon Moore. One-Path Reachability Logic. In *Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'13)*, pages 358–367, 2013.

[39] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. Quick specifications for the busy programmer. *Journal on Functional Programming*, 27:e18, 2017.

[40] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-Path Reachability Logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *Lecture Notes in Computer Science*, pages 425–440. Springer, Cham, 2014.

[41] Andrei Ştefănescu, Stefan Ciobâcă, Brandon Moore, Traian Florin Şerbănuţă, and Grigore Roşu. Reachability Logic in K. Technical report, UIUC, November 2013.

[42] Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented*

Programming, Systems, Languages, and Applications (OOPSLA'16), pages 74–91. ACM, 2016.

[43] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Automated Software Engineering*, 14(1):87–121, 2007.

[44] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering Likely Method Specifications. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering (ICFEM'06)*, volume 4260 of *Lecture Notes in Computer Science*, pages 717–736. Springer Berlin Heidelberg, 2006.

[45] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring Better Contracts. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 191–200, New York, NY, USA, 2011. ACM.

[46] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic Extraction of Object-oriented Component Interfaces. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 218–228, New York, NY, USA, 2002. ACM.

A Full program code of the Running Example

```
1   #include <stdlib.h>
2
3   struct lnode{
4    int value;
5    struct lnode *next;
6   };
7   struct set {
8    int capacity;
9    int size;
10   struct lnode *elems;
11  };
12  struct set* new(int capacity) {
13   struct set *new_set;
14
15   new_set = (struct set*) malloc(sizeof(
          struct set));
16   if(new_set == NULL)
17    return NULL; /* no memory left */
18
19   new_set->capacity = capacity;
20   new_set->size = 0;
21   new_set->elems = NULL;
22   return new_set;
23  }
24
25  int insert(struct set *s, int x) {
26   struct lnode *new_node;
27   struct lnode *n;
28   int found;
29
30   if(s==NULL)
31    return 0; /* NULL set */
32   if(s->size >= s->capacity)
33    return 0; /* no space left */
34
35   n = s->elems;
36   found = 0;
37   while(n != NULL) {
38    if(n->value == x)
39     found = 1;
40    n = n->next;
41   }
42
43   if(found)
44    return 0; /* element in the set */
45
46   new_node = (struct lnode*) malloc(sizeof(
          struct lnode));
47   if(new_node == NULL)
48    return 0; /* no memory left */
49   new_node->value = x;
50   new_node->next = s->elems;
51   s->elems = new_node;
52   s->size += 1;
53   return 1; /* element added */
54  }
55
56  int isnull(struct set *s) {
57   if(s==NULL)
58    return 1;
59   return 0;
60  }
61
62  int isempty(struct set *s) {
63   if(s==NULL)
64    return 0;
65   if(s->elems==NULL)
66    return 1; /* s is empty */
67   return 0;
68  }
69
70  int isfull(struct set *s) {
71   if(s==NULL)
72    return 0;
73   if(s->size >= s->capacity)
74    return 1; /* s is full */
75   return 0;
76  }
77
78  int contains(struct set *s, int x) {
79   struct lnode *n;
80
81   if(s==NULL)
82    return 0; /* s is NULL */
83
84   n = s->elems;
85   while(n != NULL){
86    if(n->value == x)
87     return 1; /* element found */
88    n = n->next;
89   }
90   return 0; /* element NOT found */
91  }
92
93  int length(struct set *s) {
94   struct lnode *n;
95   int count;
96
97   if(s==NULL)
98    return 0; /* s is NULL */
99   count = 0;
100  n = s->elems;
101  while(n != NULL){
102   count = count + 1;
103   n = n->next;
104  }
105  return count;
106 }
```