Automatic Inference of Specifications in the $\mathbb{K}$ Framework [*]

María Alpuente Daniel Pardo Alicia Villanueva
DSIC, Universitat Politècnica de València
Camino de Vera s/n
46022 Valencia, Spain
{alpuente,daparpon,villanue}@dsic.upv.es

Despite its many unquestionable benefits, formal specifications are not widely used in industrial software development. In order to reduce the time and effort required to write formal specifications, in this paper we propose a technique for automatically discovering specifications from real code. The proposed methodology relies on the symbolic execution capabilities recently provided by the $\mathbb{K}$ framework that we exploit to automatically infer formal specifications from programs that are written in a non–trivial fragment of C, called KERNELC. Roughly speaking, our symbolic analysis of KERNELC programs explains the execution of a (modifier) function by using other (observer) routines in the program. We implemented our technique in the automated tool KINDSPEC 2.0, which generates axioms that describe the precise input/output behavior of C routines that handle pointer-based structures (i.e., result values and state change). We describe the implementation of our system and discuss the differences w.r.t. our previous work on inferring specifications from C code.

1 Introduction

Formal specifications can be used for various software engineering activities ranging from documenting software to automated debugging, verification, and test-case generation. However, there are a variety of reasons why software companies do not currently consider formal specification to be cost-effective to apply; these include time, complexity, and tool support. Specification inference can help to mitigate these problems and is also useful for legacy program understanding and malware deobfuscation [5].

This paper describes our ongoing work in developing a specification inference system for heap-manipulating programs that are written in a non-trivial fragment of C called KERNELC [22], which includes functions, structures, pointers, and I/O primitives. We rely on the (rewriting logic) semantic framework $\mathbb{K}$ [21], which facilitates the development of executable semantics of programming languages and also allows formal analysis tools for the defined languages to be derived with minimal effort.

A language definition in $\mathbb{K}$ essentially consists of three parts: the BNF language syntax (annotated with $\mathbb{K}$ specific attributes), the structure of program configurations, and the semantic rules. Similarly to the classic operational semantics, program configurations contain an encoding for the environment, the heap, stacks, etc. and are represented as algebraic datatypes in $\mathbb{K}$. Program configurations organize the state in units called *cells*, which are labeled and can be nested.

For example, following the $\mathbb{K}$ notation, the program configuration

$$\Big\langle\ \langle \mathrm{tv}(\mathit{int},0)\rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto \mathtt{x}\rangle_{\mathsf{env}} \langle \mathtt{x} \mapsto \mathrm{tv}(\mathit{int},5)\rangle_{\mathsf{heap}}\ \Big\rangle_{\mathsf{cfg}} \tag{1}$$

models the final state of a computation whose return value is the integer 0 (stored in the k cell, which contains the current code to be run), while program variable x (stored in the env cell) has the value 5

[*] This work has been partially supported by the EU (FEDER) and Spanish MINECO under grants TIN2015-69175-C4-1-R and TIN2013-45732-C4-1-P, and by Generalitat Valenciana ref. PROMETEOII/2015/013.

Marisa Navarro (Ed.): Proceedings PROLE 2015
EPTCS 200, 2015, pp. 1–17, doi:10.4204/EPTCS.200.1

(stored in the memory address given by x in the heap cell, where information about pointers and data structures is recorded). Variables representing symbolic memory addresses are written in sans-serif font.

In $\mathbb{K}$, the configuration (1) is a friendly representation for the term

```
<cfg>
       <k> tv(int,0) </k>
       <env> x => pointer(x) </env>
       <heap> pointer(x) => tv(int,5) </heap>
</cfg>
```

Symbolic execution (SE) is a well-known program analysis technique that allows the program to be executed using *symbolic* input values instead of actual (concrete) data so that it executes the program by manipulating program expressions involving the symbolic values [17, 20]. Unlike concrete execution, where the path taken is determined by the input, in symbolic execution the program can take any feasible path. That path is given by a logical constraint on past and present values of the variables, called *path condition* because it is formed by constraints that are accumulated on the path taken by the execution to reach the current program point. 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). One of the traditional drawbacks of SE-based techniques is the high cost of decision procedures to solve path conditions. Recently, SE has found renewed interest due in part to the huge recent advances in decision procedures for logical satisfiability.

$\mathbb{K}$ semantics is traditionally[1] compiled into Maude [7] for execution, debugging, and model checking. $\mathbb{K}$ implements reachability logic in the same way that Maude implements rewriting logic. In reachability logic, a particular class of first-order formulas with equality (encoded as (boolean) terms with logical variables and constraints over them) is used. These formulas, called *patterns*, specify those concrete configurations that match the pattern algebraic structure and satisfy its constraints. Since patterns allow logical variables and constraints over them, by using patterns, $\mathbb{K}$ rewriting becomes *symbolic execution* with the semantic rules of the language [3]. The SMT solver Z3 [19] is used in $\mathbb{K}$ for checking the satisfiability of the path constraints.

Symbolic execution in $\mathbb{K}$ relies on an automated transformation of both $\mathbb{K}$ 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. The transformed, symbolic rules define how symbolic configurations are rewritten during computation. Roughly speaking, each data structure field and program variable originally holds an initial, symbolic value. Then, by symbolically executing a program statement, the configuration cells (such as k, env and heap in the example above) are updated by mapping fields and variables to new symbolic values that are represented as symbolic expressions, while the path conditions (stored in the path-condition cell) are correspondingly updated at each branching point.

For instance, the following pattern

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

specifies the set of configurations as follows: (1) the k cell contains the integer value 0; (2) in the env cell, program variable x (in typographic font) is associated to the memory address x and s is bound to the

[1] $\mathbb{K}$'s backend is currently being ported into Java, and $\mathbb{K}$ 4.0 is expected to be released when the Java backend is deemed a suitable complete replacement for Maude.

pointer s; and (3) in the heap cell, the field size of s contains the symbolic value ?s.size (symbolic values are preceded by a question mark). Additionally, s is not null and the value of its size field is greater than 0.

In this paper, we redesign the technique of [1] for discovering specifications for heap-manipulating programs by adapting the symbolic infrastructure of $\mathbb{K}$ to support the specification inference process for KERNELC programs. Specification inference is the task of discovering high-level specifications that closely describe the program behavior. Given a program P, the specification discovery problem for P is typically described as the problem of inferring a likely specification for every function m in P that uses I/O primitives and/or modifies the state of encapsulated, dynamic data structures defined in the program. Following the standard terminology, any such function m is called a *modifier*. The intended specification for m is to be cleanly expressed by using any combination of the non-modifier functions of P (i.e., functions, called *observers*), which inspect the program state and return values expressing some information about the encapsulated data. However, because the C language does not enforce data encapsulation, we cannot assume purity of any function: every function in the program can potentially change the execution state, including the heap component of the state. In other words, any function can potentially be a *modifier*; hence we simply define an *observer* as any function whose return type is different from void (i.e., potentially expresses a property concerning the final *heap* contents or the return value of the function call).

The key idea behind our inference methodology was originally described in [1]. Given a *modifier* procedure for which we desire to obtain a specification, we start from an initial symbolic state s and symbolically evaluate m on s to obtain as a result a set of pairs (s,s') of initial and final symbolic states, respectively. Then, the observer methods in the program are used to explain the computed final symbolic states. This is achieved by analyzing the results of the symbolic execution of each observer method o when it is fed with (suitable information that is easily extracted from) s and s'. More precisely, for each pair (s,s') of initial and final states, a pre/post statement is synthesized where the precondition is expressed in terms of the observers that *explain* the initial state s, whereas the postcondition contains the observers that *explain* the final state s'. 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 contrast to [1], in this work we rely on the newly defined symbolic machinery for $\mathbb{K}$, while [1] was built on a symbolic infrastructure for KERNELC that we manually developed in a quite ad-hoc and error prone way, by reusing some spare features of the formal verifier MatchC [24]. This strategic technological change will allow us to define a generic and more robust framework for the inference of specifications of languages defined within the $\mathbb{K}$ framework. Also differently from [1], here we fully use the lazy initialization approach of [2] in order to deal with complex data structures and pointers, which were only partially adopted in our previous work. With lazy initialization, the first time an uninitialized field or reference is accessed, instead of considering all the possible instances of these data structures, the execution is non-deterministically branched by simply initializing the field to the different scenarios: the field is null, points to a new object with uninitialized fields, or points to an already created object.

Contributions We summarize the main contributions of this paper as follows:

- In the current $\mathbb{K}$ system, we revisit the approach to extract lightweight specifications from heap-manipulating code of [1], which consists of a symbolic analysis that explores and summarizes the behavior of a *modifier* routine by using other available routines in the program, called *observers*.

  This corresponds to the primary motivation for this work: to migrate the specification discovery technique of [1] to the holistic framework of the latest 𝕂 release, which is based on symbolic execution, whereas [1] relied on the MatchC verification infrastructure of the old 𝕂 platform, which is currently unsupported.

- We adapt the symbolic mechanism of 𝕂 to deal with KERNELC, also adapting and implementing the lazy initialization technique for manipulating complex KERNELC input data.
- We implement our specification inference technique in the KINDSPEC 2.0 system, which fully builds on the capabilities of the SMT solver Z3 [19] to not only prove the (accumulated path) constraints as in 𝕂 but also to incrementally simplify them on the fly.

  Moreover, the synthesized pre/post axioms are further simplified (to be given more compact representation) and are eventually presented in a more friendly sugared form that abstracts from any implementation details.

Related work The wide interest in program specifications as helpers for other analysis, validation, and verification processes have resulted in numerous approaches for (semi-)automatic computation of different kinds of specifications. Specifications can be property oriented (i.e., described by pre-/post conditions or functional code); stateful (i.e., described by some form of state machine); or intensional (i.e., described by axioms), and can take the form of contracts, interfaces, summaries, assumptions, invariants, properties, component abstractions, process models, rules, graphs, automata, etc. 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, thus improving comprehension since the user is acquainted with them. A thorough comparison with the related literature can be found in [1]. Here we only try to cover those lines of research that have influenced our work the most.

Our axiomatic representation is inspired by [26], which relies on a model checker for symbolic execution and generates either Spec# specifications or parameterized unit tests. In contrast to [26], we take advantage of 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 [11] and DIDUCE [14] detect program invariants by extensive testing. Also, Henkel and Diwan [15] dynamically discover specifications for interfaces of Java classes by generalizing the results of automated tests runs as an algebraic specification. QUICKSPEC [6] 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 that are constructed using an API, similarly to [15]. ABSSPEC [4] 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 [25]. A combination of symbolic execution with dynamic testing is used in Dysy [8]. An alternative approach to software specification discovery is based on inductive matching learning: rather than using test cases to validate a tentative specification, they are used as examples to *induce* the specification (e.g., [27, 13]). Finally, Ghezzi *et al.* [12] infer specifications for container-like classes and express them as finite state automata that are supplemented with graph transformation rules.

This work improves existing approaches in the literature in several ways. Thanks to the handling of MAUDE's (hence 𝕂's) equational attributes [7], algebraic laws such as associativity, commutatitvity, or identity are naturally supported in our approach, which 1) leads to simpler and more efficient specifications, and 2) 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 $\emptyset$). As a further advantage w.r.t. [26], in our framework, the correctness of the delivered specifications can be automatically ensured by using the existing $\mathbb{K}$ formal tools [21] . Since our approach is generic and not tied to the $\mathbb{K}$ semantics specification of KERNELC, we expect the methodology developed in this work to be easily extendable to other languages for which a $\mathbb{K}$ semantics is given.

Plan of the paper In Section 2, we summarize the key concepts of the $\mathbb{K}$ framework that are crucial for this work. Section 3 introduces a running example that is used as a case study throughout the paper to discuss the adequacy and effectiveness of the proposed inference methodology. Section 4 presents how we had to adapt the symbolic machinery of $\mathbb{K}$ to support specification discovery. Finally, Section 5 describes our specification inference procedure and discusses directions for future work.

2 The $\mathbb{K}$ Framework

In this section, we recall the fundamental concepts of the $\mathbb{K}$ semantic framework [23].

$\mathbb{K}$ [23] is a framework for engineering language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, and formal analysis tools such as model checkers and deductive theorem provers at no additional cost. It also supports various backends, such as Maude and, experimentally, Coq. In other words, language semantics defined in $\mathbb{K}$ can be translated into Maude or Coq definitions. Complete formal program semantics for Scheme, Java 1.4, JavaScript, Python, Verilog, and C are currently available in $\mathbb{K}$ [21, 23].

Program configurations are represented in $\mathbb{K}$ as potentially nested structures of labeled cells (or containers) that represent the program state. They include a computation stack or continuation (named k), environments (env, heap), and a call stack (stack), among others. $\mathbb{K}$ cells can be lists, maps, (multi)sets of computations, or a multiset of other cells. Computations carry “computational meaning” as special nested list structures that sequentialize computational tasks, such as fragments of a program. The part of the $\mathbb{K}$ configuration structure for the KERNELC semantics that is relevant to this work is shown below.

$$\Big\langle\ \langle \mathrm{K}\rangle_{\mathsf{k}}\langle \mathrm{Map}\rangle_{\mathsf{env}}\langle \mathrm{List}\rangle_{\mathsf{stack}}\langle \mathrm{Map}\rangle_{\mathsf{heap}}\ \Big\rangle_{\mathsf{cfg}}$$

Rules in $\mathbb{K}$ state how configurations (terms) evolve throughout the computation. Similarly to configurations, rules can also be graphically represented and are split in 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, we show the KERNELC rule for assigning a value V of type T to the variable X. This rule uses three cells: k, env, and heap. The env cell is a mapping of variable names to their memory positions, whereas the heap cell binds the active memory positions to the actual values. Meanwhile, the k cell represents a stack of computations waiting to be run, with the left-most (i.e., top) element of the stack being the next computation to be undertaken.

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

This rule states that, if the next pending computation (which may be a part of the evaluation of a bigger expression) consists of an assignment $X = \mathrm{tv}(T,V)$, then we look for X in the environment ($X \mapsto \_$)

and we update the associated mapping in the memory with the new value V of type T (tv(T,V)). The value tv(T,V) is kept at the top of the stack (it might be used in the evaluation of the bigger expression). The rest of the cell's content in the rule does not undergo any modification (this is represented by the ··· card). This example rule reveals a useful feature of $\mathbb{K}$: «rules only need to mention the minimum part of the configuration that is relevant for their operation». That is, only the cells read or changed by the rule have to be specified, and, within a cell, it is possible to omit parts of it by simply writing "··· ". For example, the rule above emphasizes the interest in: the instruction $X = \text{tv}(T,V)$ only at the beginning of the k cell, and the mapping from variable X to its memory pointer X at any position in the env cell. Except for the subterms that are explicitly identified, upon variable assignment everything is kept unchanged.

The (desugared) $\mathbb{K}$ rule for KernelC variable assignment is

```
rule   <k> X = tv(T,V) => tv(T,V) ...</k>
       <env>... X |-> pointer(X) ...</env>
       <heap>... pointer(X) |-> (_ => tv(T,V)) ...</heap>
```

where the underscore stands for an anonymous variable. The ellipses are also part of the desugared $\mathbb{K}$ syntax and are used to replace the unnecessary parts of the cells. Hence, also in the desugared rule, the developers typically only mention the information that is absolutely necessary in their rules.

3 Running Example

Our inference technique relies on the classification scheme developed in [18] for data abstractions, where a function (method) may be either a *constructor*, a *modifier* or an *observer*. A constructor returns a new object of the class from scratch (i.e., without taking the object as an input parameter). A modifier alters an existing class instance (i.e., it changes the state of one or more of the data attributes in the instance). An observer inspects the object and returns a value characterizing one or more of its state attributes. We do not assume the traditional premise of the original classification in [18] that states that observer functions do not cause side effects on the state. This is because we want to apply our technique to any program, which may be written by third-party software producers that may not follow the observer purity discipline.

Let us introduce the leading example that we use to describe the inference methodology developed in this paper: a KernelC implementation of an abstract datatype for representing doubly-linked lists. Since the whole example includes a total of 13 methods, due to space restrictions we have chosen to comment on just one modifier and five observer methods (of which 2 are both modifiers and observers).

Example 1 *In the* KernelC *program of Figure 1, we represent a doubly-linked list as a data structure (*`struct List`*) that contains some content (field* `data`*), a pointer to the previous element in the list (field* `prev`*), and another pointer to the succesive element in the list (field* `next`*).*

A call `append(list,d)` *to the* `append` *function proceeds as follows: first, a new node* `new_node` *is allocated in memory; it is filled with the value* `d` *and its* `next` *pointer is initialized to* `NULL` *since it will become the last item in the list. Next, the function checks that the provided list* `list` *is not* `NULL`*, in which case it binds the* `next` *pointer of the final element of the list to the newly created node, and the* `prev` *pointer of the new node to the final node of* `list`*, then returns the pointer to the whole resulting* `list`*. Otherwise, when the input list* `list` *is null, then the* `prev` *pointer of* `new_node` *is initialized to* `NULL` *and the resulting full-fledged list that consists of one single element is simply returned.*

The observer function `length` *traverses the list by visiting every node in order to count the number of elements in the list. The observer function* `head` *returns the data field of the first node of the list;*

```
#include <stdlib.h>

struct List {
  void* data;
  struct List* next;
  struct List* prev;
};

struct List* append(struct List* list, void* d)
      {
  struct List* new_node;
  struct List* final;

  new_node = (struct List*) malloc(sizeof(
       struct List));
  new_node->data = d;
  new_node->next = NULL;

  if (list != NULL) {
     final = list;
     if (final != NULL) {
        while (final->next != NULL)
          final = final->next;
     }
     final->next = new_node;
     new_node->prev = final;

     return list;
  }
  else {
     new_node->prev = NULL;
     list = new_node;
     return list;
  }
}

int length(struct List* list) {
  int len;

  len = 0;
  while (list != NULL) {
    len = len + 1;
    list = list->next;
  }
  return len;
}

struct List* reverse(struct List* list) {
  struct List* final;

  final = NULL;
  while (list != NULL) {
    final = list;
    list = final->next;
    final->next = final->prev;
    final->prev = list;
  }
  return final;
}

void* head(struct List* list) {
  if (list != NULL) {
      while (list->prev != NULL)
          list = list->prev;
  }
  return list->data;
}

struct List* last(struct List* list) {
  struct List* reversed;

  reversed = reverse(list);
  return head(reversed);
}

int find(struct List* list, void* d) {
  int found;

  found = 0;
  while (list != NULL && !(found)) {
      if (list->data == d)
        found = 1;
      else
        list = list->next;
  }
  return found;
}

struct List* init(struct List* list) {
  struct List* aux;

  if (list != NULL) {
     if (list->next != NULL) {
      aux = list->next;
      while (aux->next->next != NULL)
          aux = aux->next;
      aux->next = NULL;
     }
     else
      list = NULL;
  return list;
}
```

Figure 1: KERNELC implementation of a doubly-linked list.

$$\left(\begin{array}{l} \texttt{length(list)} = 0 \wedge \\ \texttt{reverse(list)} = \texttt{NULL} \wedge \\ \texttt{find(list,d)} = 0 \wedge \\ \texttt{init(list)} = \texttt{NULL} \wedge \\ \texttt{last(list)} = \texttt{NULL} \end{array}\right) \quad \Rightarrow \quad \left(\begin{array}{l} \texttt{length(list}'\texttt{)} = 1 \wedge \\ \texttt{reverse(list}'\texttt{)} = \texttt{list} \wedge \\ \texttt{find(list}'\texttt{,d)} = 1 \wedge \\ \texttt{init(list}'\texttt{)} = \texttt{NULL} \wedge \\ \texttt{last(list}'\texttt{)} = \texttt{d} \wedge \\ \texttt{ret} = \texttt{list}' \end{array}\right)$$

$$\left(\begin{array}{l} \texttt{length(list)} = x \wedge \\ \texttt{length(list)} > 0 \end{array}\right) \quad \Rightarrow \quad \left(\begin{array}{l} \texttt{length(list}'\texttt{)} = x+1 \wedge \\ \texttt{find(list}'\texttt{,d)} = 1 \wedge \\ \texttt{last(list}'\texttt{)} = \texttt{d} \wedge \\ \texttt{ret} = \texttt{list}' \end{array}\right)$$

Figure 2: Expected specification for the `append(list,d)` function call.

`last` *delivers the data field of the last node of the list, which is done by first invoking* `reverse(list)` *to compute a mirrored version of the parameter* `list` *and then accessing the* `data` *field of its first node. The function* `init(list)` *returns the same list after removing the last item of the list. Finally, the observer* `find` *looks for the provided* `d` *value in the list, and returns* `1` *(which stands for* true*) if the* `d` *value is found; otherwise, the value* `0` *(which stands for* false*) is returned.*

From the program code of Example 1, for each modifier function m, we aim to synthesize an axiomatic specification that consists of a set of implication formulas $t_1 \Rightarrow t_2$, where t_1 and t_2 are conjunctions of equations of the form $l = r$. The left-hand side l of each equation can be either

- a call to an observer function and then r represents the return value of that call;
- the keyword `ret`, and then r represents the value returned by the modifier function m being observed.

Informally, the statements on the left-hand and right-hand sides of the symbol $\Rightarrow$ are respectively satisfied before and after the execution of a function call to m. We adopt the standard primed notation for representing variable values after the execution.

Example 2 *Consider again the program of Example 1. The specification for the (modifier) function* `append` *that inserts an element* `d` *at the end of the list* `list` *is shown in Figure 2. The specification consists of two implications stating the conditions that are satisfied before and after the execution of a symbolic function call* `append(list,d)`*. The first formula can be read as follows: if, before executing* `append(list,d)`*, the result of running* `length(list)` *is equal to 0, a call to* `find(list,d)` *returns 0 (since no value can be found in an empty list) and the results of executing* `reverse(list)`*,* `init(list)`*, and* `last(list)` *are all* `NULL` *(i.e., the list is empty), then, after executing* `append(list,d)`*, the length of the augmented list is 1, the reversed list coincides with the list itself, the value* `d` *can now be found in the list, the init segment of the list is* `NULL`*, the last element is the inserted value and the call returns the pointer to the (augmented) list. The second formula represents the general case: given a* `list` *with an arbitrary size x, the call* `append(list,d)` *causes the length to be increased by 1, the inserted value is found in the list, in particular it is returned by the* `last` *observer, and the (augmented) list is returned. Since the* `append` *function does not restrict the insertion to the cases in which the* `d` *value is still not inside the list, we cannot assume* `find` *to return 0 before running the modifier function* `append`*.*

Note that any implication formula in the specification may contain multiple facts (in the pre- or post-condition) that refer to function calls that are assumed to be run independently under the same initial conditions. This avoids making any assumptions about function purity or side-effects.

4 Symbolic Execution in the $\mathbb{K}$ Framework

Symbolic execution consists of executing programs with symbolic values instead of concrete values. It proceeds like standard execution except that, when a function or routine is called, symbolic values are assigned to the actual parameters of the call and computed values become symbolic expressions that record all operations being applied. When symbolic execution reaches a conditional control flow statement, every possible execution path from this execution point must be explored. In order to keep track of the explored execution paths, symbolic execution also records the assumed (symbolic) conditions on the program inputs that determine each execution path in the so-called *path conditions* (one per possible branch), which are empty at the beginning of the execution. A path condition consists of the set of constraints that the arguments of a given function must satisfy in order for a concrete execution of the function to follow the considered path. Without loss of generality, we assume that the symbolically executed functions access no global variables; they could be easily modeled by passing them as additional function arguments.

Example 3 *Consider again the* `append` *function of Example 1. Assume that the input values for the actual parameters* `list` *and* `d` *are the symbolic pointer* list *and the symbolic value* ?d*, respectively. Then, when the symbolic execution reaches the first* `if` *statement in the code, it explores the two paths arising from considering both the satisfaction and non-satisfaction of the guard in the conditional branching statement. The path condition of the first branch is updated with the constraint* $\mathsf{list} \neq \mathtt{NULL}$*, whereas* $\mathsf{list} = \mathtt{NULL}$ *is added to the path condition in the second branch.*

To summarize, symbolic execution can be represented as a tree-like structure where each branch corresponds to a possible execution path and has an associated path condition. The *successful* paths are those leading to a final (symbolic) configuration that encloses a satisfiable path constraint and that typically stores a (symbolic) computed result.

For the symbolic execution of KERNELC programs, we must pay attention to pointer dereference and initialization. In C, a structured datatype (`struct`) is an aggregate type that is used to comprise a nonempty set of sequentially allocated member objects[2], called fields, each of which has a name and a type. When a `struct` value is created, C uses the address of its first field to refer to the whole structure. In order to access a specific field `f` of the given structure type, C computes `f`'s address by adding an offset (the sum of the sizes of each preceding field in the definition) to the address of the whole structure.

In our symbolic setting, the pointer arithmetics and memory layout machinery are abstracted by 1) using symbolic variables as 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 identifiers of both the structure object and the field name.

Example 4 *Consider the structure type* `List` *of Example 1. The following configuration records a list variable* `l` *with: 1) the integer 7 in its* `data` *field; 2) a reference (pointer) named* first_node *as the value of its* `prev` *field; and 3) a reference (pointer)* third_node *as the value of its* `next` *field:*

$$\Big\langle \;\ldots \langle \mathtt{l} \mapsto \mathsf{l} \rangle_{\mathsf{env}} \langle \cdots\; \mathsf{l} \mapsto (\mathtt{data} \mapsto \mathsf{tv}(\mathit{int}, 7), \mathtt{prev} \mapsto \mathsf{first\_node}, \mathtt{next} \mapsto \mathsf{third\_node}) \;\cdots \rangle_{\mathsf{heap}} \ldots \;\Big\rangle_{\mathsf{cfg}}$$

In order to access a field of the list `l` *(e.g., its* `data` *field), the corresponding index is computed by juxtaposing the identifier of the* `data` *field to the pointer* l*, thus mimicking how the concrete access would be done in* C *(i.e.,* `l->data`*).*

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

Another critical point is the *undefinedness* problem that occurs in C programs when accessing uninitialized memory addresses. The KERNELC semantics that we use preserves the concrete *well-definedness* behavior of pointer-based program functions of C while still detecting the *undefinedness* cases in a way similar to the C operational semantics of [10]. However, in our inference setting, we have no a priori information regarding the memory (specifically, information about the (un)initialized memory addresses). Therefore, when symbolic execution accesses (potentially uninitialized) memory positions, two cases must be considered: the case in which the memory is actually initialized and stores an object, and the case in which it stores a null pointer. In contrast to the approach described in [1], we do not consider cases where the pointer is undefined (i.e., when the execution is halted due to forbidden pointer access). This avoids accumulating too many solutions with undefined behavior, which could cause an explosion of axioms for programs that access new objects frequently, resulting in huge and redundant output specifications. For the case when the memory positions are actually initialized with non-null objects, a strategy to reconstruct the original object in memory is needed. We adapt the lazy initialization of objects of [16] to our setting: when a symbolic address (or address expression) is accessed for the first time, SE initializes the memory object that is located at the given address with a new symbolic value. This means that the mapping in the heap cell is updated by assigning a new free variable to the symbolic address of the accessed field so that, from that point on, accesses to that field can only succeed. As a result, *undefined* computations can only occur in the case of syntactic program errors (i.e., expressions that are not accepted in the specification of the language).

Example 5 *(Example 3 continued) Before executing the first* `if` *statement for the first time, assume that the* heap *cell is empty, which means that nothing is known about the structure of the* heap. *After symbolically executing the guard of the* `while` *statement (which refers to the* `next` *field of the structured data* `final`*), by applying the lazy initialization approach, the* heap *cell gets updated to:*

$$\left\langle \begin{array}{c} \dots \\ \langle \cdots\ \mathtt{list} \mapsto \mathsf{list}, \mathtt{final} \mapsto \mathsf{list}\ \cdots \rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto (\mathtt{data} \mapsto \mathsf{undef}, \mathtt{prev} \mapsto \mathsf{undef}, \mathtt{next} \mapsto \mathsf{list.next}) \\ \mathsf{list.next} \mapsto \mathsf{undef} \end{array} \right\rangle_{\mathsf{heap}} \\ \dots \end{array} \right\rangle_{\mathsf{cfg}}$$

In other words, new symbolic bindings for the actual parameters are added, which represent the assumptions we made over the corresponding data structures. More specifically, the accessed field is initialized with a fresh symbolic pointer list.next *whereas the fields that have not been accessed yet (temporarily) remain undefined, in a state that is specified by the symbolic constant* undef.

In the following section, we describe $\mathbb{K}$'s symbolic execution machinery and how we adapted it to support discovering program specifications.

4.1 The symbolic machinery in $\mathbb{K}$

Recently, the $\mathbb{K}$ framework has been enriched with a tool that automatically compiles language definitions into symbolic semantics. In other words, any language that is formally defined in $\mathbb{K}$ can (ideally) benefit, without cost, from symbolic execution. The $\mathbb{K}$ symbolic backend automatically attaches to the configuration a new cell, called path-condition, for the conditions on the input arguments that are accumulated during the symbolic execution. Roughly speaking, the mechanism works as follows: whenever a non-deterministic choice is found (i.e., the term at the top of the k cell can be rewritten by applying different rules), the symbolic engine considers each path independently, storing the assumptions that enable each concrete execution path in the path-condition cell. Therefore, the symbolic execution of programs under the $\mathbb{K}$ framework results in a set of *patterns* (consisting of the final symbolic configuration that encloses the corresponding path-condition cell) which we call *final patterns*.

Example 6 *Assume that our $\mathbb{K}$ specification contains these two rules, which represent the possible rewritings of an* `if` *statement:*

$$\langle \underset{S}{\underline{\text{if}\,(\texttt{true})\, S \,\text{else}\, \_}} \cdots \rangle_{\mathsf{k}} \qquad \langle \underset{S}{\underline{\text{if}\,(\texttt{false})\, \_ \,\text{else}\, S}} \cdots \rangle_{\mathsf{k}}$$

Now assume that we are running the following piece of code:

```
if (x > y) return 1; else return 0;
```

with symbolic variables `x` *and* `y`*, and no initial restrictions over them (i.e., the* path-condition *cell is initialized to* `true`*). The compilation of the language with $\mathbb{K}$'s symbolic backend explores both branches (i.e., the case when the guard* `x > y` *is true and the case when the guard evaluates to false), which respectively lead to the following patterns*[3]*:*

Branch 1: $\langle\ \langle \text{tv}(\mathit{int}, 1)\rangle_{\mathsf{k}} \ldots\ \rangle_{\mathsf{cfg}} \langle\ ?\mathsf{x} > ?\mathsf{y}\ \rangle_{\mathsf{path-condition}}$

Branch 2: $\langle\ \langle \text{tv}(\mathit{int}, 0)\rangle_{\mathsf{k}} \ldots\ \rangle_{\mathsf{cfg}} \langle\ ?\mathsf{x} \leq ?\mathsf{y}\ \rangle_{\mathsf{path-condition}}$

As already mentioned, the exhaustive symbolic execution of all paths cannot always be achieved in practice because an unbounded number of paths can arise in the presence of loops or recursion. We follow the standard approach to avoid the exponential blowup that is inherent in path enumeration by exploring loops up to a specified number of unfoldings. This ensures that SE ends for all explored paths, thus delivering a finite (partial) represention of the program behavior [9]. Obviously, not all the potential execution paths are feasible, but $\mathbb{K}$ deals with this automatically and transparently to the user by using the theorem prover Z3 [19] to check the satisfiability of the path condition constraints.

It is important to note that the symbolic $\mathbb{K}$ engine is not endowed with the lazy initialization technique. As a consequence, any branching in $\mathbb{K}$'s symbolic execution trees is associated to the evaluation of a guarded instruction (conditional, while loop, etc.), whereas lazy initialization also adds bifurcations when mimicking the access to complex data structures (objects) because all possible scenarios are considered. In other words, branching is not only caused by the evaluation of guards (boolean expressions), but also by other kinds of expressions (for instance when assigning a value to a data structure).

Note that the path-condition cell (where constraints associated to guards are stored) is not under our control but is automatically handled by the $\mathbb{K}$ symbolic engine, which ensures language independence. For this reason, we have adopted the solution to introduce a new cell, called init-struct, into the configuration that is used to store those constraints associated to non-guarded instructions that refer to complex data structures. By abuse, when we refer to the path condition ϕ of a pattern, we implicitly consider that ϕ includes the constraints in init-struct as well.

In the following section, we formulate our symbolic specification inference algorithm.

5 Inferring Specifications Using $\mathbb{K}$'s Symbolic Execution

Let us introduce the basic notions that we use in our formalization. Given an input program, let $\mathscr{F}$ be the set of functions in the program. We distinguish the set of observers $\mathscr{O}$ and the set of modifiers $\mathscr{M}$.

[3] We only write those cells that are relevant for the example.

A function can be considered to be an *observer* if it explicitly returns a value, whereas any method can be considered to be a *modifier*. Thus, the set $\mathscr{O} \cap \mathscr{M}$ is generally non empty. For instance, the function `reverse` in Example 1 is both an observer and a modifier function.

Given a function $f \in \mathscr{F}$, we represent a call to f with the list of arguments *args* by $f(args)$. Then, $f(args)\{\phi\}$ is the $\mathbb{K}$ pattern built by inserting the call $f(args)$ at the top of the k cell and initializing the path condition cells with ϕ. This is helpful to start the execution of $f(args)$ under the (possibly non–empty) constraints of ϕ. We also denote by $\textsc{Se}(f(args)\{\phi\})$ the set of final patterns obtained from the symbolic execution of the pattern $f(args)\{\phi\}$ (i.e., the leaves of the deployed symbolic execution tree).

Our specification inference methodology is formalized in Algorithm 1. First, the *modifier* method of interest m is symbolically executed with fresh symbolic variables $\mathsf{a}_1, \ldots, \mathsf{a}_n$ as arguments and empty constraint `true`, and the set S of final patterns is retrieved from the leaves of the symbolic execution tree. For each pattern in S, the corresponding path condition is simplified (by calling the automated theorem prover Z3) to avoid redundancies and simplify the analysis. Then, we proceed to compute an axiom for each pattern p in S that explains (by using the observers) the properties that hold in the state before and after the execution of the method. This is done by means of the function $explain(q, as)$, where q is a pattern and as is a list of symbolic variables, given in Algorithm 2. The explanation for initial states whose symbolic execution end in p (line 4) must ensure that the input data comply with the conditions that make the path to p feasible, which is achieved by imposing the conditions given by ϕ to the input pattern to be explained (i.e. by feeding its heap cell with φ before invoking the routine *explain*). Then, we proceed to explain (also with the observers) the properties of the considered final state (the final pattern p) by invoking $explain(p, [\mathsf{a}_1, \ldots, \mathsf{a}_n])$. Finally, the return value v is retrieved from the k cell of p, and the axiom $ret = \mathsf{v}$ is added to the specification inferred. This value could be either undefined or a single typed value that represents the return from the function m under the conditions given by ϕ.

Algorithm 1 Specification Inference.

```
Require: $m \in \mathscr{M}$ of arity $n$;
 1. $S = \textsc{Se}(m(\mathsf{a}_1, \ldots, \mathsf{a}_n)\{\texttt{true}\})$
 2. $axiomSet := \emptyset$;
 3. for all $p \in S$ with path-condition cell $\phi_p$, init-struct cell $\varphi$ and return value $\mathsf{v}$ do
 4.   $eqs_{pre} := explain(\langle\langle m(\mathsf{a}_1, \ldots, \mathsf{a}_n)\rangle_{\mathsf{k}} \langle\varphi\rangle_{\mathsf{heap}} \cdots \rangle_{\mathsf{cfg}} \langle\phi_p\rangle_{\mathsf{path-condition}}, [\mathsf{a}_1, \ldots, \mathsf{a}_n])$;
 5.   $eqs_{post} := explain(p, [\mathsf{a}_1, \ldots, \mathsf{a}_n])$;
 6.   $eq_{ret} := (ret = \mathsf{v})$;
 7.   $axiomSet := axiomSet \cup \{eqs_{pre} \Rightarrow (eqs_{post} \cup eq_{ret})\}$;
 8. end for
 9. spec := $simplify(axiomSet)$
10. return spec
```

The computed axioms are implications of the form $l_i \Rightarrow r_i$, where l_i is a conjunction of preconditions and r_i is a conjunction of postconditions. Note that a conjunction of equations is represented as an equation set in Algorithm 2. The function *simplify* implements a post-processing which consists of: (1) disjoining the preconditions l_i that have the same postcondition r_i and simplifying the resulting precondition; and (2) conjoining the postconditions r_i that share the same precondition and simplifying the resulting postcondition.

Let us illustrate the application of the inference algorithm with the following example.

Example 7 *Let us compute a specification for the* `append` modifier *function of Example 1. Following the algorithm, we first compute* $\textsc{Se}(\texttt{append(list,d)}[\texttt{true}])$ *with* `list` *and* `d` *(free) symbolic variables.*

Since there are no constraints in the initial symbolic configuration, the execution covers all possible initial concrete configurations. For simplicity, we set the number of loop unrollings to one; as a consequence, the symbolic execution computes three final patterns. The following pattern e represents the final state for the path where the body of the `while` *statement never gets executed (0 iterations):*

$$\left\langle \begin{array}{c} \langle \mathsf{tv}(\mathit{struct\ \ List*},\ \mathsf{list}) \rangle_{\mathsf{k}} \\ \langle \mathtt{list} \mapsto \mathsf{list}, \mathtt{d} \mapsto \mathsf{d}, \mathtt{new\_node} \mapsto \mathsf{new\_node}, \mathtt{final} \mapsto \mathsf{list} \rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto (\mathtt{data} \mapsto \mathrm{undef}, \mathtt{prev} \mapsto \mathrm{undef}, \mathtt{next} \mapsto \mathsf{new\_node}) \\ \mathsf{new\_node} \mapsto (\mathtt{data} \mapsto \mathsf{d}, \mathtt{prev} \mapsto \mathsf{list}, \mathtt{next} \mapsto \mathsf{NULL}) \\ \mathsf{d} \mapsto \mathsf{tv}(\mathit{void},\ ?\mathsf{d}) \end{array} \right\rangle_{\mathsf{heap}} \end{array} \right\rangle_{\mathsf{cfg}} \left\langle \begin{array}{c} \mathsf{list} \neq \mathsf{NULL} \wedge \\ \mathsf{list} \Rightarrow \mathtt{next} = \mathsf{NULL} \end{array} \right\rangle_{\mathsf{init-struct}}$$

The execution of this path returns the pointer to the resulting list: the returned pointer is represented by the typed value $\mathsf{tv}(struct\ List*, \mathsf{list})$ *in the* k *cell. The field* $\mathsf{list} \Rightarrow \mathtt{next}$ *is accessed only after checking that* list *is not null: it has been assumed* `list != NULL` *at the first conditional expression, thus the constraint* $\mathsf{list} \neq \mathsf{NULL}$ *has been added to the path condition, whereas* `final->next != NULL` *(the guard of the* `while` *loop) is assumed false, thus the constraint* $\mathsf{list} \Rightarrow \mathtt{next} = \mathsf{NULL}$ *has been gathered. Note that, although the variable accessed in the code is* `final`*, the generated path constraint refers to the pointer* list *since both* `final` *and* `list` *are bound to the same memory address in the environment.*

Let us now describe Algorithm 2 which defines the function $explain(q, as)$. Given a $\mathbb{K}$ pattern q and a list of symbolic variables as, this function describes q as a set of equations that are obtained by executing the observer functions in the state. Each equation relates the call to an observer function (or *built-in* function) with the (symbolic) value that the call returns. In the algorithm, $As \sqsubseteq as$ means that the list of elements As is a permutation of some (or all) elements in as.

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

```
Require: $q$ : the pattern to be explained (with path condition $\phi$)
Require: $as$ : a list of symbolic variables
1. $\mathscr{C}$: the universe of observer calls;
2. $eqSet := \emptyset$;
3. for all $o(As) \in \mathscr{C}$ with $As \sqsubseteq as$ do
4.     $S = \mathrm{SE}(o(As)\{\phi\})$
5.     if $\nexists\ q_1, q_2 \in S$ s.t. $q_1$ and $q_2$ contain a different return value $\mathsf{k}$ in their k cell then
6.         $eqSet := eqSet \cup (o(As) = \mathsf{k})$
7.     end if
8. end for
9. return $eqSet$
```

Roughly speaking, given a pattern q, $explain(q, as)$ first generates the universe of observer function calls $\mathscr{C}$, which consists of all the function calls $o(As)$ that satisfy that:

- o belongs to $\mathscr{O}$ or to the set of (predefined) built-in functions,
- the argument list $As \sqsubseteq as$ respects the type and arity of o.

Then, for each call $o(As) \in \mathscr{C}$, Algorithm 2 checks whether all the final symbolic configurations (leaves) resulting from the symbolic execution of $o(As)$, under the constraints given by ϕ, have the same return value. When the call satisfies this requirement, an equation is generated (line 6 in Algorithm 2). Otherwise, the observation is inconclusive and no explanation is delivered in terms of the executed observer function. The algorithm finally returns the set of all the explanatory equations inferred.

Example 8 (Example 7 continued) *Let us show how we compute the explanation for the final state of pattern p in Example 7. Given the observer functions* `length`*,* `reverse`*,* `head`*,* `last`*,* `find`*, and* `init`*, and the symbolic variables* `list` *and* `d`*, the universe of observer calls is* `length(list)`*,* `reverse(list)`*,* `head(list)`*,* `last(list)`*,* `find(list,d)`*, and* `init(list)`*. Let us consider the case for the observer call* `length(list)` *in detail.*

When we symbolically execute `length(list)` *on the pattern p, we obtain a single final pattern:*

$$\left\langle \begin{array}{c} \langle \mathrm{tv}(\mathit{int},\, 2)\rangle_{\mathsf{k}} \\ \langle \texttt{list} \mapsto \mathsf{list}, \texttt{length} \mapsto \mathsf{length}\rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto \big(\texttt{data} \mapsto \mathrm{undef}, \texttt{prev} \mapsto \mathrm{undef}, \texttt{next} \mapsto \mathsf{new\_node}\big) \\ \mathsf{new\_node} \mapsto \big(\texttt{data} \mapsto \mathsf{d}, \texttt{prev} \mapsto \mathsf{list}, \texttt{next} \mapsto \mathsf{NULL}\big) \\ \mathsf{d} \mapsto \mathrm{tv}(\mathit{void},\, ?\mathsf{d}) \\ \mathsf{length} \mapsto \mathrm{tv}(\mathit{int},\, 2) \end{array} \right\rangle_{\mathsf{heap}} \end{array} \right\rangle_{\mathsf{cfg}} \left\langle \begin{array}{c} \mathsf{list} \neq \mathsf{NULL} \wedge \\ \mathsf{list} \Rightarrow \texttt{next} = \mathsf{NULL} \end{array} \right\rangle_{\mathsf{init-struct}}$$

Since there are no observer paths returning different values and the associated return value is the integer 2, then the equation $\texttt{length}(\texttt{list}) = 2$ *is computed as a (partial) explanation for the final pattern under consideration. Thus, this term is added to the set of equations eqSet that are computed by Algorithm 2.*

A preliminary implementation of our specification inference methodology has been developed in the prototype system KINDSPEC 2.0, that is available at `safe-tools.dsic.upv.es/kindspec2`. The system is built upon a centralized call-and-return architecture that is shown in Figure 3, where the main, front-end module orchestrates the different subprocesses of the inference algorithm and glues together their results. Given the source code of the program to be analyzed and the program function *f* whose specification needs to be inferred, first the system invokes the 𝕂 interpreter, `krun`, providing it (through the coupling module) the name of the modifier function *f* and an initial empty state. By using the compiled definition of the programming language (KERNELC, in our case), the 𝕂 interpreter carries out the symbolic execution of *f*. As a result, a textual representation of the leaves of the symbolic tree is returned, and then parsed in order to obtain a higher level representation based on internal data structures and objects. Once the information retrieved from the final states of the execution is available, the explanation module synthesizes the specification as described in Algorithm 2, and then a conveniently simplified version of the axioms is output to the user.

The specification computed for our leading example is shown below:

$$\left(\begin{array}{l} \texttt{length}(\texttt{list}) = 0 \wedge \\ \texttt{reverse}(\texttt{list}) = \texttt{NULL} \wedge \\ \texttt{find}(\texttt{list},\texttt{data}) = 0 \wedge \\ \texttt{init}(\texttt{list}) = \texttt{NULL} \wedge \\ \texttt{last}(\texttt{list}) = \texttt{NULL} \end{array} \right) \quad \Rightarrow \quad \left(\begin{array}{l} \texttt{length}(\texttt{list}') = 1 \wedge \\ \texttt{reverse}(\texttt{list}') = \texttt{list} \wedge \\ \texttt{find}(\texttt{list}',\texttt{data}) = 1 \wedge \\ \texttt{init}(\texttt{list}') = \texttt{NULL} \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{array} \right)$$

$$\left(\begin{array}{l} \texttt{length}(\texttt{list}) = 1 \end{array} \right) \quad \Rightarrow \quad \left(\begin{array}{l} \texttt{length}(\texttt{list}') = 2 \wedge \\ \texttt{find}(\texttt{list}',\texttt{data}) = 1 \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{array} \right)$$

$$\left(\begin{array}{l} \texttt{length}(\texttt{list}) = 2 \end{array} \right) \quad \Rightarrow \quad \left(\begin{array}{l} \texttt{length}(\texttt{list}') = 3 \wedge \\ \texttt{find}(\texttt{list}',\texttt{data}) = 1 \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{array} \right)$$

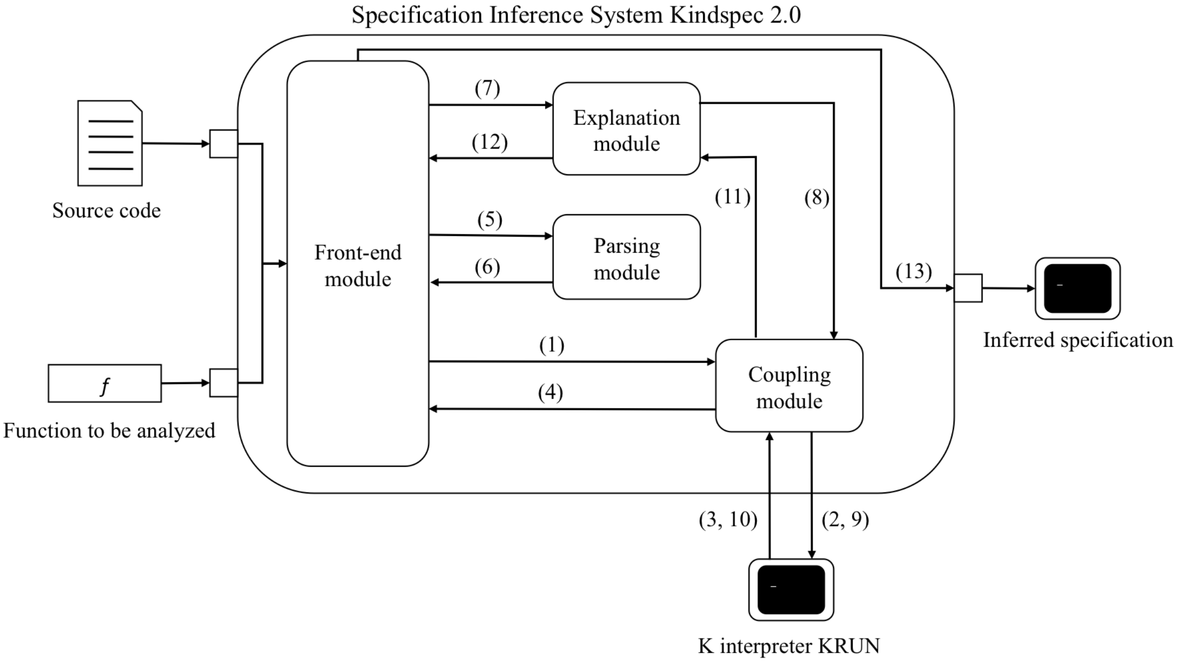

Figure 3: Architecture of the inference system KINDSPEC 2.0.

Note that, in contrast to the two axioms of Example 2, three axioms are computed. This is due to the unrolling of loops. Note that the second and third computed axioms are instances of the second axiom of the intended specification.

Similarly to [1], due to bounded loop unrolling we cannot ensure completeness of the inferred specifications since we do not cover all possible execution paths. This is evident when comparing the automatically inferred axioms shown in the pattern above w.r.t. the expected specification given in Example 2. An effective generalization methodology is needed to properly cover all possible executions without incurring (hopefully) in significant loss of correctness. We informally discuss our key ideas towards this endeavour in the following subsection.

5.1 Future directions

We are currently working on defining a *generalization* algorithm that can distill more general axioms (such as the second axiom in Example 2) that we are not yet able to obtain. We follow the common synthesis approach that is based on using “skeletons” of generalizations, which are then refined to obtain a correct generalization of a set of axioms (w.r.t. the skeleton). The function that computes such skeletons basically induces them from iterations (loops and recursive calls) and is considered to be a parameter of the algorithm. Without entering into too much detail, candidate skeletons are given by a so-called “admissible template”, that is, a non-ground $\mathbb{K}$ term that is used to guess the form that a given general axiom can have. A common drawback when resorting to skeletons is that the burden of defining/selecting the most suitable templates for a given problem usually rests with the user; hence usability is a key point that we cannot dismiss. Even if extensive research is still needed, our preliminary experiments reveal that axioms like the aforementioned more general one can be easily inferred automatically. Obviously, since we are using a threshold to stop loops, correctness cannot be ensured for all the general axioms that we compute, but they can still be useful for other verification processes or even be verified afterwards. A second, longer-term direction for research is to follow the abstraction-based, subsumption approach for symbolic execution of [2] to finitize symbolic execution while getting rid of any thresholds.

From the experimental point of view, there are certainly several ways that our prototype implementation can be improved. A refinement post-processing was defined in [1] that improves the quality of inferred specifications. Roughly speaking, when an observed pattern cannot be explained because its symbolic execution leads to final patterns that do not agree in the same result, the call pattern is (incrementally) split into multiple refined patterns until the considered observers eventually suffice to explain it. We plan to implement this refinement process in KindSpec 2.0 and measure the inference power gains. Actually, the main motivation of our work was not to improve efficiency but rather to improve robustness, generality and mantainability.

References

[1] M. Alpuente, M. A. Feliú & A. Villanueva (2013): *Automatic Inference of Specifications using Matching Logic*. In: *Proc.e ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013*, ACM, pp. 127–136, doi:10.1145/2426890.2426914.

[2] S. Anand, C. S. Pasareanu & W. Visser (2009): *Symbolic execution with abstraction*. *International Journal on Software Tools for Technology Transfer (STTT)* 11(1), pp. 53–67, doi:10.1007/s10009-008-0090-1.

[3] A. Arusoaie, D. Lucanu, V. Rusu, T.-F. Serbanuta, A. Stefanescu & G. Roşu (2014): *Language Definitions as Rewrite Theories*. In: *10th International Workshop on Rewriting Logic and Its Applications (WRLA), Revised Selected Papers*, pp. 97–112, doi:10.1007/978-3-319-12904-4_5.

[4] G. Bacci, M. Comini, M. A. Feliú & A. Villanueva (2012): *Automatic Synthesis of Specifications for First Order Curry Programs*. In: *Proc. of the 14th Intl. Symp. on ACM Principles and Practice of Declarative Programming (PPDP'12)*, ACM Press, pp. 25–34, doi:10.1145/2370776.2370781.

[5] M. Christodorescu, S. Jha & C. Kruegel (2007): *Mining Specifications of Malicious Behavior*. In: *Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, ACM, pp. 5–14, doi:10.1145/1287624.1287628.

[6] K. Claessen, N. Smallbone & J. Hughes (2010): *QuickSpec: Guessing Formal Specifications Using Testing*. In: *Proc, 4th Int'l Conf. on Tests and Proofs (TAP 2010)*, *Lecture Notes in Computer Science* 6143, Springer, pp. 6–21, doi:10.1007/978-3-642-13977-2_3.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer & C. Talcott (2007): *All About Maude: A High-Performance Logical Framework*. *Lecture Notes in Computer Science* 4350, Springer-Verlag, doi:10.1007/978-3-540-71999-1_7.

[8] C. Csallner, N. Tillmann & Y. Smaragdakis (2008): *DySy: Dynamic Symbolic Execution for Invariant Inference*. In: *Proc. 30th International Conference on Software Engineering (ICSE 2008)*, ACM, pp. 281–290, doi:10.1145/1368088.1368127.

[9] V. D'Silva, D. I. Kroening & G. Weissenbacher (2008): *A Survey of Automated Techniques for Formal Software Verification*. *IEEE Trans. on CAD of Integrated Circuits and Systems* 27(7), pp. 1165–1178, doi:10.1109/TCAD.2008.923410.

[10] C. Ellison & G. Roşu (2012): *An Executable Formal Semantics of C with Applications*. In: *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, ACM, pp. 533–544, doi:10.1145/2103656.2103719.

[11] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz & C. Xiao (2007): *The Daikon System for Dynamic Detection of Likely Invariants*. *Sci. Comput. Program.* 69(1-3), pp. 35–45, doi:10.1016/j.scico.2007.01.015.

[12] C. Ghezzi, A. Mocci & M. Monga (2009): *Synthesizing Intensional Behavior Models by Graph Transformation*. In: *Proc. 3st Int'l Conf. on Software Engineering (ICSE 2009)*, IEEE, pp. 430–440, doi:10.1109/ICSE.2009.5070542.

[13] D. Giannakopoulou & C. S. Pasareanu (2009): *Interface Generation and Compositional Verification in JavaPathfinder*. In: *Proc. 12th In'l Conf. on Fundamental Approaches to Software Engineering (FASE 2009)*, *Lecture Notes in Computer Science* 5503, Springer, pp. 94–108, doi:10.1007/978-3-642-00593-0_7.

[14] S. Hangal & M. S. Lam (2002): *Tracking down Software Bugs using Automatic Anomaly Detection*. In: *Proc. 22rd International Conference on Software Engineering (ICSE 2002)*, ACM, pp. 291–301, doi:10.1145/581339.581377.

[15] J. Henkel & A. Diwan (2003): *Discovering Algebraic Specifications from Java Classes*. In: *Proc. ECOOP*, pp. 431–456, doi:10.1007/978-3-540-45070-2_19.

[16] S. Khurshid, C. S. Pasareanu & W. Visser (2003): *Generalized Symbolic Execution for Model Checking and Testing*. In: *TACAS*, pp. 553–568, doi:10.1007/3-540-36577-X_40.

[17] J. C. King (1976): *Symbolic execution and program testing*. *Commun. ACM* 19(7), pp. 385–394, doi:10.1145/360248.360252.

[18] B. Liskov & J. Guttag (1986): *Abstraction and specification in program development*. MIT Press.

[19] L. M. de Moura & B. Nikolaj (2008): *Z3: An Efficient SMT Solver*. In: *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[20] C. S. Pasareanu & W. Visser (2009): *A Survey of new Trends in Symbolic Execution for Software Testing and Analysis*. *STTT* 11(4), pp. 339–353, doi:10.1007/s10009-009-0118-1.

[21] G. Roşu (2015): *From Rewriting Logic, to Programming Language Semantics, to Program Verification*. In: *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, *Lecture Notes in Computer Science* 9200, Springer Verlag, pp. 598–216, doi:10.1007/978-3-319-23165-5_28.

[22] G. Roşu, W. Schulte & T.-F. Serbanuta (2009): *Runtime Verification of C Memory Safety*. In: *Runtime Verification (RV'09)*, *Lecture Notes in Computer Science* 5779, pp. 132–152, doi:10.1007/978-3-642-04694-0_10.

[23] G. Roşu & T.-F. Serbanuta (2010): *An Overview of the $\mathbb{K}$ Semantic Framework*. *J. Log. Algebr. Program.* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[24] G. Roşu & A. Stefanescu (2011): *Matching Logic: A New Program Verification Approach*. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, ACM, pp. 868–871, doi:10.1145/1985793.1985928.

[25] M. Taghdiri & D.Jackson (2007): *Inferring Specifications to Detect Errors in Code*. *Autom. Softw. Eng.* 14(1), pp. 87–121, doi:10.1007/s10515-006-0005-x.

[26] N. Tillmann, F. Chen & W. Schulte (2006): *Discovering Likely Method Specifications*. In: *Proc. 8th Int'l Conf. on Formal Engineering Methods (ICFEM 2006)*, *Lecture Notes in Computer Science* 4260, Springer, pp. 717–736, doi:10.1007/11901433_39.

[27] J. Whaley, M. C. Martin & M. S. Lam (2002): *Automatic extraction of object-oriented component interfaces*. In: *Proc. ISSTA 2002*, pp. 218–228, doi:10.1145/566172.566212.