Automated Synthesis of Software Contracts with KindSpec*

María Alpuente[0000−0002−9268−1178] and Alicia Villanueva[0000−0003−1090−5009]

Valencian Research Institute for Artificial Intelligence, VRAIN
Universitat Politècnica de València
{alpuente,alvilga1}@upv.es

Abstract In this paper, we describe KindSpec, an automated tool that synthesizes software contracts from programs that are written in a significant fragment of C that supports pointer-based structures, heap manipulation, and recursion. By relying on a semantic definition of the C language in the $\mathbb{K}$ semantic framework, KindSpec leverages the symbolic execution capabilities of $\mathbb{K}$ to axiomatically explain any program function. This is done by using observer routines in the same program to characterize the program states before and after the function execution. The generated contracts are expressed in the form of logical axioms that specify the precise input/output behavior of the C routines, including both general axioms for default behavior and exceptional axioms for the specification error behavior. We summarize the main services provided by KindSpec, which also include a novel refinement facility that improves the quality and accuracy of the synthesized contracts. Finally, we provide an experimental evaluation that assesses its effectiveness.

Keywords: Contract inference · Symbolic execution · Abstract subsumption · Exceptions.

1 Introduction

Software contracts provide mathematical specification for the terms of the service that software components can provide. Contracts on software are essentially written by using program preconditions and postconditions, which are similar to Hoare formulas that formalize the mutual obligations and benefits of the software units or routines [34]. Contract checking can improve software reliability but requires contracts to always be guaranteed to be consistent with the program code, which places a heavy burden on programmers and hinders its applicability. Moreover, while exceptional (or error) behavior specification should be integral to the contract, error specification is highly prone to introduction of mistakes and oversight.

* This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215, grant RTI2018-094403-B-C32 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and by Generalitat Valenciana PROMETEO/2019/098.

This paper presents KindSpec, an automated contract synthesis tool that is based on abstract *symbolic execution* for a significant fragment of C called KernelC [18]. KernelC supports recursive function and data structure definition, pointers, and dynamic memory allocation and deallocation (`malloc` and `free` routines), but it lacks pointer arithmetic and the possibility to import external code. The contracts that we synthesize essentially consist of logical assertions that characterize the behavior of a program function in terms of what can be observed in the states before and after the function execution. The inferred axioms include default (general) rules and exceptions to these rules that specify exceptional (or error) behavior; e.g., undesirable use cases or execution side effects.

The overall quality of programs and specifications can be fairly improved by systematically dealing with errors and exceptions. While several mainstream languages such as C++ and Java provide built-in support for exception handling, the C ANSI/ISO standard does not foresee any high-level way to define, throw, and catch exceptions [1]. The usual way for handling errors in C is to define special error return values through program constants, with the caller's duty being to check the returned value and to take appropriate action [21]. A known disadvantage of this practice is that it obscures the program control flow and is highly prone to oversight. Since exception failures can account for up to 2/3 of system crashes and 50% of security vulnerabilities [22], the capability to infer exceptional axioms from program code can be very helpful in this regard.

KindSpec implements an extension of the contract-discovering technique developed in [2,3], which is based on symbolic execution, a well-known program analysis technique that runs programs by using *symbolic* input values rather than actual (concrete) values [5,30]. By abstractly representing inputs as symbols, symbolic execution can simultaneously explore multiple paths that a program could take, resorting to constraint solvers to construct actual instances that would exercise the path. Roughly speaking, in the discovery methodology of [2], given a function f of a program P, and a root-to-leaf path from the pre-state s to the post-state s' in the symbolic execution tree for f in P, an implicative axiom $(p \Rightarrow q)$ is synthesized that *explains* the symbolic path from s to s'. Essentially, the antecedent p (resp. consequent q) of the axiom consists of a sequence of equations of the form $o(x_1, \ldots x_m) = v_s$ (resp. $o(x_1, \ldots x_m) = v_{s'}$) where each v_s (resp. $v_{s'}$) is the result of applying the m-ary *observer* function o of P to s (resp. s'). For example, for the case of a classical function `push(x,t)` that piles up an element `x` at the top of a given bounded stack `t`, the inferred logical axiom describes the expected behavior that, provided `t` was not full, the new top element is `x` and the stack size is increased by one: `size(t)=n` $\wedge$ `isfull(t)=0` $\wedge$ `top(t)=?e` $\Rightarrow$ `size(t)=n+1` $\wedge$ `isfull(t)=?b` $\wedge$ `top(t)=x`, where `?e` and `?b` stand for symbolic values.

The symbolic infrastructure of KindSpec is built on top of the rewriting-based, programming language definitional framework $\mathbb{K}$, which facilitates the development of executable semantics of programming languages and related formal analysis techniques and tools, such as type inferencers or program verifiers

[38]. In [2], the recent symbolic execution capabilities of $\mathbb{K}$ –that are available from $\mathbb{K}$ 3.4 on– were enriched with two new features not provided by $\mathbb{K}$: 1) lazy initialization, to effectively handle symbolic memory objects; and 2) abstract subsumption, to ensure termination without imposing fixed depth bounds. Due to abstraction, some of the inferred axioms cannot be guaranteed to be correct and are kept apart as candidate (or overly general) axioms.

KindSpec builds upon a previous, preliminary prototype presented in [2] and improves it in several ways: 1) we have fairly improved the maturity and robustness of the tool, giving support to more precise abstract domains that allow us to deal more accurately with complex dynamic allocated data structures such as linked lists and doubly-linked lists (including circular/cyclic lists); 2) we improved the accuracy of the inferred contracts by extending the original refinement process implemented in KindSpec that gets rid of less general axioms with new functionality for supporting axiom trusting and falsification; 3) we have extended the coverage of the analysis with the capability to infer axioms that express exceptional behavior; this not only improves the quality of the specification but may also suggest suitable program fixes that prevent execution failures to occur due to the faults. The KindSpec tool is publicly available at http://safe-tools.dsic.upv.es/kindspec2_2.

Manuel's pioneering work on concurrent logic programming with assertions has been a source of inspiration for our research on semantics of concurrent languages and symbolic execution since we met in the 1990s. The aim of this work is to honor Manuel with this paper that contributes to further advancing the intertwining between these areas.

2 Inferring Software Contracts with KindSpec

The wide interest in formal specifications as helpers for a variety of analysis, validation, and verification tools has resulted in numerous approaches for (semi-)automatically computing different kinds of specifications that can take the form of contracts, snippets, summaries, process models, graphs, automata, properties, rules, interfaces, or component abstractions. In this work, we focus on input-output relations; given a precondition for the state, we infer which modifications in the state are implied, and we express the relations as logical implications that reuse the program functions themselves. In order to achieve this, the inference technique of KindSpec relies on a classification scheme for program functions where a function may be a *modifier* or an *observer*, or it can play both roles. As defined in [31], observers are operations that can be used to inspect the state of an object, while modifiers change it. Since the C language does not enforce data encapsulation, we cannot presume purity of any function. Hence, we do not assume the traditional premise that observer functions do not modify the program state and we consider as observer any function whose return type is different from `void`.

Symbolic execution of a function call can be represented as a tree-like structure where each branch corresponds to a set of possible execution paths. At any

time of the execution, KindSpec's symbolic execution engine maintains a state $s = (pc, stmt, \sigma, h, \phi)$, where pc (the program counter), $stmt$ (the next statement to evaluate), σ (the symbolic program store that associates program variables with symbolic expressions), and h (the symbolic heap used to store dynamically allocated objects) are akin to standard configurations used in operational semantics. As for the path constraint ϕ, it is a formula that expresses a set of assumptions that are generated whenever a branching condition on primitive fields is taken in the execution to reach $stmt$. Intuitively, when symbolic execution reaches a conditional control flow statement, the logical condition that enables each branch is conjuncted to the accumulated path constraint of each diverging path. When the executed path ends, the associated path constraint represents the condition that input values must satisfy in order for the execution to reach the current program point.

To provide for contract discovering, we enriched the symbolic states supported by the symbolic $\mathbb{K}$ framework with a new component ι (called the initial heap) that is aimed to keep track of the heap constraints that are generated during *lazy initialization*. Roughly speaking, when an instruction performs a first access to an uninitialized object reference field, the symbolic execution forks the current state with three different heap configurations, in which the field is respectively initialized with: (1) `null`, (2) a reference to a new object with all symbolic attributes, and (3) a previously introduced concrete object of the desired type.

In order to synthesize a contract for the function of interest f, a symbolic call to f is executed with a sequence $x_1, \ldots x_n$ of fresh variables (simply denoted by $\overline{x_n}$) as arguments and initial path constraint `true`, yielding as a result a set $\mathcal{F}$ of final states. Then, for each state F in $\mathcal{F}$, an instantiated initial state $I = (0, f(\overline{x_n}), \emptyset, \iota, \phi)$ which stands for the program state before executing f, is built by joining together the initial call $f(\overline{x_n})$ with the path constraint ϕ and the lazy initialization constraint ι that are both retrieved from the final state F. The symbolic execution path from I to F is then described by means of an axiom $(p \Rightarrow q)$, which is obtained by:

1. symbolically running every (*feasible*) m-ary program observer o on both states I and F, over any subsequence of m arguments taken from $\overline{x_n}$ and all its possible permutations. The *feasible* observers are those having a subset of f's arguments as parameters. Each observer execution contributes an equational explanation $o(\overline{x_m}) = v_I$ (resp. $o(\overline{x_m}) = v_F$) to the premise p (resp. the consequent q) of the synthesized axiom.
2. Adding to q a last equation $ret = v$, where v is the value returned by the function f at the final symbolic execution state F.

The expectation that observation functions exist or can easily be written is reasonable. Observer calls are independently executed on I (resp. F) so that they cannot contaminate each other. Those observer calls that are found out to modify the given state are disregarded since the observation could have corrupted the observed behavior. Also, we note that lazy initialization is never applied during the symbolic execution of observer functions since they would be exploring fresh states beyond the analyzed symbolic execution configurations. When it is not the

case that all of the symbolic execution branches for $o(\overline{x_m})$ return the same value, the observation is inconclusive and a symbolic equation $o(\overline{x_m}) = ?v$ is built, for fresh symbolic variable $?v$.

Specification of exceptional behavior. Error specification and handling has traditionally been a challenge to the theory of abstract data types [36], which is considered a major tool for writing hierarchical, modular, implementation-independent specifications [24]. The main reason for this is that initial algebra semantics considers errors just as ordinary data and then spends much effort to discriminate errors from correct data. Our error handling approach borrows some ideas from *order-sorted* semantics, which supports many different styles for dealing with errors [26]. Roughly speaking, at the semantic level we provide the semantic definition of the language with an explicit *error supersort (supertype)* S for each program type T, such that error handling is naturally achieved by overloading every program operator $f : T_1, T_2, \ldots T_n \to T$ in the corresponding error supertypes, i.e., $f : S_1, S_2, \ldots S_n \to S$. By this means, error return values belong to the supertype S and are valid results for the evaluation of operator f although they are not compatible with correct data return values of T. This is comparable in a sense to the handling of errors in the ACSL contract specification language [6], where special error return values are introduced in the C semantics.

In [25], Goguen suggests including all exceptional behaviors and error messages directly in the specifications by providing as much information as is helpful about what is wrong or exceptional. In order to identify exceptional state behavior and errors directly from the program code, we have enriched the symbolic execution of [2] so that exceptional behavior is integral to the inferred contract specification. First, we have identified the most common undesirable (or erroneous) behaviors that may occur while running a KernelC[1] program and provided each of them with an error code, as shown in Table 1. Then, we created a new predefined KernelC data type (universal supertype) consisting of the set of error codes, and we redefined the KernelC semantic rules such that error return values of the form $(errorCode, pc)$ are allowed for all types, where the program counter pc aims to identify the precise statement of the program code that caused the error. Finally, we provided overloaded definitions of the program functions as explained in Section 1. By this means, every circumstance where an exception is triggered is witnessed by the corresponding error return value, which is not only useful for debugging purposes but can also be used to ascertain suitable program patches that avoid the errors. KindSpec internally represents each error e (e.g., null dereference, division by zero, etc.) as the repair problem $(e, pc, \mathcal{V})$, with $\mathcal{V}$ being the set of affected variables, whose solutions would represent a particular program fix. By the time being, such a fix just consists of suggesting the insertion of safety checks on the variables of $\mathcal{V}$ at the right program points to avoid e. The general problem of automated program repair

[1] Some standard C syntactic errors such as `IRT` are not statically detected by $\mathbb{K}$, thus they show up at (symbolic) execution time.

Table 1. Most common exceptions added to the KindSpec definition of KernelC.

Error code	Exception	Description
`NPE`	*Null Pointer Error*	Null dereferencing
`DBZ`	*Division By Zero*	Division of any number by 0
`VVA`	*Void Value Access*	Access to a non-pointer value of type `void`
`NMS`	*Non-valid Malloc Size*	Calling `malloc` with a negative or zero object size
`NOD`	*Null Object Destruction*	Calling `free` over a `null` reference
`UMA`	*Undefined Memory Access*	Access to an undefined memory segment (e.g., immediately after a pointer declaration)
`OOS`	*Out Of Scope*	Access to a variable that is out of scope
`IRT`	*Incorrect Return Type*	Type of return value does not match the function profile
`IAT`	*Incompatible Assign Types*	Mismatch between type of variable and assigned value
`NEF`	*Non-Existing Function*	The called function is not defined or declared
`UAC`	*Unsuitable Call Arguments*	Function call does not match the function profile

for heap-manipulating programs is another major endeavour that has received increasing attention (see, e.g., [20,41]) and we left for future work.

In [8], exception handling and error recovery cases are specified by means of "declarations" that separate the correct values and the error values into two different subsets of their carrier sets. The semantic approach that we adopt is more akin to [36], which differentiates compile-time *sorts* from run-time *types*, where compile-time sorts are used to agglutinate both error and correct values (with the error values being interpreted as meta-level data) while run-time types are restricted to correct values. Similarly, our approach allows errors to be dealt with at *inference time* (at symbolic execution level) even if the generated logical axioms are unsorted.

The inferred contract. Given the set $IA = \{p_1 \Rightarrow q_1, \ldots, p_n \Rightarrow q_n\}$ of inferred axioms and the subset $EA \subseteq IA$ of exceptional axioms, let us denote as DA the set of default axioms, $DA = (IA - EA)$. The resulting contract is given by $<Pre,\ Post,\ Loc>$, where: 1) Pre is the function precondition given by $(\bigvee p \mid (p \Rightarrow q) \in DA)$ that represents the admissible program input data; 2) $Post$ is the function postcondition given by IA; and 3) Loc is a set of references to memory *locations* (function parameters and data-structure pointers and fields) whose value might be affected by the function execution. The Loc component of the contract is comparable to the `assignable` clause in standard contract specification languages such as ACSL or JML, while the Pre and $Post$ components are similar to the ACSL pre- and post-conditions in contracts with *named behaviors* [6].

Since we are using abstraction, some inferred axioms for function f cannot be guaranteed to be correct and are kept apart as *candidate* axioms. A refinement

post-processing is implemented in KindSpec that 1) allows the user *trust* that a candidate axiom is, in fact, true, and then adds the axiom to the final contract; 2) provides support for semi-automated (testing-based) candidate axiom falsification, removing those candidate axioms for which an instance is refuted; and 3) filters out some redundant elements from the surviving axioms by detecting axiom subsumption. In order to deal with arithmetic, we adopt a constrained representation $p \wedge c \Rightarrow p' \wedge c'$ of axioms, where p and p' are conjunctions of equations of the form $o(\overline{x_m}) = y$, and c and c' are integer arithmetic constraints (e.g., $y = z + 1 \wedge z \geq 1$). This constrained representation is easily achieved by flattening each equation $o(\overline{x_m}) = t$, with t being a nonvariable term, to the constrained form $o(\overline{x_m}) = y \wedge y = t$. Then we check axioms for constraint subsumption [35]: a constraint c_1 is said to subsume[2] c_2 if c_2 implies c_1 (e.g., the constraint $y = z + 1 \wedge z \geq 1$ subsumes $y = 2$). The notion of constraint subsumption is naturally extended to constrained axioms in the obvious way: we say that a constrained axiom $p_1 \wedge c_1 \Rightarrow p'_1 \wedge c'_1$ subsumes another constrained axiom $p_2 \wedge c_2 \Rightarrow p'_2 \ \wedge \ c'_2$, if $p_1 \cup p'_2$ is a subset of $p'_1 \cup p_2$ modulo renaming γ, and the constraint $(c_1 \wedge c'_2)\gamma$ subsumes $(c_2 \wedge c'_1)\gamma$ (by abuse we consider any conjunction p of equations $e_1 \wedge \ldots \wedge e_n$ as the equation set $\{e_1, \ldots, e_n\}$). Although checking for subsumption is not generally an easy task, we are able to make most common cases run fast by applying standard heuristics that can detect failures early [39]. Also, in some cases KindSpec further simplifies the final set of axioms by applying some simple, commonly occurring constraint generalization patterns to compute more general axioms under constraint subsumption.

3 KindSpec at a glimpse

In this section, we outline the main features of the KindSpec tool. A starting guide that contains a complete description of all the settings and detailed sessions can be found at the tool homepage.

The granularity of the specification units (contracts) that can be generated by KindSpec is at the level of one function, as in many state-of-the-art contract specification approaches.

Given a program file and selected program function, the output of KindSpec is a structured Java object that represents the inferred contract. The contract can be either exported into a human-readable text file through the Save option of the File menu) or saved in serialized format (through the Export contract option) that can be then processed automatically by other techniques or tools.

Let us describe the graphical user interface (GUI) of the tool, as shown in Figure 1. In the upper part of the right-hand side section of the input panel, a KernelC program can be uploaded from the computer or selected from a drop-down list of built-in program examples. In the lower part of this section, all of the functions from the considered program are automatically loaded so that the user can select the function for which the contract is to be inferred. Two extra

[2] From a model-theoretic viewpoint, this is to say that the solution set of c_1 contains the solution set of c_2.

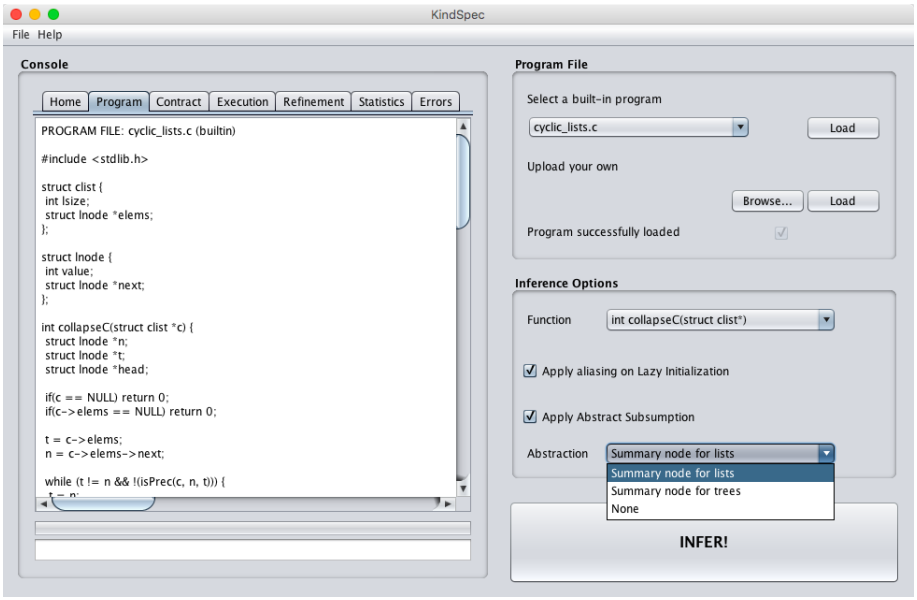

Figure 1. Graphical interface of KindSpec.

inference options are provided for enabling/disabling aliasing and/or abstract subsumption (explained in the following subsections). Once everything is set, the contract inference process is triggered by pressing the INFER! button. All of the process details are available through several tabs at the Console that is shown on the left-hand side section of the input panel: 1) the input Program; 2) the inferred Contract; 3) Execution intermediate outcomes (e.g., the symbolic execution tree for the considered function and the raw axiom set that is generated prior to any subsequent refinement); 4) the candidate axioms that can be selected for the Refinement process that admits *trusting* (i.e., explicitly marking as correct some candidate axioms), gets rid of many redundant and spurious axioms, and achieves in some cases falsification (i.e., disproving a candidate axiom); 5) some Statistics of interest, including the elapsed symbolic execution time, inference time, and number of inferred axioms; and 6) any eventual Errors that might have arisen during KindSpec execution. Note that the Refinement tab does not only show information, but also offers interactive entry points (through buttons) to the axiom refinement features of KindSpec.

3.1 A running example

Figure 2 shows a fragment of a KernelC program that implements an abstract data type for representing single-linked cyclic lists. The program code is composed of five functions: 1) the function `collapseC(c)` implicitly assumes `c` is a singly-linked cyclic list (i.e., either a circular list or a lasso) and deletes all of

the elements in the cycle except the first one, which becomes a self-cycle; 2) the function `isN(c)` returns 1 if the pointer `c` references to `NULL` memory; 3) `isE(c)` returns 1 if `c` points to an empty list (i.e., `c->elems` is `NULL`); 4)`lenC(c)` counts up the number of elements in the circular segment of `c`; and 5) the auxiliary function `isPrec(c, n, t)` that is used to identify the beginning of a cycle and proceeds by checking whether the node referenced by the pointer `n` precedes the node pointed by `t` in `c`.

```
1  #include <stdlib.h>
2
3  struct lnode {
4   int value;
5   struct lnode *next;
6  };
7  struct clist {
8   int lsize;
9   struct lnode *elems;
10 };
11
12 int collapseC(struct clist *c) {
13  struct lnode *n;
14  struct lnode *t;
15  struct lnode *head;
16
17  if(c == NULL) return 0;
18  if(c->elems == NULL) return 0;
19
20  t = c->elems;
21  n = c->elems->next;
22  while (t!=n && !(isPrec(c,n,t))) {
23   t = n;
24   n=n->next; }
25
26  head = n;
27  n = head->next;
28  while(n != head) {
29   t = n;
30   n = n->next;
31   free(t);
32   c->lsize--; }
33
34  head->next = head;
35  return 1; }
36
37 int isN(struct clist *c) {
38  return c == NULL; }
39
40 int isE(struct clist *c) {
41  return c->elems == NULL; }
42
43 int lenC(struct clist *c) {
44  struct lnode *n;
45  struct lnode *t;
46  struct lnode *head;
47  int counter;
48
49  if(c == NULL) return 0;
50  if(c->elems == NULL) return 0;
51
52  t = c->elems;
53  n = c->elems->next;
54  while (t!=n && !(isPrec(c,n,t))) {
55   t = n;
56   n=n->next; }
57
58  head = n;
59  n = head->next;
60  counter = 1;
61  while(n != head) {
62   counter++;
63   n = n->next; }
64
65  return counter;
66 }
67
68 int isPrec(struct clist *c, struct lnode *n,
       struct lnode *t) {
69 [...] }
```

Figure 2. KernelC implementation of a cyclic list data type.

Since C does not ensure purity of functions, any program function can be chosen for contract generation. We have selected `collapseC` for the running example.

Setting the inference options. Let us describe the inference options that are available in the right-hand side section of the panel.

Aliasing on Lazy Initialization. As we previously discussed in Section 2, when a symbolic address is accessed for the first time, three lazy initialization cases are considered: 1) null; 2) a reference to a new object of its respective type, and 3) a reference to an already existing object in the heap, which allows cyclic

data structures to be dealt with. This avoids requiring any a priori bound size for symbolic input structures. In the third case, lazy initialization generates a new path for each object of the same type that already exists in the heap. In order to avoid state blow-up, the Apply aliasing on Lazy Initialization option can be enabled on demand, with a due loss of precision on cyclic data structures, in exchange for efficiency, when disabled.

Abstract Subsumption. Symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition depends on symbolic data. A classical solution is to establish a bound to the depth of the symbolic execution tree by specifying the maximum number of unfoldings for each loop and recursive function. As a better approach, KindSpec implements the abstract subsumption technique of [4] that determines the length of the symbolic execution paths in a dynamic way by using abstraction.

Following the classical abstract interpretation approach, programs are (symbolically) executed in KernelC by using *abstract* (approximated) data and operators rather than concrete ones. With regard to the data abstraction, when dealing with linked lists and trees we consider *summary nodes* for approximating a number of nodes in the list or tree [4]. For system states, the state abstraction function α is defined as a source-to-source transformation that approximates both primitive data and heaps. The abstract value of a primitive type object field e in an abstract (summary) node n^α is the set $\{v_1, \ldots v_k\}$ that contains the k distinct valuations v_i, $i = 1 \ldots k$, of e in the m individual nodes that are approximated by n^α, with $k \leq m$. A relation $\sqsubseteq^\alpha$ between abstract states is naturally induced such that, given two abstract states s and s', $s' \sqsubseteq^\alpha s$ whenever the set of concrete states represented by s' is included in the set of concrete states that are represented by s. Checking $\sqsubseteq^\alpha$ generally implies reasoning about logical subsumption (implication) for constraints involving primitive data, for which the Z3 SMT solver is used.

In the abstract symbolic execution of a program function, before entering a loop at the current (abstract) state s', $s' \sqsubseteq^\alpha s$ is checked for every comparable predecessor (abstract) state s of s' in the same branch. If the check succeeds, the execution of the loop stops.

With regard to the program functions, and particularly the observers, for each observer function a corresponding abstract version operates on summary nodes and preserves the original behavior. For instance, consider an observer `neg(c,n)` that returns `1` when `n` points to a node of the list `c` that contains a negative number in the `value` field, and returns `0` otherwise. The abstract version of this observer may access an abstract list that contains a summary node at the position pointed to by `n`. In such a case, it returns `1` only if all the concrete values in the abstract `value` field of the summary node are negative, `0` when all of them are positive, and a symbolic value `?v` otherwise.

3.2 KindSpec output

KindSpec provides two main outputs: 1) the contract <*Pre*, *Post*, *Loc*> for the selected function; and 2) a list of (not necessarily correct) *Candidate* axioms.

Figure 3 shows the synthesized contract and candidate axioms for our running example (with enabled aliasing and abstract subsumption) as they are displayed.

```
PRECONDITION Pre:
(isN(c)=0 ^ lenC(c)=0 ^ isE(c)=1) || (isN(c)=0 ^ lenC(c)=1 ^ isE(c)=0) ||
(isN(c)=0 ^ lenC(c)=2 ^ isE(c)=0) || (isN(c)=0 ^ lenC(c)=3 ^ isE(c)=0)
--------------------------------------------------
POSTCONDITION Post:
A1: (isN(c)=0 ^ lenC(c)=(NPE, 56) ^ isE(c)=0) =>
    (isN(c)=0 ^ lenC(c)=(NPE, 56) ^ isE(c)=0 ^ ret=(NPE, 24))
A2: (isN(c)=0 ^ lenC(c)=0 ^ isE(c)=1) => (isN(c)=0 ^ lenC(c)=0 ^ isE(c)=1 ^ ret=0)
A3: (isN(c)=1 ^ lenC(c)=0 ^ isE(c)=(NPE, 41)) =>
    (isN(c)=1 ^ lenC(c)=0 ^ isE(c)=(NPE, 41) ^ ret=0)
A4: (isN(c)=0 ^ lenC(c)=1 ^ isE(c)=0) => (isN(c)=0 ^ lenC(c)=1 ^ isE(c)=0 ^ ret=1)
A5: (isN(c)=0 ^ lenC(c)=2 ^ isE(c)=0) => (isN(c)=0 ^ lenC(c)=1 ^ isE(c)=0 ^ ret=1)
A6: (isN(c)=0 ^ lenC(c)=3 ^ isE(c)=0) => (isN(c)=0 ^ lenC(c)=1 ^ isE(c)=0 ^ ret=1)
--------------------------------------------------
LOCATIONS Loc:
c->lsize
c->elems
c->elems->next
c->elems->next->next
c->elems->next->next->next
--------------------------------------------------
CANDIDATE AXIOMS Post#:
C1: (isN(c)=0 ^ isE(c)=0 ^ lenC(c)=?l0 + 2 ^ ?l0 >= 2) =>
    (isN(c)=0 ^ isE(c)=0 ^ lenC(c)=?l0 ^ ?l0 >= 2 ^ ret=1) ^
C2: (isN(c)=0 ^ isE(c)=0 ^ lenC(c)=?l0 + 2 ^ ?l0 >= 2) =>
    (isN(c)=0 ^ isE(c)=0 ^ lenC(c)=1 ^ ret=1)
```

Figure 3. Inferred contract for the `collapseC` function in Figure 2.

First, the precondition is shown as the disjunction of all the initial scenarios for which the contract is defined (admissible inputs). Following the C convention, note that the value `0` is used to represent the boolean value `false`, whereas the value `1` stands for `true`. The postcondition consists of the generated axioms that describe all (successful and exceptional) inferred program behaviors. We note that one single axiom might correspond to a number of branches in the symbolic execution tree of the function. The third contract component is the set of overwritten program locations in the final symbolic states, which are identified and harvested as a by-product of the symbolic execution.

Every axiom $(p \Rightarrow q)$ that describes exceptional behavior can be easily identified since it contains (in either p or q) at least one equation $l = (errorCode, pc)$, where *errorCode* is an error identifier (see Table 1) and *pc* is the last executed instruction that triggered the exception. In Figure 3, the exceptional axiom `A1` describes an execution scenario where, starting from a list `c` that is neither null nor empty, both the observer `lenC` and the target function `collapseC` itself return an exception. The associated program counters, `56` and `24`, correspond

to individual instructions `n = n->next` attempting to access the `next` field of a `null` pointer `n`. In fact, this may happen in the case when `c` is not cyclic, although cyclicity of `c` was taken for granted in the data type implementation. The exceptional axiom `A3` characterizes the case when the input argument is a reference that points to a `null` position, which causes `isE` to trigger an exception. As for the axiom `A2`, it specifies that, whenever the input list is empty, nothing is deleted and the list is still empty after the execution. Axioms `A4` to `A6` specify the cases when the list contained a cycle (whose length is respectively equal to 1, 2, and 3) and it was actually collapsed.

With regard to the (overly-general) candidate axioms `C1` and `C2`, they result from cutting down an infinite loop by means of abstract subsumption and can be later refined as follows: 1) First, for those candidate axioms that are suspicious to have spurious instances, a *falsification* subprocess can be triggered. This process is undertaken by i) building initial configurations that satisfy the axiom antecedent; ii) running the modifier function on those initial configurations; and iii) checking if the results comply with the axiom consequent. The initial configurations (input values) are currently generated interactively (with specific values provided by the user). If the falsification check succeeds, the axiom is considered to be *falsified* and is consequently left out. 2) However, some candidate axioms might be indeed correct (hence they cannot be falsified). To deal with this, users are allowed to mark *trusted* candidates as correct, so that they become a part of the contract. 3) Finally, redundant axioms are removed by means of a subsumption checking process that gets rid of duplicate axioms and less general instances. In our leading example, candidate `C1` is spurious and can be trivially falsified for any input list, whereas `C2` is correct and can be trusted. Moreover, a generalization of `C2` can then be computed that subsumes `A4-6`. Generalizations are achieved by recognizing families of axioms such as `C2` and `A4-6`, which are sets of axioms where all observer equations of the antecedent and consequent are equal modulo renaming except for one observer (arithmetic constraints can differ too), and then hypothesizing a more general axiom that can be used to replace all of the family axioms. This is done by simply trying some frequent patterns for constraint generalization; e.g., the constraint `?l0>= 1` generalizes a series of constraints `?l0=1`; ...; `?l0=i`; `?l0>i`, with `i >=2`, when these constraints appear in the antecedent of `i` different axioms. However, since generalization for arithmetic constraints is still an open problem, some of our constraint generalization patterns might not lead to correct generalizations and Z3 is queried to check if the involved constraints are actually equivalent. In the case when the verification fails, the user is prompted to either accept or reject the generated hypothesis. In the example, after simplifying `(lenC(c)=?l0+2 ^ ?l0>=2)` in the antecedent of `C2` into `(lenC(c)=?l0 ^ ?l0>3)`, and flattening the equations `lenC(c)=v` of `A4-6` as `(lenC(c)=?l0 ^ ?l0=v)`, for `v=1` to `v=3`, we can recognize the pattern `?l0=1`; `?l0=2`; `?l0=3`; `?l0>3` that is logically equivalent to `?l0>=1`. Then, the following hypothesis `H1` is generated that subsumes `C2` and `A4-A6`:

```
(isN(c)=0^lenC(c)=?l0^isE(c)=0^?l0>= 1) =>
                                    (isN(c)=0^lenC(c)=1^isE(c)=0^ret=1).
```

At the end of the process, the final contract hence consists of axioms `A1-3` and `H1`.

The errors reported by the exceptional axioms `A1` and `A3` may be later used for providing provisional program patches by using the program counter *pc* to determine the right program point to insert the patch. In our leading example, regarding the exception `(NPE, 56)` in axiom `A1`, a simple patch may consist in guarding the offending memory access with an appropriate check so that the guarded access is safe. This can be easily done replacing the sentence `n = n->next;` on line 56 by the guarded assignment `if(n != NULL) {n = n->next;} else {break;}`

4 System architecture

The architecture of the KindSpec tool is depicted in Figure 4. It essentially consists of a main module that orchestrates the inference by invoking a number of specialized components as follows:

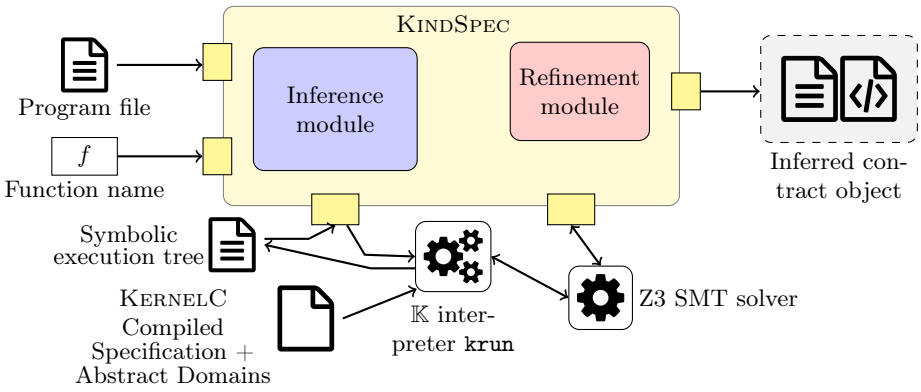

Figure 4. Architecture of the KindSpec system.

1. The 𝕂 interpreter (named `krun`) symbolically executes the compiled 𝕂 specification of the KernelC language and relies on the SMT Solver Z3 for pruning infeasible execution branches. Z3 is also used to simplify the path conditions and optimize the process. The 𝕂 interpreter runs in Linux and MacOS X.
2. The inference module builds the axioms that explain the initial and final states of the symbolic execution paths and generates the inferred contract by piecing together the function pre-condition, post-condition, and affected program locations. Since the elapsed time for each execution of the 𝕂 interpreter is rather high (15-20 seconds each, on average), in order to improve performance, our implementation exploits multithreading, with an independent thread for each symbolic execution path.

3. The refinement module applies the refinement post-processing, which consists in duplicate elimination, trusting, (test-based) falsification of overly general axioms, and axiom subsumption checking to get rid of less general axioms.

KindSpec is currently able to infer contracts for KernelC programs with the summary node abstraction for linked data structures such as lists. Nevertheless, the implemented infrastructure has been designed to support further abstract domains and other languages for which a $\mathbb{K}$ semantics is given.

The implementation of KindSpec contains about 7500 lines of Java source code for the back-end and 2300 lines of $\mathbb{K}$ code for the extended, abstract KernelC language specification. The abstract domain and operators have been integrated into the abstract KernelC semantic definition written in $\mathbb{K}$. Since summary nodes occur in the memory heap during symbolic execution, this means that abstractions are directly handled by $\mathbb{K}$'s symbolic engine.

5 Experiments

We evaluated KindSpec on a set of classical contract inference benchmark programs that size in the hundreds or tens of lines of code. Our test platform was an Intel Core2 Quad CPU Q9300(2.50GHz) with 6 GB of RAM running $\mathbb{K}$ v3.4 on Maude v2.6. Table 2 summarizes the figures that we obtained for programs that contain (both cyclic and acyclic) data structures. The specific feature that we test within each example is described in the *Program* column. The *LOC* column shows the program size (in lines of code). The *Function* column indicates the name of the target function. The *#Obs* column is the number of observer program functions. The *#Paths* column shows the number of root-to-leaf symbolic paths in the deployed trees, while the *#Axms* column reflects how many *different* axioms are retrieved from the final states of the paths. The *#Cand ax* column indicates the number of overly general axioms, and the *Final contract* column indicates the final number of correct axioms that are distilled as a result of the whole process. It might happen that this number is smaller than *#Axms* due to the reduction given by generalized candidate axioms subsuming more specific axioms.

With respect to the time cost, specification inference is known to be expensive for accurate and strong properties. We distinguish between the amount of time taken for the symbolic execution of methods performed by $\mathbb{K}$ and the elapsed time of the processing applied by our inference algorithm. The time spent in $\mathbb{K}$'s symbolic execution ranges from 1 min. to 5 min. depending on the quantity and complexity of the method definitions and the number of cores in the user's CPU. On the other hand, the time taken for actual inference of contracts (once the symbolic execution trees have been deployed) ranges from approximately 150 ms to 300 ms. Our results are very encouraging since they show that KindSpec can infer compact and general contracts for programs that deal with common data structures without fixing the size of dynamic data structures or limiting the number of iterations to ensure termination. The tool infers contracts for

Table 2. Experimental results for KindSpec on programs manipulating lists.

Program	**LOC**	**Function**	**#Obs**	**#Paths**	**#Axms**	**#Cand ax**	**Final contract**
cyclic_lists.c *(running example)*	95	collapseC	3	22	8	2	4
insert.c *(linked lists)*	120	insert	5	17	10	3	5
insert_excp.c *(version with errors)*	90	insert	5	16	9	3	4
deallocate.c *(reduction of heap size)*	59	deallocate	2	5	5	1	2
reverse.c *(heap mutation)*	70	reverse	4	7	6	1	3
del_circular.c *(circular lists)*	69	delCircular	3	13	7	1	4
append.c *(2 symbolic lists, 1 loop)*	60	append	3	32	32	10	4

challenging programs that have recursive predicates, linked and doubly-linked lists, and circular/cyclic lists. Assuming the program contains an appropriate set of observers, KindSpec is able to infer accurate contracts for all of our benchmarks.

Let us provide a brief discussion of relative benefits w.r.t. existing tools for related tasks. Most of the tools that implement contract inference techniques that are described in the related literature are no longer publicly available for use or experimentation. In the following, we compare KindSpec with the three available tools Daikon [19] (which is based on testing), *AngelicVerifier* [16] (which implements a weakest precondition calculus), and the commercial tool for proving memory safety of C code Infer [10] (which infers separation logic assertions[3] aimed to ease the identification of program bugs).

Table 3 illustrates a comparison of key features of the considered tools. There is a column for each tool, and the nine rows stand for the accepted *input language(s)*; the artifacts that have to be provided as *tool input*; the *specification type* (either full contracts or just function preconditions) and its nature,

[3] In separation logic [37], heap predicates are constituted by "separated" sub-formulae which hold for disjoint parts of the heap. They represent either individual memory cells, which are encoded by using points-to heap predicates (i.e., $e_1 \mapsto e_2$ represents that the heap contains a cell at address e_1 with contents e_2), or sub-heaps (*heaplets*), which are encoded by predicates that collapse various heap locations.

Table 3. Comparison between KindSpec and other competing tools.

	KindSpec	**Daikon**	***Angelic Verifier***	**Infer**
Input language	C	C, .NET, Java, Perl, Eiffel	C, Java	C, .NET, Java
Tool input	Source code (C) + function name	Source code (Daikon)+ test cases	Intermediate code (Boogie) + input specification	Intermediate code (SIL)
Specification type	Function-level contracts	Heap-level contracts	Function-level preconditions	Heap-level contracts
Error cases	Yes	No	No	No
Technology	Abst. Symb. Exec. in $\mathbb{K}$	Instrumentation + Testing	Weakest prec. calculus	Abst. interp. + Bi-abduction
GUI	Yes (desktop)	No	No	Yes (online)
Last update	2020	2021	2018	2021
Operating system	Linux, MacOS X	Windows, Linux, MacOS X	Windows, Linux	Windows, Linux, MacOS X
Standalone	Yes	No	No	Yes

i.e., whether it is described at *function-level* (meaning that it is expressed in terms of the observer program functions) or at *heap-level* (that strictly capture the heap assignments); whether *error* (or exception) *cases* are captured in the specification; the underlying inference *technology*; the availability of a *GUI*; the date of the *last update* of the tool; *operating system* compatibility;[4] and finally, whether it is a *standalone* artifact. As shown in Table 3, KindSpec leverages symbolic execution infrastructure to generate meaningful specifications for heap-manipulating C code. Actually, only KindSpec delivers high (function-)level whole contracts, easier to read by the user, that moreover cope with exceptional behavior in an explicit way.

Daikon [19] (and the no longer available DySy [14]) aims to obtain (heap-level) properties by extensive testing. Daikon works by running an instrumented program over a given test suite, then storing all the values taken by the program variables at both the start and the end of these runs.[5] Microsoft's *Angelic Verifier* [16] applies a weakest precondition calculus to infer likely (function-level) preconditions from a given set of program traces that failed to be verified (and thus were considered as *uncertain*), aimed to retry the verification task. The contract discovery tool Infer applies to very large programs that can be written in several source languages (C, .NET languages, and Java) but focuses on pointer safety properties concerning the heap. Unlike KindSpec, it reasons over a semantic, analysis-oriented Smallfoot Intermediate Language (SIL) that repre-

[4] We tested the tools in Windows (versions 7 and 10), Linux (Ubuntu 18.04) and MacOS X (10.13 High Sierra).

[5] In contrast, DySy relied in concolic execution (a combination of symbolic execution with dynamic testing) to obtain more precise (heap-level) axiomatic properties for non-instrumented programs.

sents source programs in a simpler instruction set describing the program's effect on symbolic heaps [41]. This is similar to *AngelicVerifier*, which relies on the intermediate language Boogie, designed for verification. While several compilers translate to Boogie programs that are written in high-level languages supporting heap manipulation (e.g., C), the inferred preconditions are expressed in terms of Boogie, thus lacking a direct correspondence to the source language.

While Infer synthesizes Hoare triples that imply memory safety and can identify potential flaws (which is indeed its main feature), no precondition is synthesized for failing attempts to establish safety; these findings are simply returned to the user in the form of a bug report. Also, the contracts generated by Infer are not accessible to users through the web interface of the tool. A last distinguished feature of our tool is the refinement functionality that provides interactive support, through a graphical user interface, for axiom falsification and trusting.

6 Conclusion and related work

Let us briefly discuss those strands of research that have influenced our work the most, independently of the current availability of a companion automated tool. Our axiomatic representation is inspired by Axiom Meister [40] (currently unavailable), which relied on a model checker for (bounded) symbolic execution of .NET programs and generates either Spec# specifications or parameterized unit tests. Similarly to [40], we aim to infer rich, *function-level* characterizations that are easily understandable; however, we 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 also kept separate in $\mathbb{K}$. Moreover, our approach is generic, and thus potentially transferable with reasonable effort to other programming languages for which a semantic definition is formalized in $\mathbb{K}$.

Besides Daikon [19] and DySy [14], other approaches based on testing led to the development of AutoInfer [42] for inferring heap-level postconditions, the invariant discovery tool DIDUCE [28], the QuickSpec tool [13] that distils equational laws concerning Haskell functions, and the (never released) experimental prototype of Henkel and Diwan [29] that generalizes the results of running tests on Java class interfaces as an algebraic specification.

An alternative approach to software specification discovery is based on inductive machine learning (ML) such as the PSYCO project for Java Pathfinder [23] (that combines ML with symbolic execution to synthesize temporal interfaces for Java classes in the form of finite-state automata), and Adabu [15] (that mines state-machine models of object behavior from runs of Java programs).

Regarding the specific thread of research that concerns the inference of specifications for heap-manipulating programs with dynamic data structures, special mention deserve *angelic verification* [16] and the distinct separation logic-based approaches, from the early *footprint* analysis technique that discovers program preconditions [11] to the automatic deep-heap analysis tool Infer [9]. Typical

properties that can be inferred by these tools regard safe memory access or the absence of memory leaks. No longer maintained are Infer's predecessor, Abductor [12], and the shape analysis tool SpInE that synthesizes heap summaries à la Hoare [27]. Also based on separation logic are [32] and [33], which rely on symbolic execution with abstraction to provide verified program repair and (heap-level) invariant generation, respectively.

This work improves existing approaches and tools in several ways besides those mentioned in Section 5. While testing-based approaches and learning-based approaches are limited to ascertain properties that *have not been previously falsified* by a (finite) number of examples or tests, KindSpec is able to guarantee correctness/completeness under some conditions in many practical scenarios [2]; moreover, the correctness of the delivered specifications can also be ensured by using the existing $\mathbb{K}$ formal analysis tools. In comparison to classical symbolic methods, we do not need to fix the size of arrays and dynamic structures or limit the number of iterations to ensure termination in the presence of loops; instead, we handle unbounded structures by means of lazy initialization and ensure termination of symbolic execution procedures by using abstraction. Finally, our experiments in Section 5 show that KindSpec infers rich contracts for challenging programs having recursive predicates and complex, dynamically allocated nested data structures such as singly/doubly linked lists, being them circular/cyclic or not, which are handled by few competing tools. In order to improve accuracy and applicability of our tool, in future work we plan to extend the supported abstract domains to cope with more sophisticated data structures [7,17] and provide support for automated verification.

References

1. ANSI/ISO IEC 9899:1999 Standard for C Language (C99), Technical Corrigendo 3 (2007)
2. Alpuente, M., Pardo, D., Villanueva, A.: Symbolic Abstract Contract Synthesis in a Rewriting Framework. In: Proc. LOPSTR '16. LNCS, vol. 10184, pp. 187–202 (2016). https://doi.org/10.1007/978-3-319-63139-4_11, https://link.springer.com/chapter/10.1007/978-3-319-63139-4_11
3. Alpuente, M., Pardo, D., Villanueva, A.: Abstract Contract Synthesis and Verification in the Symbolic K Framework. Fundamenta Informaticae (2020), to appear
4. Anand, S., Păsăreanu, C.S., Visser, W.: Symbolic execution with abstraction. STTT **11**(1), 53–67 (2009). https://doi.org/10.1007/s10009-008-0090-1, https://link.springer.com/article/10.1007/s10009-008-0090-1
5. Baldoni, R., Coppa, E., D'Elia, D., Demetrescu, C., Finocch, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. **51**(3) (2018)
6. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2010), https://frama-c.com/download/acsl_1.4.pdf
7. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Proc. CAV '07. LNCS, vol. 4590, pp. 178–192 (2007). https://doi.org/10.1007/978-3-540-73368-3_22, https://link.springer.com/chapter/10.1007/978-3-540-73368-3_22

8. Bidoit, M.: Algebraic specification of exception handling and error recovery by means of declarations and equations. In: Proc. ALP 1984. LNCS, vol. 172, pp. 95–108 (1984). https://doi.org/10.1007/3-540-13345-3_8, https://link.springer.com/chapter/10.1007/3-540-13345-3_8
9. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Proc. NFM '11. LNCS, vol. 6617, pp. 459–465 (2011)
10. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving Fast with Software Verification. In: Proc. NFM '15. LNCS, vol. 9058, pp. 3–11 (2015). https://doi.org/10.1007/978-3-319-17524-9_1
11. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Footprint Analysis: A Shape Analysis That Discovers Preconditions. In: Proc. SAS '07. LNCS, vol. 4634, pp. 402–418 (2007). https://doi.org/10.1007/978-3-540-74061-2_25, https://link.springer.com/chapter/10.1007/978-3-540-74061-2_25
12. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. J. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700, http://doi.acm.org/10.1145/2049697.2049700
13. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: Guessing Formal Specifications Using Testing. In: Proc. TAP '10. LNCS, vol. 6143, pp. 6–21 (2010). https://doi.org/10.1007/978-3-642-13977-2_3, https://link.springer.com/chapter/10.1007/978-3-642-13977-2_3
14. Csallner, C., Tillmann, N., Smaragdakis, Y.: DySy: Dynamic Symbolic Execution for Invariant Inference. In: Proc. ICSE '08. pp. 281–290. ACM (2008). https://doi.org/10.1145/1368088.1368127, http://doi.acm.org/10.1145/1368088.1368127
15. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining Object Behavior with ADABU. In: Proc. WODA '06. pp. 17–24. ACM (2006). https://doi.org/10.1145/1138912.1138918, http://doi.acm.org/10.1145/1138912.1138918
16. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic Verification: Precise Verification Modulo Unknowns. In: Proc. CAV '15. LNCS, vol. 9206, pp. 324–342 (2015). https://doi.org/10.1007/978-3-319-21690-4_19
17. Distefano, D., O'Hearn, P.W., Yang, H.: A Local Shape Analysis Based on Separation Logic. In: Proc. TACAS '06. LNCS, vol. 3920, pp. 287–302 (2006). https://doi.org/10.1007/11691372_19, https://link.springer.com/chapter/10.1007/11691372_19
18. Ellison, C., Roşu, G.: An Executable Formal Semantics of C with Applications. In: Proc. POPL '12. pp. 533–544. ACM (2012). https://doi.org/10.1145/2103656.2103719, http://doi.acm.org/10.1145/2103656.2103719
19. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015, http://www.sciencedirect.com/science/article/pii/S016764230700161X
20. Gazzola, L., Micucci, D., Mariani, L.: Automatic Software Repair: A Survey. IEEE Transactions on Software Engineering pp. 1–1 (2018). https://doi.org/10.1109/TSE.2017.2755013

21. Gehani, N.H.: Exceptional C or C with exceptions. Software: Practice and Experience **22**(10), 827–848 (1992). https://doi.org/10.1002/spe.4380221003, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380221003
22. Gherghina, C., David, C.: A Specification Logic for Exceptions and Beyond. In: Proc. ATVA '10. LNCS, vol. 6252, pp. 173–187 (2010)
23. Giannakopoulou, D., Rakamaric, Z., Raman, V.: Symbolic Learning of Component Interfaces. In: Miné, A., Schmidt, D. (eds.) Proc. SAS '12. LNCS, vol. 7460, pp. 248–264 (2012). https://doi.org/10.1007/978-3-642-33125-1_18
24. Gogolla, M., Drosten, K., Lipeck, U.W., Ehrich, H.D.: Algebraic and operational semantics of specifications allowing exceptions and errors. Theoretical Computer Science **34**(3), 289–313 (1984). https://doi.org/10.1016/0304-3975(84)90056-2, http://www.sciencedirect.com/science/article/pii/0304397584900562
25. Goguen, J.: Abstract Errors for Abstract Data Types. In: Formal Description of Programming Concepts, pp. 491–522. North-Holland (1979)
26. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theoretical Computer Science **105**(2), 217–273 (1992). https://doi.org/10.1016/0304-3975(92)90302-V, http://www.sciencedirect.com/science/article/pii/030439759290302V
27. Gulavani, B.S., Chakraborty, S., Ramalingam, G., Nori, A.V.: Bottom-up Shape Analysis Using LISF. TOPLAS '11 **33**(5), 17:1–17:41 (2011). https://doi.org/10.1145/2039346.2039349, http://doi.acm.org/10.1145/2039346.2039349
28. Henkel, J., Diwan, A.: Discovering Algebraic Specifications from Java Classes. In: Proc. ECOOP '03. LNCS, vol. 2743, pp. 431–456 (2003). https://doi.org/10.1007/978-3-540-45070-2_19, https://link.springer.com/chapter/10.1007/978-3-540-45070-2_19
29. Henkel, J., Reichenbach, C., Diwan, A.: Discovering Documentation for Java Container Classes. IEEE Transactions on Software Engineering **33**(8), 526–543 (2007). https://doi.org/10.1109/TSE.2007.70705
30. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252, http://doi.acm.org/10.1145/360248.360252
31. Liskov, B., Guttag, J.: Abstraction and Specification in Program Development. MIT Press (1986)
32. Logozzo, F., Ball, T.: Modular and Verified Automatic Program Repair. In: Proc. OOPSLA '12. pp. 133–146. ACM (2012). https://doi.org/10.1145/2384616.2384626, http://doi.acm.org/10.1145/2384616.2384626
33. Magill, S., Nanevski, A., Clarke, E., Lee, P.: Inferring invariants in Separation Logic for imperative list-processing programs. In: Proc. 3rd SPACE Workshop (2006)
34. Meyer, B.: Applying 'design by contract'. Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279
35. Padmanabhuni, S., Ghose, A.K.: Inductive constraint logic programming: An overview. In: Antoniou, G., Ghose, A.K., Truszczyński, M. (eds.) Learning and Reasoning with Complex Representations. pp. 1–8. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)
36. Poigné, A.: Partial algebras, subsorting, and dependent types. In: Recent Trends in Data Type Specification, Proc. ADT '87. LNCS, vol. 332, pp. 208–234 (1987)
37. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. LICS '02. pp. 55–74 (2002). https://doi.org/10.1109/LICS.2002.1029817

38. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012, http://www.sciencedirect.com/science/article/pii/S1567832610000160
39. Schulz, S.: Simple and Efficient Clause Subsumption with Feature Vector Indexing. In: Bonacina, M.P., Stickel, M.E. (eds.) Automated Reasoning and Mathematics - Essays in Memory of William W. McCune. Lecture Notes in Computer Science, vol. 7788, pp. 45–67. Springer (2013)
40. Tillmann, N., Chen, F., Schulte, W.: Discovering Likely Method Specifications. In: Proc. ICFEM '06. LNCS, vol. 4260, pp. 717–736 (2006). https://doi.org/10.1007/11901433_39, https://link.springer.com/chapter/10.1007/11901433_39
41. van Tonder, R., Goues, C.: Static Automated Program Repair for Heap Properties. In: Proc. ICSE '18. pp. 151–162. ACM (2018). https://doi.org/10.1145/3180155.3180250, http://doi.acm.org/10.1145/3180155.3180250
42. Wei, Y., Furia, C.A., Kazmin, N., Meyer, B.: Inferring Better Contracts. In: Proc. ICSE '11. pp. 191–200. ACM (2011). https://doi.org/10.1145/1985793.1985820, http://doi.acm.org/10.1145/1985793.1985820