Automatic Inference of Specifications using Matching Logic [*]

María Alpuente Marco A. Feliú [†] Alicia Villanueva

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

Abstract

Formal specifications can be used for various software engineering activities ranging from finding errors to documenting software and automatic test-case generation. Automatically discovering specifications for heap-manipulating programs is a challenging task. In this paper, we propose a technique for automatically inferring formal specifications from C code which is based on the symbolic execution and automated reasoning tandem "MATCHING LOGIC / $\mathbb{K}$ framework". We implemented our technique for a fragment of C, called KERNELC, in the automated tool KINDSPEC, 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. These specifications can be written either in MATCHING LOGIC itself, which is useful for further automated analysis within the $\mathbb{K}$ formal environment, or in sugared axiomatic form, which favors better human inspection. Since we rely on rewriting logic $\mathbb{K}$ semantics specification of programming languages, our approach can be easily extended to any language for which a formal semantics in $\mathbb{K}$ is given.

Categories and Subject Descriptors F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs

Keywords Specifications inference, Symbolic execution, Matching Logic

1. Introduction

Formal specifications can document code unambiguously and are important for rigorous software development. Algorithm verification and program testing can often diagnose discrepancies between implementation and specification automatically. Unfortunately, formal specifications are notoriously hard to write and debug, and many programs lack appropriate documentation or it may be too low-level to understand. Specification inference can help to mitigate these problems and is also useful for legacy program understanding and malware deobfuscation, where the challenge is to understand what the malicious code is doing [5]. This paper describes a rule-based technique that automatically infers high-level, formal specifications for C-like, heap-manipulating code by using the notation of MATCHING LOGIC ($\mathbb{ML}$), which is a novel program verification foundation that is built upon operational semantics [25]. The considered KERNELC language [26] is a non-trivial fragment of C that includes functions, structures, pointers and I/O primitives. Besides the code itself, in $\mathbb{ML}$, as in operational semantics, program states contain an encoding for the environment, the heap, stacks, etc., which are represented as algebraic datatypes, called (concrete) configurations. Axiomatic specifications of program states are represented as configuration *patterns* that are a particular class of first-order formulas with equality. Patterns are encoded as (boolean) terms with variables and constraints over them. A pattern specifies those configurations that match its algebraic structure and satisfy its constraints. For example, the pattern

$$\langle\langle \cdots\ \mathtt{top} \mapsto \mathtt{?top}\ \cdots\ \rangle_{\mathsf{env}} \langle \cdots\ \mathrm{list}(\mathtt{?top})(\mathtt{L})\ \cdots\ \rangle_{\mathsf{heap}}\rangle_{\mathsf{cfg}} \wedge \mathtt{L} \neq \mathrm{empty}$$

specifies the set of configurations where the program variable `top` points to the non-empty list `L`. Hence, separation (meaning that the heap can be split into two disjoint parts where the separate formulas hold [21]), is achieved at the structural (i.e., term) level. Marked variables like `?top` are bound (i.e., existentially quantified) over the pattern, while `L` is free. By allowing specifications to directly refer to the structure of the program configurations, $\mathbb{ML}$ facilitates access and reasoning about rich sub-patterns of the program states, such as disjoint lists, trees and graphs, or any shared mutable data structures that are dynamically allocated in the heap.

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 [18, 20]. Intuitively, symbolic execution means that each data structure field and program variable initially hold a symbolic value. Then, each program statement execution can update the configuration cells (such as *env*, *heap* and *cfg* in the example above) by mapping fields and variables to (symbolic) values represented as relational expressions. Recently, SE has found renewed interest due in part to the advances in new algorithmic developments and decision procedures. The symbolic execution of KERNELC programs is supported in $\mathbb{ML}$ by using the MATCHC verifier, which has several applications including the verification of functional correctness and static detection of runtime errors [25]. MATCHC is implemented using the $\mathbb{K}$ language definitional frame-

[*] This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, ref. TIN 2010-21062-C02-0, and by Generalitat Valenciana, ref. PROMETEO2011/052.

[†] This author was partially supported by the Spanish MEC FPU grant AP2008-00608

PEPM'13, January 21–22, 2013, Rome, Italy
Copyright © 2013 ACM 978-1-4503-1842-6/13/01. . . $15.00

work [24], which compiles into the high-performance programming language Maude [7].

In this paper, we develop a technique for discovering formal specifications for heap-manipulating programs by using the symbolic infrastructure of MatchC, which is driven by $\mathbb{ML}$ formulas. We needed to extend MatchC in order to support the analysis for the inference of specifications. The key idea behind our method is as follows. We have developed a procedure on top of MatchC that is executed symbolically in $\mathbb{K}$. The execution of the symbolic procedure delivers an environment for the post-state that gives the value of each variable and field in terms of the values of program variables and fields of the abstract pre-state. Following the common symbolic approach to finitize program execution [27], a loop is handled by unrolling it to a fixed depth, i.e., the loop is executed a fixed number of times. We also limit the heap's size so that the space of possible heaps is also finitized.

The proposed inference technique relies on the classification scheme developed in [2] for data abstractions in general, where a function (method) may be either a *constructor*, *modifier* or *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 [2] 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, maybe written by third-party software producers that may not follow the observer purity discipline.

Our symbolic analysis of KernelC programs allows us to explain the execution of a *modifier* function m by using other (observer) routines in the program. Starting from an initial symbolic state s^0, we first evaluate symbolically m on s_0 obtaining as a result a set of pairs (s_i^0, s_i^f) of refined initial and final symbolic states, respectively. In order to compute suitable explanations for the routine m, we symbolically evaluate the observer methods on each state s_i^0 and s_i^f so that when the observer returns the same value at the end of each of its branches, then we can say that the observer is a (partial) observational abstraction or explanation of the constraints in the state. For each pair of refined 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_i^0, whereas the postcondition contains the observers that *explain* the final state s_i^f. Then, the synthesized pre/post axioms that abstract from any implementation details are further simplified (to be given more compact representation), and are eventually presented in a more friendly sugared form.

We have applied our methodology on KernelC implementations of *collection* libraries for manipulating linked lists in C such as those provided in the GDSL generic data structure library [1]. To evaluate our method, we performed two experiments using the prototype tool implementation KindSpec. In one experiment, we applied it to a small component for which complete specifications were already available: a standard implementation of sets in C by using linked lists. In the second experiment, we applied the tool to other kinds of heap-allocated mutable structures. These components are more typical of object-oriented style code written in C. We wrote specifications for the source code in the style of the specifications our tool infers; this allowed us to assess the accuracy of our generated specifications, which properly target the properties that constrain the structure of the data in the heap. Actually, we found that only in a few cases there was a significant loss of information.

In summary, the main contributions of this paper are as follows:

- A new approach to extract lightweight specifications from heap-manipulating code that consists of a symbolic analysis that explores and summarizes the behavior of a *modifier* program routine by using other available routines in the program, called *observers*.
- Correctness conditions for our specification discovery technique;
- A practical demonstration that the technique is capable of extracting accurate specifications from nontrivial procedures and functions;
- An implementation of the framework that targets KernelC programs and uses $\mathbb{K}$ as an intermediate language to translate KernelC to $\mathbb{ML}$ constraints. Specification inference takes advantage of the unsat core generated by the SAT solver CVC3 [4] that is coupled to $\mathbb{ML}$.

This paper improves existing approaches in the literature in several ways. On one hand, our technique is the first approach that can automatically ensure delivery of correct specifications, which is done by using the same MatchC verifier that we use for the specification discovering. On the other hand, since our approach relies on the $\mathbb{K}$ semantics specification of KernelC, the methodology developed in this work can be easily extended to cope with any language for which a $\mathbb{K}$ semantics is given, like Java 1.4, Scheme and Verilog [24]. There is an executable formal semantics for C that describes the semantics of the whole C99 standard, and it will be possible to use it in our framework as soon it is coupled into the MatchC verifier.

Plan of the Paper In the next section, we review the concepts of the $\mathbb{K}$ framework that are crucial for this work. Section 3 introduces the case that has been used to evaluate the adequacy, performance, and effectiveness of the proposed inference method and which serves as a running example throughout this paper. It also allows us to outline the major research problems addressed. Section 4 describes how we extended the symbolic machinery in order to support our inference approach. Section 5 introduces two algorithms that mechanize the inference technique and provides some experimental results. Finally, Section 6 discusses the related work and concludes.

2. Preliminaries

In this section, we recall the fundamental concepts of the $\mathbb{K}$ framework and Matching Logic.

2.1 The $\mathbb{K}$ framework

$\mathbb{K}$ is an executable semantic framework in which programming languages, calculi, as well as type systems and formal analysis tools can be defined making use of configurations, computations, and rules. The most complete formal program semantics in the literature for Scheme, Java 1.4, Verilog, and C are currently available in $\mathbb{K}$. $\mathbb{K}$ semantics are compiled into Maude [7] for execution, debugging, and model checking.

Program configurations are represented in $\mathbb{K}$ as potentially nested structures of labeled cells that represent the state of the program. $\mathbb{K}$ cells are containers that can act as lists, maps, (multi)sets of computations, or as a multiset of other cells. Computations carry "computational meaning" as special nested list structures that sequentialize computational tasks, such as fragments of a program. Rules in $\mathbb{K}$ state how configurations (terms) can evolve during computation.

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

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

Containers (or cells) in a configuration represent pieces of the program state, including a computation stack or continuation (named k), environments (env, heap), and a call stack (stack), among others.

Rules are graphically represented in two levels and state how configurations change. Changes in the current configuration (which is shown in the upper level) are explicitly represented by underlying 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 two cells, k and env. The env cell is a mapping of variables to their values, whereas 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.

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

The 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 with the new value V. The typed value $\mathrm{tv}(T,V)$ 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 $\cdots$ card).

This example rule reveals a 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 "$\cdots$ ". For example, the rule above emphasizes the interest in: the instruction $X = \mathrm{tv}(T,V)$ only at the beginning of the k cell, and the mapping from variable X to any value "_" at any position in the env cell.

2.2 Matching Logic

MATCHING LOGIC is a logic for the verification of programs. We use the $\mathbb{ML}$ formalization given in [22, 23]. Formulas in $\mathbb{ML}$ are called *patterns* and they represent sets of concrete program configurations. Patterns can be formalized as first-order logic (FOL) formulas whose atomic propositions include the constructs for representing program configurations. Intuitively, patterns introduce logical variables into program configurations and also introduce formulas that constrain those logical variables. Pattern variables are typed.

Starting from the operational semantics of KERNELC specified in $\mathbb{K}$ that we mentioned above, a handy axiomatic semantics that is based on $\mathbb{ML}$ patterns has been systematically defined for verification purposes within the $\mathbb{ML}$ framework [23]. The $\mathbb{ML}$ patterns of the KERNELC axiomatic semantics consist of $\mathbb{K}$ configuration cells that are additionally constrained with FOL formulas. We use the sugared notation of [22, 23] that embeds the FOL formula within a special cell called ϕ and substitutes explicit existential logical quantifiers with a special mark ? at the beginning of the existentially quantified variable's name. For example, the set of KERNELC configurations where the specific program variable `x` holds a value that is greater than `5` can be represented with the following $\mathbb{ML}$ pattern, where free variable E matches any other information in the env cell, and C matches the rest of the cells in the configuration:

$$\Big\langle \ \langle \mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E} \rangle_{\mathsf{env}} \langle ?\mathsf{x} > 5 \rangle_{\phi} \ \mathsf{C} \ \Big\rangle_{\mathsf{cfg}} \qquad (1)$$

Program variables[1] in $\mathbb{ML}$ formulas such as `x` are written in *teletype* font and logical variables such as x are written in sans font. Metavariables in $\mathbb{ML}$ are written in capital letters.

The $\mathbb{ML}$ axiomatic semantics uses inference rules to derive sequents $\mathrm{P}_1 \Downarrow \mathrm{P}_2$, called *correctness pairs*, which relate patterns before and after the execution of a program fragment. The meaning of a sequent $\mathrm{P}_1 \Downarrow \mathrm{P}_2$ is that the execution of a concrete configuration matching P_1 yields a configuration matching P_2. The basic axiom for final symbolic configurations that indicate the end of a successful (axiomatic semantics) derivation is as follows:

$$\frac{}{\langle \Gamma \rangle_{\mathsf{cfg}} \Downarrow \langle \Gamma \rangle_{\mathsf{cfg}}}$$

with Γ being a final pattern, i.e., a pattern whose k cell is empty or consists of just a value. As an example of $\mathbb{ML}$ inference rule, let us describe the rule that corresponds to the assignment of a program variable.

$$\frac{\langle \langle \mathsf{K} \rangle_{\mathsf{k}} \mathsf{C} \rangle_{\mathsf{cfg}} \Downarrow \langle \langle \mathsf{I} \rangle_{\mathsf{k}} \langle \rho \rangle_{\mathsf{env}} \mathsf{C}' \rangle_{\mathsf{cfg}}}{\langle \langle \mathsf{X} = \mathsf{K}; \rangle_{\mathsf{k}} \mathsf{C} \rangle_{\mathsf{cfg}} \Downarrow \langle \langle \rangle_{\mathsf{k}} \langle \rho[\mathsf{X} \mapsto \mathsf{I}] \rangle_{\mathsf{env}} \mathsf{C}' \rangle_{\mathsf{cfg}}}$$

In the rule above, K matches a computation expression, I an integer value, X a program variable, ρ a map, and C and C' a bag of configuration cells (including the ϕ cell). The rule hypothesis can be read as: «starting from any configuration with K as the only remaining computation in the k cell, assume that a configuration that matches $\langle \langle \mathsf{I} \rangle_{\mathsf{k}} \langle \rho \rangle_{\mathsf{env}} \mathsf{C}' \rangle_{\mathsf{cfg}}$ is obtained, with I being the result of evaluating the expression K». Then, the conclusion can be read as: «starting from a configuration that consists of C plus the assignment of expression K to the variable identified by X as the only remaining computation, its execution yields a configuration consisting of C', an empty k cell (i.e., no pending computations remain), and an updated environment $\rho[\mathsf{X} \mapsto \mathsf{I}]$ in which I is assigned to the variable X ».

The following example illustrates the application of the assignment inference rule.

EXAMPLE 1. *Let us consider the following (axiomatic semantics) symbolic configuration, which is an instance of the $\mathbb{ML}$ pattern of equation (1) and contains the assignment instruction* `x=3` *in the* k *cell:*

$$\Big\langle \ \langle \mathtt{x} = 3; \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E} \rangle_{\mathsf{env}} \langle ?\mathsf{x} > 5 \rangle_{\phi} \mathsf{C}'' \ \Big\rangle_{\mathsf{cfg}}$$

This symbolic configuration is also a pattern that matches the left-hand pattern of the correctness pair in the consequent of the inference rule. Moreover, the antecedent of the rule holds since

$$\langle \langle 3 \rangle_{\mathsf{k}} \mathsf{C} \rangle_{\mathsf{cfg}} \Downarrow \langle \langle 3 \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E} \rangle_{\mathsf{env}} \mathsf{C}' \rangle_{\mathsf{cfg}}$$

with $\mathsf{C} = \langle \mathtt{x} \mapsto ?\mathsf{x}, \mathsf{E} \rangle_{\mathsf{env}} \ \mathsf{C}'$ and $\mathsf{C}' = \langle ?\mathsf{x} > 5 \rangle_{\phi} \mathsf{C}''$. This is because, according to MAUDE *(and $\mathbb{K}$) type systems,* `3` *is both an integer value and a program expression, thus it respectively matches the I and the K variables in the rule hypothesis. Therefore, the derivation computed by the application of the rule is*

$$\Big\langle \langle \mathtt{x} = 3; \rangle_{\mathsf{k}} \mathsf{C} \Big\rangle_{\mathsf{cfg}} \Downarrow \Big\langle \langle \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto 3, \mathsf{E} \rangle_{\mathsf{env}} \mathsf{C}' \Big\rangle_{\mathsf{cfg}}$$

Note that $\mathbb{ML}$ does not need logical "separation" because it achieves it at the structural level. That is, any pair of subterms in a pattern configuration that are not related by the containment order are considered to be distinct and disjoint; and, if a pattern matches two terms in a multiset, the two terms have to be distinct. For example, in pattern (1), the binding $\mathtt{x} \mapsto ?\mathsf{x}$ is matched with the current configuration separately from the rest of the environment E, and, thus, no overlapping of bindings can occur.

[1] Note that program variables are constants in $\mathbb{ML}$ formulas.

3. Specification Discovery

A logic specification is a logical relation between inputs and outputs of a program. Specification discovery is the task of inferring high-level specifications that closely describe the program behavior. Obviously, these specifications can only be correct with respect to user intent if the original program is correct itself. But even if it is not correct, the ascertained specification can still be very helpful in several important scenarios such as improving program understanding, synthesizing test units, and helping the programmer to debug the code.

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*, that 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*. As a consequence, we simply define an *observer* as any function whose return type is different from void, hence, potentially expressing an observed property regarding the value of the function arguments or the contents in the *heap*.

The following example introduces the case that we use as a running example throughout this paper.

EXAMPLE 2. *The program in Figure 1 implements an abstract datatype for representing sets. A set is internally represented in* KERNELC *as a data structure (*struct set*) that contains a pointer (*struct lnode **) to a list of elements (field* elems*), the number of elements in the set (field* size*) and the maximum number of elements that may contain (field* capacity*).*

The new *function allocates memory for storing a* struct set *data structure with initial size* 0*, the capacity given by the input value for the* capacity *parameter, and the* NULL *value for the pointer that references the list of elements in the set. Upon completion, it returns the address of the allocated structure.*

A call add(s,x) *to the* add *function proceeds as follows: it first checks that the pointer* s *to the set is different from* NULL*; next, it checks that the size of* s *is lower than its capacity; and then, it checks that* x *is not an element of* s *yet. Provided all these conditions hold, it allocates a new list node (*struct lnode*)* *new_node *whose first element is* x *and that is followed by the list of elements representing the original set; finally, it increases the size of the set by 1. If the insertion operation* add *succeeds, the call returns* 1 *once the new element has been added to the list; otherwise, it returns* 0 *(standing for unsuccessful insertion).*

isfull *returns* 1 *if the size of the set argument* s *is greater than or equal to its capacity; otherwise, it returns* 0*.* isnull *returns* 1 *if the address of the set argument is* NULL*; it returns* 0 *otherwise. Finally, the execution of* contains(s,x) *returns* 1 *if the argument element* x *belongs to* s *and returns* 0 *otherwise.*

Technically, the inferred specification for a given function consists of a set of implication formulas of the form $t_1 \Rightarrow t_2$ where t_1 and t_2 are conjunctions of equations of the form $l = r$. Each left-hand side l can be either

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

```
    isnull(s) = 1   ⇒   ret = 0 ∧ isnull(s’) = 1
    isfull(s) = 1   ⇒   ret = 0 ∧
                        contains(s,x) = contains(s’,x)
contains(s,x) = 1   ⇒   ret = 0 ∧ contains(s’,x)= 1

isnull(s) = 0 ∧     ⇒   ret = 1 ∧
isfull(s) = 0 ∧         isnull(s’) = 0 ∧
contains(s,x) = 0       contains(s’,x) = 1
```

Figure 2. Inferred specification for the add function.

Informally, the statements at the left-hand and right-hand sides of the symbol ⇒ are respectively satisfied before and after the execution of the function call. We adopt the standard primed notation for representing variable values before and after execution. For instance, given a variable s that stands for the value of the parameter s before the function is executed, the primed version s' stands for the value after the execution.

EXAMPLE 3. *Consider again the program of Example 2. The specification for the (modifier) function* add(s,x) *(that inserts an element* x *in the set* s*) is shown in Figure 2.*

The specification consists of four implications stating the conditions that are satisfied before and after the execution of the considered function. For instance, the first formula can be read as follows: if the result of running isnull(s) *is equal to* 1 *before executing* add(s,x)*, then the value returned by the call* add(s,x) *is* 0*, and, after its execution, the outcome of* isnull(s’) *is also* 1*.*

Even though the observers isnull *and* isfull *behave as boolean functions (predicates) in this example, we prefer not to write them in sugared relational form (i.e.,* isfull(s) *instead of* isfull(s)=1 *) since a specific datatype for Boolean numbers does not exist in* C*. Hence, even when we can detect that the observer function only returns two scalar values, say* 0 *and* 1 *as in the example, we cannot give it the semantics of a logical predicate.*

Note that any implication formula in the inferred 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 assumptions about the function purity or side-effects.

Our technique for inferring specifications relies on the symbolic execution engine of the MATCHING LOGIC verifier MATCHC. MATCHC works in a forward manner by symbolically executing an $\mathbb{ML}$ pattern that is provided as the program precondition, and non-deterministically obtaining a set of final patterns that are then used to discharge the postcondition. This is an instance of a general strategy to calculate the strongest postcondition of a predicate transformation semantics as explained in [14]. However, MATCHC is incomplete for the purpose of general symbolic execution in the sense that its symbolic machinery does not support incremental assumptions regarding the initial structure of the program memory; it can only assume the structure that is implicitly imposed by the initial pattern. For the inference purposes of this paper, we cannot assume any *ex-ante* condition for the initial program state; on the contrary, we need to incrementally collect all the assumptions that allow each symbolic execution path to be successfully executed. In the following section, we explain how we extended MATCHC to support collecting assumptions on-the-fly within the symbolic configurations as needed.

```
#include <stdlib.h>

struct lnode {
 int value;
 struct lnode *next;
};

struct set {
 int capacity;
 int size;
 struct lnode *elems;
};

struct set* new(int capacity)
{
 struct set *new_set;

 new_set =
  (struct set*) malloc(sizeof(struct set));
 if (new_set == NULL)
  return NULL; /* no memory left */

 new_set->capacity = capacity;
 new_set->size = 0;
 new_set->elems = NULL;

 return new_set;
}

int add(struct set *s,int x)
{
 struct lnode *new_node;
 struct lnode *end_node;
 struct lnode *n;

 if (s == NULL)
  return 0; /* NULL set */

 if (s->size >= s->capacity)
  return 0; /* no space left */

 n = end_node = s->elems;
 while (n != NULL) {
  if (n->value == x)
   return 0; /* element already added */
  end_node = n;
  n = n->next;
 }

 /* Initialize new node */
 new_node =
  (struct lnode*) malloc(sizeof(struct lnode));
 if (new_node == NULL)
  return 0; /* no memory left */
 new_node->value = x;
 new_node->next = s->elems;

 /* Link new node */
 s->elems = new_node;

 /* Update set info */
 s->size += 1;

 return 1; /* element added */
}

int isfull(struct set *s)
{
 if (s == NULL)
  return 0; /* NULL set provided */
 if (s->size >= s->capacity)
  return 1; /* is full */
 return 0; /* is not full */
}

int isnull(struct set *s)
{
 if (s == NULL)
  return 1;
 return 0;
}

int contains(struct set *s,int x)
{
 struct lnode *n;

 if (s == NULL)
  return 0; /* NULL set */

 n = s->elems;
 while (n != NULL) {
  if (n->value == x)
   return 1; /* element found */
  n = n->next;
 }

 return 0; /* element NOT found */
}
```

Figure 1. KERNELC implementation of a set with linked lists.

4. Extending the $\mathbb{ML}$ Symbolic Machine

Symbolic execution typically proceeds like standard execution except that, when a function or routine is called, symbolic values are assigned to its actual parameters and computed values become symbolic expressions that record the operations applied to them. When symbolic execution reaches a conditional control flow statement, every possible execution path from this statement 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 constraints* (one per possible branch), which are empty at the beginning of the execution. A path constraint 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 4. *Consider again the* `add` *function of Example 2. Assume that the input values for the actual parameters* `s` *and* `x` *are the symbolic values* s *and* x*, 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 statement. The path constraint of the first branch is updated with the constraint* $\mathsf{s} = \mathtt{NULL}$*, whereas* $\mathsf{s} \neq \mathtt{NULL}$ *is added to the path constraint of 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 constraint. When the path constraint is satisfiable, the *successful* path ends in a final (symbolic) configuration that typically stores a (symbolic) computed result.

For the symbolic execution of C programs, we must pay attention to pointer dereference and initialization. In C, a structured datatype (`struct`) is an aggregate type that describes 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, all the pointer arithmetic is done by means of symbolic address expressions that may appear in (the domain of) heap cells of MatchC patterns.

Example 5. *(Example 4 continued) Consider the second* `if` *statement of the* `add` *function given in Example 2. The evaluation of the guard of the conditional statement requires accessing both* `p->size` *and* `p->capacity`*. Since* `capacity` *is the first field in the* `struct set` *definition, its location coincides with the (base) address* `p`*. However, in order to access* `p->size`*, its address must be computed by adding an offset*[3] *of* `1` *to the (base) address* `p` *(i.e., if we assume that the symbolic address held by the variable* `p` *is* p*, then the computed address for the field* `size` *is* $\mathsf{p}+1$*.)*

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 the discovery setting of our approach, we have no a priori information regarding the memory (in particular, 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 the case in which it is not. In the second case, the symbolic execution gets stuck, thus identifying *undefined* behavior as in [10]. For the case in which the memory positions are actually initialized and execution should proceed, a strategy to reconstruct the original object in memory is needed. We adapt to our setting the lazy initialization of fields of [17]: when a symbolic address (or address expression) is accessed for the first time, SE initializes the memory object that is located at that 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. In contrast, in the case of failure, an `undefined` computation is pushed onto the k cell, which stops the execution.

Example 6. *(Example 5 continued) Before executing the second* `if` *statement, assume that the* heap *cell is empty, which means that nothing is known about the structure of the* heap *cell. After symbolically executing the guard of the* `if` *statement (which refers to the* `capacity` *field and* `size` *field of the structured type in* `s`*), the* heap *cell has the form:*

$$\langle \mathsf{s} \mapsto \mathsf{s.capacity}, \mathsf{s}+1 \mapsto \mathsf{s.size} \rangle_{\mathsf{heap}}$$

In other words, new symbolic bindings regarding the actual parameters are added, which represent the assumptions we made over the corresponding data structures.

In the following, we augment MatchC symbolic configurations (MatchC patterns) with new cells and naturally extend its symbolic execution machinery to work with the augmented patterns.

4.1 The MatchC Extension

The heap cell of MatchC patterns cannot be used to keep track of the assumptions made by the lazy initialization described above since the subsequent assignment statements that may occur during the symbolic execution typically overwrite the heap cell values. Therefore, we have extended MatchC patterns by introducing two additional cells: the iheap and ik cells. The iheap cell monotonically stores all the (structural) assumptions that are dynamically made for the *initial* heap. In this way, when symbolic execution finishes, the iheap cell, together with the ϕ cell, does contain the path constraint for the symbolic parameters that point to the dynamic data structures that were accessed along the branch. The ik cell stores the contents of the k cell when the symbolic execution starts. In our case, this cell always contains a symbolic (initial) function call.

The following example illustrates how the iheap cell is used. It also illustrates how the KernelC rules have been conservatively augmented to manipulate extended configurations, thanks to the modularity and underlying type structure of the $\mathbb{K}$ framework.

Example 7. *The rule in Figure 3 states that, whenever the symbolic interpreter accesses an uninitialized piece of memory (condition* $I \notin \mathrm{keys}(\mathit{Heap})$[4]*), a new symbolic variable* $\mathsf{NewFreeVar}$ *is introduced for that position in the* heap *cell, and also in the* iheap *cell, thus making that assumption persistent independently of the effect of subsequent assignments on the heap.*

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 inherent in path enumeration by exploring loops and recursion up to a specified number of unfoldings. This ensures that SE ends for all explored paths, thus representing a subset of the program behavior [9]. Obviously, not all the potential execution paths are feasible. We use the automatic theorem prover CVC3 [4] to check the satisfiability of *path constraints*, to simplify path conditions and to eliminate unfeasible symbolic computations whenever the corresponding path constraint is unsatisfiable.

In order to facilitate the specification inference, in the following section we define two types of patterns, called *observation* patterns (the *call-pattern* and the *return-pattern*), that we extract from the symbolic final configurations at the end of the symbolic execution paths.

4.2 The Pattern Extraction

We define a *call-pattern* as a pattern whose k cell consists of just a function call with (possibly symbolic) arguments. A *return-pattern* is a pattern that only has either a `return` instruction with the corresponding value or an `undefined` computation at the top of its k cell. The *call-patterns* and *return-patterns* (called observation patterns) respectively represent the *observable* state of a program before and after a specific function call is executed.

We note that all the information needed to extract the observation patterns is accumulated in the final MatchC symbolic configurations. In order to reuse the MatchC verification machinery, we formalize the two extracted patterns in terms of traditional $\mathbb{ML}$ patterns in the following way:

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

[3] We assume that the memory is indexed by words and that a value of type `int` has the size of a word.

[4] In C, the function $\mathrm{keys}(M)$ returns the domain of the mapping M. Thus, function call $\mathrm{keys}(\mathit{Heap})$ represents the set of initialized memory positions in the heap.

$$\left\langle \frac{\text{load}(T,I)}{\text{tv}(T,\mathsf{NewFreeVar})} \cdots \right\rangle_{\mathsf{k}} \left\langle \mathit{Heap}\ \frac{\phantom{I \mapsto \mathsf{NewFreeVar}}}{I \mapsto \mathsf{NewFreeVar}} \right\rangle_{\mathsf{heap}} \left\langle \cdots \frac{\phantom{I \mapsto \mathsf{NewFreeVar}}}{I \mapsto \mathsf{NewFreeVar}} \cdots \right\rangle_{\mathsf{iheap}} \quad \text{if } I \notin \text{ keys}(\mathit{Heap})$$

Figure 3. Rule for the symbolic execution for accessing a value in the memory.

- The call-pattern is defined by filling the heap cell with the content of the iheap cell, and the k cell with the content of the ik cell, and then discarding the iheap and ik cells.
- The return-pattern is obtained by simply deleting the iheap and ik cells.

In the following, we call *initial extended pattern* or *initial symbolic configuration* to the pattern that starts the symbolic execution of a function.

EXAMPLE 8. *To symbolically execute the* `int add(struct set *s,int x)` *function by using the extended* MATCHC *verifier, we start from the initial symbolic configuration*[5]

$$\overline{p} = \langle \cdots \langle \mathtt{add}(\mathsf{s},\mathsf{x})\rangle_{\mathsf{k}} \langle \mathtt{add}(\mathsf{s},\mathsf{x})\rangle_{\mathsf{ik}} \langle\rangle_{\mathsf{heap}} \langle\rangle_{\mathsf{iheap}} \langle\rangle_{\phi} \cdots \rangle_{\mathsf{cfg}}\,,$$

and from this extended pattern, at the end we obtain a set of final extended patterns $\overline{p_1} \ldots \overline{p_n}$*, where each* $\overline{p_i}$ *has the form*

$$\overline{p_i} =$$
$$\langle \cdots \langle \mathtt{return}\ \mathsf{Value}_i \rangle_{\mathsf{k}} \langle \mathtt{add}(\mathsf{s},\mathsf{x})\rangle_{\mathsf{ik}} \langle \mathsf{Heap}_i \rangle_{\mathsf{heap}} \langle \mathsf{IHeap}_i \rangle_{\mathsf{iheap}} \langle \Phi_i \rangle_{\phi} \cdots \rangle_{\mathsf{cfg}}$$

The call patterns and return patterns are extracted from $\overline{p_1} \ldots \overline{p_n}$*:*

$$\mathit{call\_pattern}(\overline{p_i}) = \langle \cdots \langle \mathtt{add}(\mathsf{s},\mathsf{x})\rangle_{\mathsf{k}} \langle \mathsf{IHeap}_i \rangle_{\mathsf{heap}} \langle \Phi_i \rangle_{\phi} \cdots \rangle_{\mathsf{cfg}}\,,$$
$$\mathit{return\_pattern}(\overline{p_i}) =$$
$$\langle \cdots \langle \mathtt{return}\ \mathsf{Value}_i \rangle_{\mathsf{k}} \langle \mathsf{Heap}_i \rangle_{\mathsf{heap}} \langle \Phi_i \rangle_{\phi} \cdots \rangle_{\mathsf{cfg}}.$$

Formally, the extracted observation patterns are also $\mathbb{ML}$ patterns and satisfy $\mathit{call\_pattern}(\overline{p_i}) \Downarrow \mathit{return\_pattern}(\overline{p_i})$. This is because, by construction, $\mathit{call\_pattern}(\overline{p_i})$ records all the assumptions needed to ensure that $\mathit{return\_pattern}(\overline{p_i})$ holds at the end of the symbolic execution branch following the $\mathbb{ML}$ proof system implemented in MATCHC. This allows us to ascertain the conditions for the completeness of our inference technique:

If the disjunction of the extracted call patterns is logically equivalent to the $\mathbb{ML}$ pattern that is obtained by removing the iheap and ik cells from the initial symbolic configuration (the extended pattern $\overline{p}$), then the set $\{\mathit{call\_pattern}(\overline{p_i}) \Downarrow \mathit{return\_pattern}(\overline{p_i})\}$, $i \in \{1 \ldots n\}$ of correctness pairs fully describes the input/output behavior of the considered program function.

In the following section, we formulate an algorithm that symbolically executes the program and automatically extracts and combines the call and return patterns in order to infer the pursued logical specifications.

5. Inference process

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

We denote with the symbol $\cdot$ the *universal* $\mathbb{ML}$ pattern that represents every possible program state, i.e., it imposes no constraint to the state. Given a function $f \in \mathcal{F}$, we represent the call to f with a list of arguments *args* as $f(\mathit{args})$. Then, $f(\mathit{args})[p]$ is the extended pattern built by first adding both the ik and iheap cells to the pattern p, next inserting the call $f(\mathit{args})$ into the ik and k cells, and then copying to the iheap cell the contents of the heap cell of p. The intuition is that $f(\mathit{args})[p]$ propagates the information in p to the execution of $f(\mathit{args})$. $f(\mathit{args})[\cdot]$ stands for the extended pattern that represents the execution of f with arguments *args* under a state without constraints, i.e., with empty information brought in by the universal pattern. Given an initial extended pattern $\overline{p}$, we denote as $\textsc{Se}(\overline{p})$ the set of final extended patterns $\{\overline{p_i}\}_{i>0}$ resulting from the symbolic execution of $\overline{p}$ in our MATCHC system, i.e., the leaves of the symbolic execution tree for $\overline{p}$. Each $\overline{p_i}$ has associated a correctness pair $\mathit{call\_pattern}(\overline{p_i}) \Downarrow \mathit{return\_pattern}(\overline{p_i})$. Given a return pattern q, $q|_{ret}$ is the projection of q to its `return` value or `undefined` computation, which are in the k cell.

Our specification inference methodology is formalized in Algorithm 1. First, the *modifier* method of interest is symbolically executed with fresh symbolic variables $\mathsf{a}_1, \ldots, \mathsf{a}_n$ as arguments. As a result, the set of final extended patterns S is computed. Then, by extracting and processing the call and return patterns of each $\overline{p_i} \in S$, a set of axioms is obtained that defines the behavior of the program. This is done by means of the function $\mathit{explains}(p, as)$ given in Algorithm 2. The computed axioms are implications of the form $l_i \Rightarrow r_i$. The function *simplify* implements a post-processing which consists on (1) disjoin the preconditions l_i that have the same postcondition r_i and simplify the resulting precondition, and (2) conjoin the postconditions r_i that share the same precondition and simplify the resulting postcondition.

Algorithm 1 Computing specifications.

```
Require: $m \in \mathcal{M}$ of arity $n$;
$S = \textsc{Se}(m(\mathsf{a}_1, \ldots, \mathsf{a}_n)[\cdot])$
$\mathit{axiomSet} := \emptyset$;
for all $\overline{p_i} \in S$ do
    $\mathit{eqs}_{pre} := \mathit{explains}(\mathit{call\_pattern}(\overline{p_i}), [\mathsf{a}_1, \ldots, \mathsf{a}_n])$;
    $\mathit{eqs}_{post} := \mathit{explains}(\mathit{return\_pattern}(\overline{p_i}), [\mathsf{a}_1, \ldots, \mathsf{a}_n])$;
    $\mathit{eq}_{ret} := \mathit{ret} = \mathit{return\_pattern}(\overline{p_i})|_{ret}$;
    $\mathit{axiomSet} := \mathit{axiomSet} \cup \{\mathit{eqs}_{pre} \Rightarrow (\mathit{eqs}_{post} \cup \mathit{eq}_{ret})\}$;
end for
spec := $\mathit{simplify}(\mathit{axiomSet})$
return spec
```

Let us show an example of the application of the algorithm.

EXAMPLE 9. *Assume that we want to infer a specification for the* `add` modifier *function of Example 2. Following the algorithm, we first compute* $\textsc{Se}(\mathtt{add}(\mathsf{s},\mathsf{x})[\cdot])$*, with* s *and* x *(free) symbolic variables. Since there are not initial assumptions for the initial symbolic configuration, the execution covers all possible initial concrete configurations. The symbolic execution computes five final extended patterns*[6]*. The following extended pattern* e *represents the path that ends in the body of the second* `if` *statement:*

$$\left\langle \begin{array}{c} \langle \cdots\ \mathit{ret} \mapsto 0, \mathtt{s} \mapsto \mathsf{s}, \mathtt{x} \mapsto \mathsf{x} \cdots \rangle_{\mathsf{env}} \\ \langle \mathsf{s} \mapsto \mathsf{s.capacity}, \mathsf{s}+1 \mapsto \mathsf{s.size} \rangle_{\mathsf{heap}} \\ \left\langle \begin{array}{c} \mathsf{s} \mapsto \mathsf{s.capacity}, \\ \mathsf{s}+1 \mapsto \mathsf{s.size} \end{array} \right\rangle_{\mathsf{iheap}} \\ \left\langle \begin{array}{c} \mathsf{s} \neq 0 \wedge \\ \mathsf{s.capacity} \leq \mathsf{s.size} \end{array} \right\rangle_{\phi} \end{array} \right\rangle_{\mathsf{cfg}}$$

[5] We only write those cells that we need to consider for the inference.

[6] For simplicity, we set the number of loop unrollings to one.

The execution of this path returns the value 0*; the fields* `s->size` *and* `s->capacity` *are accessed after checking that* s *is not* `NULL` *(i.e.,* 0*). In the extended pattern e, the return value* 0 *is represented by the binding* $ret \mapsto 0$ *in the* env *cell. The failed check of* `s == NULL` *adds the constraint* $\mathsf{s} \neq 0$ *to the* ϕ *cell. Given our assumption that any access to a field through a non-*`NULL` *pointer does succeed, the symbolic fields* s.capacity *and* s.size *are generated in the* iheap *cell. The successful check of* `s->size >= s->capacity` *adds an analogous constraint to the* ϕ *cell. Note that, since during the execution of this path the* heap *is not modified, the* heap *and* iheap *cells are identical, i.e., the initial and the final* heaps *are the same.*

Next, for each final extended pattern the algorithm explains its call- *and* return-patterns *by using the function* $explains(p, as)$*, which delivers suitable sets of equations. For the extended pattern e, the equations* $\texttt{isnull(s)} = 0$ *and* $\texttt{isfull(s)} = 1$ *are generated for both the* call- *and* return-patterns*. Additionally, the equation* `ret` $=$ `0` *is generated with the return value of e. Finally, by combining these equations we generate the following axiom:*

$$\texttt{isnull(s)} = 0 \land \texttt{isfull(s)} = 1 \Rightarrow$$
$$\texttt{ret} = 0 \land \texttt{isnull(s)} = 0 \land \texttt{isfull(s)} = 1$$

which is the computed explanation for e.

Let us now describe Algorithm 2 that defines the function $explains(p, as)$. Given an $\mathbb{ML}$ pattern p and a list of symbolic variables as, this function computes a set of equations as the description of p. These equations are composed of calls to observer functions and *built-in* functions that are bound to the (symbolic) values that are returned by the calls. In the algorithm, $args \sqsubseteq as$ states that the list of elements $args$ is a permutation of some (or all) elements in as.

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

```
Require: $p$ : the pattern to be explained
Require: $as$ : a list of symbolic variables
1. $\mathcal{C}$: the universe of observer calls;
2. $eqSet := \emptyset$;
3. for all $o(args) \in \mathcal{C}$ and $args \sqsubseteq as$ do
4.     $S = \textsc{Se}(o(args)[p])$
5.     if $\nexists\ \overline{p_1}, \overline{p_2} \in S$ s.t. $return\_pattern(\overline{p_1})|_{ret} \neq return\_pattern(\overline{p_2})|_{ret}$ then
6.         $eqSet := eqSet \cup (t = return\_pattern(\overline{p_1})|_{ret})$
7.     end if
8. end for
9. return $eqSet$
```

Roughly speaking, given a pattern p, $explains(p, as)$ first generates the universe of observer function calls $\mathcal{C}$, which consists of all the function calls $o(args)$ that satisfy that:

- o belongs to $\mathcal{O}$ or to the set of (predefined) built-in functions,
- $args$ in the call $o(args)$ is a suitable selection of variables from the symbolic variable list as that is received as argument, respecting the type and arity of o.

Then, for each call $o(args) \in \mathcal{C}$, it checks whether all the final symbolic configurations (leaves) resulting from the execution of $o(args)$ on a state that satisfies the constraints in p have the same return value. For the calls that satisfy this condition, an equation is generated (line 6 in Algorithm 2). The intuition of this step is that, if we symbolically execute the observer at a given initial state and for all its execution branches we get the same value, then the observer together with the return value (partially) characterize the considered state. The last step of the algorithm returns the set of all the generated equations.

Example 10 (Example 9 continued). *Let us show how we compute the explanation for the* return-pattern *of Example 9 given the symbolic variables considered in the example.*

Given the observer functions `isfull`, `isnull` *and* `contains`, *and the symbolic variables* s *and* x*, the universe of observer calls is* `isfull(s)`, `isnull(s)` *and* `contains(s,x)`*. Let us show in detail the case for the observer* `isnull(s)`.

When we symbolically execute `isnull(s)` *over the considered* return-pattern*, we only obtain the extended pattern:*

$$\left\langle \begin{array}{c} \langle \cdots\ ret \mapsto 0\ \cdots \rangle_{\mathsf{env}} \\ \langle \mathsf{s} \mapsto \mathsf{s.capacity}, \mathsf{s}+1 \mapsto \mathsf{s.size} \rangle_{\mathsf{heap}} \\ \langle \mathsf{s} \mapsto \mathsf{s.capacity}, \mathsf{s}+1 \mapsto \mathsf{s.size} \rangle_{\mathsf{iheap}} \\ \langle \mathsf{s} \neq 0 \land \mathsf{s.capacity} \leq \mathsf{s.size} \rangle_{\phi} \end{array} \right\rangle_{\mathsf{cfg}}$$

Since there are no observer paths returning different values because there is only one path (the one for computed extended pattern) and its associated return value is 0*, the equation* $\texttt{isnull(s)} = 0$ *can be used as a (partial) explanation for the pattern under consideration. Then, this equation is added to the set of equations eqSet that will be returned by Algorithm 2.*

Due to bounded loop unrolling, we cannot ensure completeness of the inferred specifications as we do not cover all possible execution paths. Also due to generalization, we cannot ensure that all inferred axioms are sound. Nevertheless, when we consider loops that are characterized by an invariant that matches into a given set of invariant templates, then we could guarantee both the soundness and completeness of our technique.

5.1 Refining the inference process

There are some cases in which explanations cannot be achieved due to the lack of sufficiently precise observers. In order to mitigate this problem, we present a refinement process that allows more accurate specifications to be computed and that can be applied automatically. The idea is to split the pattern that could not be explained (because there were different computed results in the leaves of the symbolic execution of observers) into multiple refined patterns that the observer functions are then able to explain.

Algorithm 3 describes how to refine a pattern p by using the observer call c. First, we compute $\textsc{Se}(c[p])$, which obtains a set S of final extended patterns. Note that the *call-pattern* of each of these final patterns $\overline{p_i} \in S$ contains additional constraints that are imposed by the symbolic execution of the observer call. In other words, the set consisting of the new extracted call patterns forms a refinement of p.

Running the observer c under the new refined patterns delivers a single return value for each run, which brings the corresponding explanation using c. An interesting open question is to be able to determine when we have enough observers, that is, to determine which observers are missing.

Algorithm 3 Computing refined explanations for an observer: $refined\_explains(p, c)$

```
Require: $p$ : the pattern to be explained
Require: $c$ : the observer call that will explain $p$ by refinement
$expl := \emptyset$;
$S = \textsc{Se}(c[p])$
for all $\overline{p_i} \in S$ do
    $expl := expl \cup \{p \mapsto explains(call\_pattern(\overline{p_i}))\}$;
end for
return $expl$
```

A prototype implementation of the inference methodology described in this paper has been developed, called KindSpec. We evaluated the accuracy of the inferred specifications for our running example by comparing the automatically inferred specification with the original specification written by hand, and we find out that only in a few cases there was a significant loss of information. Table 1 summarizes the obtained results for a set of selected benchmark programs. For each program example, the first three columns show the number of modifiers, observers and lines of code of the corresponding program, respectively. Column # corresponds to the number of explored paths as determined by the imposed termination criteria. The last two columns indicate the number of generated axioms and the number of inferred axioms that are sound, respectively. This last measurement is also related to the imposed termination criteria: the greater the depth of unrolling, the highest number of correct axioms is obtained. With respect to the time cost of the inference, it ranges from 1s. to 10s. which is comparable to the performance of similar tools, e.g. [19], and quite promising.

Of course, our synthesis system KINDSPEC inherits the current limitations of the underlying MATCHC verifier. To effectively synthesize highly accurate specifications for larger programs that involve more complicated reasoning, more efficient verifiers are needed that are actually forthcoming.

6. Conclusions and related Work

We have proposed an algorithm for automatically synthesizing formal specifications for heap-manipulating programs that are written in KERNELC. Formal specifications improve software understanding and are useful for automated program testing and verification.

Specification extraction is itself not new. The automatic generation of likely specifications (either in the form of contracts, interfaces, summaries, assumptions, invariants, properties, component abstractions, process models, rules, graphs, automatas, etc.) from program code has received increasing attention. Specifications can be property oriented (i.e., described by pre-/post conditions or functional code); stateful (i.e., described by some form of state machines); or intensional (i.e., described by axioms). Here we only try to cover those lines of research that have influenced our work most.

Unlike our symbolic specification inference method, Daikon [11] and DIDUCE [15] detect program invariants by extensive test runs. Also, Henkel and Diwan [16] built a tool that dynamically discovers specifications for interfaces of Java classes by first generating, using the class signature, many test cases that consist of terms representing sequences of method invocations, and then generalizing the results of these tests to algebraic specifications. QUICKSPEC [6] is another inference tool that is based on testing and can be used to generate laws that a Haskell program satisfies. Whereas Daikon discovers invariants that hold at existing program points, QUICKSPEC discovers equations between arbitrary terms constructed using an API, similarly to [16]. Also, they use a similar overall approach that is based on testing: they generate terms and evaluate them, then dynamically identify terms that are equal, and finally generate equations, filtering away redundant ones. ABSSPEC [3] 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 [27]. A combination of symbolic execution with dynamic testing is used in Dysy [8]. Ghezzi *et al.* [12] infer specifications of container-like classes as finite state automatas combined with graph transformation rules. All these proposals observe that conditional equations would be useful, but neither tool generates them nor include associative or commutative operators, which are naturally supported in our approach thanks to the handling of MAUDE's (hence $\mathbb{K}$'s) equational attributes [7]. By supporting the modular combination of associative, commutative, idempotent and unity equational attributes for function symbols (which makes these combinations transparent to the developer), the $\mathbb{K}$ framework naturally conveys enough expressive power to reason about typed data structures such as lists (list concatenation is associative with unity element nil), multisets (insertion is associative-commutative with unity $\emptyset$), or sets (insertion is associative-commutative-idempotent with unity $\emptyset$). By using equational attributes to declare such properties, we can avoid nontermination problems and achieve much more efficient evaluation of terms containing such operators. We take advantage of these capabilities at three levels:

- for the definition of the extended language semantics, where the heap structures and pointer handling are represented as appropriate data structures and their associated operations,
- for the mechanization of the inference process: we efficiently handle (eventually huge) sets of axioms, paths and constraints
- most importantly, for computing the sugared version of the specification, where, by simply imposing an order relation on terms, we get a first simplification of axioms almost for free, and we infer specifications where the function symbols can be given equational attributes as well.

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 to *induce* the specification. Much of the work on specification mining is targeted at inferring API protocols dynamically. For instance, Whaley *et al.* [28] describe a system to extract component interfaces as finite state machines from execution traces. The work in [8] offers a thorough revision of data mining approaches for inferring different kinds of specifications, typically from traces or observed program runs (e.g., models, summaries, regular invocation patterns or state machines). An algorithm for interface generation of software components using learning techniques is presented in [13] and implemented in the JavaPathfinder model-checking framework.

Our approach differs from most of the above because we do not infer abstract properties by observations of (concrete) program runs. Our axiomatic representation of functions and of their effects is inspired by [19]. However, our approach does not rely on a model checker for symbolic execution, as opposed to Tillmann's approach. Also, we do not generate the output as parameterized unit tests or Spec# specifications; we have simpler and more accurate formulas that avoid reasoning with the global heap but rather separate the different pieces of the heap that are reachable from the function argument addresses. Moreover, we can refine the observers by means of Algorithm 3 so that we are able to get more accurate specifications, although more experiments have yet to be done to compare all these inference methods in larger code.

As a further advantage w.r.t. [19], in our framework, correctness of the inferred axioms can be checked automatically by using the very same MATCHC verifier. Also, our methodology can be easily applied to any language which is given a semantics in the $\mathbb{K}$ framework.

References

[1] The Generic Data Structures Library GDSL. URL `http://home.gna.org/gdsl/`.

[2] J. G. B. Liskov. *Abstraction and specification in program development.* MIT Press, 1986.

[3] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. 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)*, volume to appear. ACM Press, 2012.

[4] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer*

Table 1. Experimental results

Module	Modifiers	Obs.	LOC	#	Axioms	Sound
Set List	`add`	4	70	87	4	4
	`remove`	5	90	54	2	2
Double-Linked Lists	`append`	9	180	43	4	3
	`remove`	9	180	66	2	2
Double-ended Queues	`push_head`	4	100	106	3	3
	`push_tail`	4	100	142	3	3
	`pop_head`	4	100	16	2	2
	`pop_tail`	4	100	24	2	2
Stack	`push`	3	25	14	2	2
	`pop`	2	25	15	2	2

Aided Verification (CAV '07), volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[5] M. Christodorescu, S. Jha, and C. Kruegel. 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)*, pages 5–14. ACM, 2007.

[6] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *Proc, 4th Int'l Conf. on Tests and Proofs (TAP 2010)*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

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

[9] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[10] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012. doi: 10.1145/2103656.2103719.

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

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

[13] D. Giannakopoulou and C. S. Pasareanu. Interface Generation and Compositional Verification in JavaPathfinder. In *Proc. 12th In'l Conf. on Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2009.

[14] M. Gordon and H. Collavizza. Forward with hoare. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, pages 101–121. Springer London, 2010. ISBN 978-1-84882-911-4. doi: 10.1007/978-1-84882-912-1_5. URL `http://dx.doi.org/10.1007/978-1-84882-912-1_5`.

[15] S. Hangal and M. S. Lam. Tracking down Software Bugs using Automatic Anomaly Detection. In *Proc. 22rd International Conference on Software Engineering (ICSE 2002)*, pages 291–301. ACM, 2002.

[16] J. Henkel and A. Diwan. Discovering Algebraic Specifications from Java Classes. In *Proc. ECOOP*, pages 431–456, 2003.

[17] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003. ISBN 3-540-00898-5.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. ISSN 0001-0782.

[19] N.Tillmann, F. Chen, and W. Schulte. Discovering Likely Method Specifications. In *Proc. 8th Int'l Conf. on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 717–736. Springer, 2006.

[20] C. S. Pasareanu and W. Visser. A Survey of new Trends in Symbolic Execution for Software Testing and Analysis. *STTT*, 11(4):339–353, 2009.

[21] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74. IEEE Computer Society, 2002.

[22] G. Roşu and W. Schulte. Matching logic — extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.

[23] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, 2010.

[24] G. Rosu and T.-F. Serbanuta. An Overview of the $\mathbb{K}$ Semantic Framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[25] G. Rosu and A. Stefanescu. Matching Logic: a new Program Verification Approach. In R. N. Taylor, H.Gall, and N. Medvidovic, editors, *Proc. 33rd International Conference on Software Engineering (ICSE 2011)*, pages 868–871. ACM, 2011.

[26] G. Rosu, W. Schulte, and T. F. Serbanuta. Runtime verification of C memory safety. In S. Bensalem and D. A. Peled, editors, *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009. doi: doi:10.1007/978-3-642-04694-0_10.

[27] M. Taghdiri and D.Jackson. Inferring Specifications to Detect Errors in Code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.

[28] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. ISSTA 2002*, pages 218–228, 2002.