An Active Learning Approach to Synthesizing Program Contracts

Sandip Ghosal[1], Bengt Jonsson[1], and Philipp Rümmer[1,2]

[1] Uppsala University, Sweden
[2] University of Regensburg, Germany

Abstract. Contracts capture assumptions (preconditions) and guarantees (postconditions) of functions in a software program, and are an important paradigm for documenting program code, for program understanding, and to enable modular program verification. In this paper, we focus on contracts for stateful software modules, for instance modules implementing data-structures like queues. Such modules offer different kinds of functions to their environment: *observers,* which are pure functions used to query the state of the module; and *mutators,* which can change the module state. We present a novel technique to synthesize contracts for the mutators of a module, in which pre- and postconditions are expressed as Boolean combinations of the observers. Our method builds on existing algorithms for active learning of register automata to model the possible behaviours of the stateful module. We then present techniques for synthesizing contracts from a learned register automaton. The entire method is fully black-box and automated. Based on our proposed approach, we develop a tool called `CoGent` that generates a set of contracts for a mutator from a given register automaton of a module. Finally, we evaluate our tool using the APIs for various data structures.

1 Introduction

The annotation of program functions with contracts, consisting of pre- and postconditions, serves several purposes. Contracts are an important form of documentation, and are as such widely used to describe the intended use of library and API functions. Contracts give rise to the Design-by-Contract (DbC) methodology [18], by stating both the assumptions made about the states in which a function may be called, and the guarantees established in return by the function. In formal verification, contracts are the main vehicle to decompose larger programs into smaller units that can be analysed in isolation (e.g., [6, 7]).

It is non-trivial, however, to come up with correct contracts for a given function. In most of today's code bases, functions are documented only with unstructured text, or with informal contracts in which pre- and postconditions are stated in natural language. Like any kind of formal specification, the process of writing *formal* contracts (with pre- and postconditions being logical formulas) is an extremely time-consuming and error-prone process, and is in fact sometimes considered the main bottleneck preventing application of formal methods in an industrial context.

Over the last years, researchers have therefore considered the automated inference of formal contracts from implementations (e.g., [1,2,4,5,13,19,20,22]). Such inferred contracts can serve as documentation of existing programs, and as auxiliary annotations in verification. However, although various approaches to contract inference have been proposed, methods that are (i) scalable enough to handle real-world code bases, (ii) precise enough to generate correct and complete contracts, (iii) refined enough to produce contracts that are human-readable so far remain elusive.

In this paper, we present a new approach to automatically infer contracts for software modules. Our approach starts with applying existing active black-box learning methods [8,9] to build a behavioural model of a program in the form of a finite-state register automaton. We then construct contracts for all mutator functions of the software module in terms of the available observer functions: for this, state transitions associated with a mutator are analysed, and the effects of the transitions are summarized using observers. Under certain assumptions on the shape of the automaton, the computed contracts are guaranteed to describe only the behaviour of the module that is reachable, i.e., they implicitly take module invariants into account. This is because the reachable states of the automaton correspond to the module states that are reachable from some designated initial state.

The *contributions* of this paper are:

– A new black-box framework for synthesizing program contracts for software modules (Section 4).
– An algorithm to extract program contracts from finite-state register automata (Section 4).
– An implementation of our approach in the tool `CoGent` (Section 5), and an evaluation of our method using software modules taken from the Java API (Section 6).

Outline: The remainder of this paper is structured as follows. Section 2 illustrates program contracts with an example of a stateful data structure that serves as a running example for this paper. Section 3 describes the semantics of program contract, and introduces to basic concepts and notations for dealing with register automata and active automata learning. Section 4 outlines the steps for synthesizing program contracts with illustrations using the running example. Section 5 describes the implementation details of our tool, `CoGent`, and Section 6 presents its evaluation on various data structures. Section 7 compares our approach with some of the earlier attempts for synthesizing program contracts in the literature, followed by the conclusions in Section 8.

2 Motivating Example

As an illustration, consider the Java class `BoundedList` in Fig. 1. It contains methods `BoundedList`() for constructing a list object, `list`, of maximum size defined by `maxSize`. The class further contains the methods `push` and `pop`, which

```java
public class BoundedList {
  private LinkedList<Integer> list;
  private int maxSize;

  public BoundedList(){
    maxSize = DEFAULT_SIZE;
    list = new LinkedList<Integer>();
  }

  public void push(Integer e) {
    if(maxSize > list.size()) list.push(e);
  }

  public int pop() {
    return list.pop();
  }

  public boolean contains(Integer e) {
    return list.contains(e);
  }

  public boolean isEmpty() {
    return (list.size == 0) ?$\true$: false;
  }

  public boolean isFull() {
    return (list.size == maxSize) ?$\true$: false;
  }
}
```

Fig. 1: A module for a `BoundedList` (in Java)

$$\{(p = q) \wedge \neg\texttt{isFull}()\} \quad \texttt{push}(p) \quad \{\texttt{contains}(q)\}$$

$$\left\{\begin{array}{c} (\neg(p = q) \wedge \neg\texttt{contains}(q)) \\ \vee\ (\texttt{isFull}() \wedge \neg\texttt{contains}(q)) \end{array}\right\} \quad \texttt{push}(p) \quad \{\neg\texttt{contains}(q)\}$$

Fig. 2: Contracts for the `BoundedList` module in Fig. 1

are mutators, and the observer methods `contains`, `isEmpty`, and `isFull`. The class `BoundedList` internally uses the `LinkedList` class available in JDK v1.8. The method `push` takes an integer as an input parameter and inserts the integer into the `list`. Method `pop` does not accept any parameter but removes an element from the `list` and outputs the removed integer. The method `contains` returns *True* if the argument passed in the method already exists in the `list`, and *False* otherwise. Methods `isEmpty` and `isFull` return *True* if the `list` is empty and full respectively, otherwise, *False*. The module serves as a running example for illustrating our proposed approach.

A contract relates a method call with the module states immediately before and after that call. Being in a black-box setting, we cannot refer directly to the internal module state. Instead we refer to the module state indirectly via the return values of calls to observer methods. An observer method does not modify the state of the module, and is used to extract information about the module state.

Let a *condition* be a Boolean combination of observer calls $f(r_1, \ldots, r_m)$, where $r_1, \ldots, r_m$ are variables of the appropriate types, and constraints formulated using the predicates from some background theory. In this paper, we define a *contract* for a method m with parameters $p_1, \ldots, p_n$ in a module to be of the form

$$\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$$

where P, called the *precondition* and Q, called the *postcondition*, are conditions. The conditions P, Q can contain variables $p_1, \ldots, p_n$, as well as further variables used to relate pre- and post-states. As a simplifying assumption, and without loss of generality, every variable in P or Q that does not occur among the $p_1, \ldots, p_n$ has to occur as an argument of some observer in P or Q. Let the *parameters* of the contract be $p_1, \ldots, p_n$ together with additional variables appearing in P, Q.

A contract $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$ is *valid* if, for every valuation of variables occurring in the contract, whenever m is called in a reachable state of the module in which P is *True*, the method call $m(p_1, \ldots, p_n)$ terminates and leaves the module in a state in which Q is *True*.

As illustration, for the module `BoundedList` in Fig. 1, we aim to synthesize contracts for the mutators `push` and `pop`. The contracts may include the given Boolean observer methods `contains`, `isEmpty`, and `isFull`, as well as relations between the occurring parameters. To this end, we first compute a model of the module in terms of a register automaton. While the behaviour of a software module can in general not be described by a finite register automaton, such a finite automaton can be derived for data structures with bounded capacity. For `BoundedList` with `maxSize = 2`, for example, the computed automaton has four locations and two registers, see Fig. 3. The register automaton captures the reachable behaviour of both the mutators and the observers.

To generate contracts for a mutator m, we then consider the transitions that are associated with m. Such transitions describe how the return values of observer methods can change as a result of calling m: transitions can update the values of registers, and the observers are described by location-specific guards. We present an algorithm, which for each location generates a location-specific contract for m from its outgoing m-transitions; as a second step, the location-specific contracts are then combined to obtain an overall contract for m.

In general, we would like generated contracts to be both *valid* and *maximal,* by which we mean that the precondition cannot be weakened without making the contract invalid. Two example contracts for the `push` method are given in Fig. 2. We can observe that the first contract in Fig. 2 is valid, but it is not maximal, since the postcondition could also be established by assuming `contains`(q) already in the precondition. The second contract is both valid and maximal.

3 Background

In this section, we give background for the contract synthesis approach, described in Section 4. The synthesis approach works by using active automata learning to obtain a register automaton model of the stateful behaviour of the module. The register automaton then forms the basis for contract synthesis. In this section, we describe program contracts, register automata and active learning.

3.1 Contracts

Throughout the paper, we assume a *background theory*, i.e., a (many-sorted) first-order language with constant, function, and relation symbols, with fixed interpretation over the appropriate domains. Terms and formulas are constructed as usual from those symbols, as well as from variables taken from a set $\mathcal{V}$. A *valuation* μ is a mapping from variables $\mathcal{V}$ to their domains. Valuations are extended to terms and assertions in the usual way. We write $\mu \models \phi$ to express that ϕ evaluates to *True* in μ.

We assume a set $\mathcal{M}$ of *methods*, each with a signature that determines the number of input parameters, their types, and the return type of the method. We assume a distinguished subset of $\mathcal{M}$, the set of *observer methods*: an observer method is special in that it does not modify any state variables. Throughout the paper, we assume that each observer method returns a Boolean value. The other methods are called *mutators.*

A *method call* is a term of form $m(d_1, \ldots, d_n)$, where m is a method action and $d_1, \ldots, d_n$ are data values from the appropriate domains. A *parameterized method call* is a term of form $m(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are variables; in this context we sometimes call them *formal parameters* of the method call.

As mentioned in Section 2, a *condition* is a Boolean combination of observer calls $f(r_1, \ldots, r_m)$, where $r_1, \ldots, r_m$ are variables of the appropriate types, and constraints formulated using the predicates from the background theory. We say that a condition P *entails* condition Q, written $P \Rightarrow Q$, if the formula $P \rightarrow Q$ is valid when every observer method is considered as an uninterpreted first-order predicate.

A *contract* is a triple $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$ consisting of a precondition P, a mutator call $m(p_1, \ldots, p_n)$, and a postcondition Q.

3.2 Register Automata

We assume a set of *registers* $x_1, x_2, \ldots$.

Definition 1 (Register automaton). *A* register automaton *(RA) is a tuple $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma)$, where L is a finite set of* locations*, $l_0 \in L$ is the* initial location*, $\mathcal{X}$ maps each location $l \in L$ to a finite set $\mathcal{X}(l)$ of registers, where in particular $\mathcal{X}(l_0) = \emptyset$, and Γ is a finite set of* transitions*. Each transition in Γ is of form*

$$\langle l, m(p_1, \ldots, p_n), g, e_{out}, x_{i_1} := e_{i_1}, \ldots, x_{i_m} := e_{i_m}, l' \rangle,$$

where $l \in L$ is a source location, $l' \in L$ is a target location, $m(p_1, \ldots, p_n)$ is a parameterized method call, g is a guard*, i.e., a conjunction of negated and non-negated relations over $p_1, \ldots, p_n$ and $\mathcal{X}(l)$, e_{out} is an expression over $p_1, \ldots, p_n$ and $\mathcal{X}(l)$, and $x_{i_1} := e_{i_1}, \ldots, x_{i_m} := e_{i_m}$ is an* assignment *which updates the registers $x_{i_1}, \ldots, x_{i_m}$ in $\mathcal{X}(l')$ with the values of expressions $e_{i_1}, \ldots, e_{i_m}$. In this work, we assume that each expression e_{i_j} is either a register in $\mathcal{X}(l)$ or a formal parameter in $p_1, \ldots, p_n$.* □

We write $\overline{x}$, $\overline{p}$, and $\overline{e}$ for tuples of registers, parameters, and expressions. Let us formalize the semantics of RAs. A *state* of an RA $\mathcal{A} = (L, l_0, \mathcal{X}, \Gamma)$ is a pair $\langle l, \mu \rangle$ where $l \in L$ and μ is a valuation over $\mathcal{X}(l)$, i.e., a mapping from $\mathcal{X}(l)$ to the appropriate domains. The *initial state* is the pair $\langle l_0, \mu_0 \rangle$ where μ_0 is the empty mapping. A *step* of $\mathcal{A}$, denoted $\langle l, \mu \rangle \xrightarrow{m(\overline{d})/\mu(e_{out})} \langle l', \mu' \rangle$, transfers $\mathcal{A}$ from $\langle l, \mu \rangle$ to $\langle l', \mu' \rangle$ on the method call $m(\overline{d})$, returning $\mu(e_{out})$, if there is a transition $\langle l, m(\overline{p}), g, e_{out}, x_{i_1} := e_{i_1}, \ldots, x_{i_m} := e_{i_m}, l' \rangle \in \Gamma$ such that

– $\mu \models g[\overline{d}/\overline{p}]$, i.e., $\overline{d}$ satisfies the guard g under the valuation μ, and
– μ' is the updated valuation which maps x_i to $\mu(e_i)$ when x_i is in $x_{i_1}, \ldots, x_{i_m}$, and maps other registers x_i in $\mathcal{X}(l')$ to $\mu(x_i)$.

A state $\langle l, \mu \rangle$ is *reachable* if there is a sequence of steps

$$\langle l_0, \mu_0 \rangle \xrightarrow{m_1(\overline{d_1})/o_1} \langle l_1, \mu_1 \rangle \quad \langle l_1, \mu_1 \rangle \xrightarrow{m_2(\overline{d_2})/o_2} \langle l_2, \mu_2 \rangle \quad \cdots \quad \langle l_{n-1}, \mu_{n-1} \rangle \xrightarrow{m_n(\overline{d_n})/o_n} \langle l, \mu \rangle$$

leading from the initial state $\langle l_0, \mu_0 \rangle$ to $\langle l, \mu \rangle$.

We can now define validity of a contract relative to a register automaton $\mathcal{A}$. Let P be a condition, let σ be a valuation of the variables in P, and let $\langle l, \mu \rangle$ be a state of $\mathcal{A}$. We say that P *is true in* $\langle l, \mu \rangle$ *under* σ, denoted $\langle l, \mu \rangle \models \sigma(P)$, if $\sigma(P)$ evaluates to true when each observer call in $\sigma(P)$, of form $obs(d_1, \ldots, d_n)$, is replaced by the value returned when calling $obs(d_1, \ldots, d_n)$ in $\langle l, \mu \rangle$.

Definition 2. *A contract* $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$ *is* valid *for a RA* $\mathcal{A}$ *if for any assignment* σ *of values to the parameters of the contract, and any reachable state* $\langle l, \mu \rangle$ *of* $\mathcal{A}$ *with* $\langle l, \mu \rangle \models \sigma(P)$*, we have that*

– *there is an output* o *and state* $\langle l', \mu' \rangle$ *with* $\langle l, \mu \rangle \xrightarrow{m(\sigma(p_1, \ldots, p_n))/o} \langle l', \mu' \rangle$*, and*
– *for any such output* o *and state* $\langle l', \mu' \rangle$ *it holds that* $\langle l', \mu' \rangle \models \sigma(Q)$.

Example
Fig. 3 showcases such a RA that serves as a model for capturing the behaviour of the `BoundedList` API (cf. Fig. 1) when the maximum capacity of the list is set to 2. The language for the model consists of sequences of API method calls. An execution of such a sequence may result in modifying the state of the list, causing state transitions, thereby producing an output sequence that adheres to the expected I/O behaviour of the methods within the sequence. The RA is composed of nodes, each representing a specific state of the list, and edges that signify state transitions. Each edge is labeled to denote the actions performed by a method during execution. In the following, we illustrate the labels corresponding to the edges for a mutator and an observer:

$$(i) \quad \frac{\texttt{pop}()\ [\!]\ \mathit{true} \rightarrow \{x_1 := x_2\}}{x_1} \qquad (ii) \quad \frac{\texttt{contains}(q)\ [\!]\ (x_1 = q) \vee (x_2 = q)}{\mathit{true}}$$

Consider a state ℓ_2 where the list has two elements stored in registers x_1 and x_2, with x_1 holding the most recently pushed element. In this state, a state

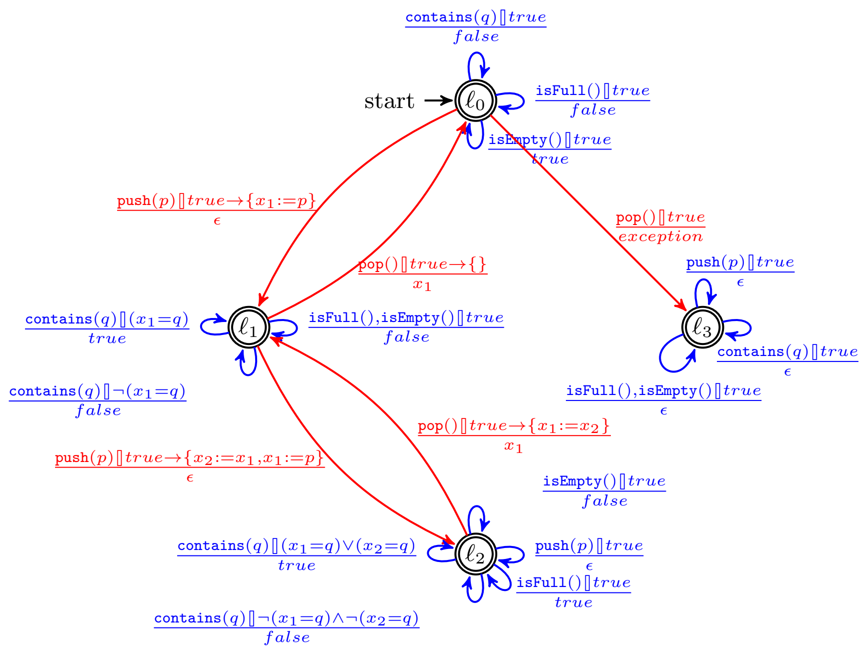

Fig. 3: Register automaton modeling a list (Fig. 1) with maximum capacity 2

transition occurs when the method `pop()` (a *mutator*) is invoked, as indicated by an edge labeled with (*i*). This label indicates that the method's guard condition is satisfied (*true*) and it outputs the recently pushed element stored into x_1 while moving the second element x_2 into x_1, thereby releases x_2. In fact, a call to `pop()` always outputs the last stored element unless the list is empty, in which case it throws an exception while leading to a trap state. The mutator `push`, however, in ℓ_2 does not change the state as the list has reached its maximum capacity. We use the notation ϵ to denote the `void` return type for method `push`. On the other hand, a method call `contains`(q) (an *observer*) in ℓ_2 checks if an element passed by the parameter q is present in the list, is labeled with (*ii*), meaning the method outputs *true* upon satisfying the condition $(x_1 = q) \vee (x_2 = q)$. Note that the label (*ii*) has no register assignments since observers do not modify register values and therefore, do not change the module state. In some cases we represent a single label for more than one method calls, method signature separated by comma (','), those exhibit similar behaviour.

3.3 Active Learning of Register Automata

The first step of our contract generation uses active automata learning (AAL) to automatically learn a register automaton model of the system under test (SUT). AAL is an automated black-box technique which *a priori* needs know only a module's methods and their signatures. Classical AAL learns finite automata or

Mealy machines from tests, using, e.g., the classic L^* algorithm [3] or the more recent TTT algorithm [16]. These, and other AAL algorithms are implemented in *LearnLib* [15]. Finite-state models do not capture how parameter values in method calls affect the module state and successive method calls. In order to capture data aspects of module behaviour, finite-state models can be, and commonly are, equipped with variables, sometimes called registers. Variables can store the values of data parameters; they can influence control flow by means of guards, and the control flow can cause variable updates. Finite state machines with variables are often called extended finite state machines (EFSMs). We will employ a specific such model, namely *register automata* (RAs), in which registers are used as variables. An extension of AAL to learning of RAs is SL^* [9], which has been implemented in RALib, an extension of LearnLib [8].

The SL^* algorithm must know the set of methods and their signatures. Like other AAL algorithms, it operates in two alternating phases: hypothesis construction and hypothesis validation. During hypothesis construction, sequences of method calls are submitted on the SUT, and the corresponding return values are observed to collect information about the module behaviour. When certain convergence criteria are met, the AAL algorithm constructs a *hypothesis*, which is a minimal deterministic RA that is consistent with the observations so far. To validate that the hypothesis agrees with the behaviour of the SUT, learning then moves to the validation phase, in which the SUT is subject to a conformance testing algorithm which aims to validate that the behaviour of the SUT agrees with the hypothesis. If conformance testing does not find any counterexample, learning terminates and returns the current hypothesis as the inferred model of the SUT. If a counterexample (i.e., a sequence of method calls on which the SUT and the hypothesis disagree) is found, the hypothesis construction phase is re-entered to build a more refined hypothesis. If the loop of hypothesis construction and validation does not terminate, this indicates that the behaviour of the SUT cannot be captured by a deterministic RA whose size and complexity is within reach of the employed learning algorithm. Still, even in these cases, the last constructed hypothesis can be used as an *approximate model* of the SUT.

4 Contract Synthesis

In this section, we describe our approach for inferring contracts for a module.

4.1 Learning a Behavioural Model

The first step of our approach is to obtain a register automaton model of the module. Sometimes, such an automaton model is readily available and can be supplied directly for generating contracts. Otherwise, such a register automaton can be learned using AAL as described in Section 3.3. Recall that AAL is fully automated and black-box, but may have practical limitations on the size of the learned model. For these reasons, we may modify the module so that its behaviour can be captured by an RA of modest size. A typical modification for container modules is to bound their capacity so that they become "full" for a small number of contained items: this will not change the set of valid contracts, as long as they do not count the number of contained items. For our running example such an automaton is shown in Fig. 3.

4.2 Generating Contracts from a Register Automaton

Given a register automaton model of our module, we present an algorithm for synthesizing contracts for each mutator method m. Recall that a contract is of form $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$. Our methodology considers synthesizing contracts for one postcondition at a time. This means, our algorithm synthesizes contracts of the above form, given as input a postcondition Q, as well as a set $\mathcal{V}_{contr}$ of variables that can occur in P; the set $\mathcal{V}_{contr}$ should include $p_1, \ldots, p_n$ and the variables in Q. In our running example, a starting postcondition Q could be $\texttt{contains}(q)$, where q is a parameter, or even $\neg\texttt{contains}(q)$. In the following description, we will use generation of preconditions P in contracts of form

$$\{P\}\ \texttt{push}(p)\ \{\texttt{contains}(q)\}$$

to illustrate the successive steps in our algorithm. The generation of contracts proceeds through the following steps.

Step 1: Generating Weakest Preconditions: For each location l, we derive the weakest precondition $wp_l(m, Q)$, i.e., the weakest condition on the registers of l under which m will terminate and yield a state in which Q evaluates to true. This can be done using standard techniques (e.g., [11]). For each location l, let $[\![Q]\!]_l$ be the condition on the registers of l and parameters of Q under which Q evaluates to *True*. The condition $[\![Q]\!]_l$ can be obtained from Q by replacing each nonnegated observer call $obs(\overline{p})$ by the disjunction of the guards of transitions from l in which $obs(\overline{p})$ return *True*, and analogously for negated observer calls. Then, letting $t_1, \ldots, t_m$ be the outgoing transitions from l for method m, $wp_l(m, Q)$ is obtained as

$$wp_l(m, Q) = \bigvee_i g_i \ \wedge\ \bigwedge_i \left(g_i \rightarrow [\![Q]\!]_{l'_i}[e_{i_1}/x_{i_1}, \ldots, e_{i_m}/x_{i_m}]\right) \tag{1}$$

where g_i is the guard, l'_i is the target location, and $x_{i_1} := e_{i_1}, \ldots, x_{i_m} := e_{i_m}$ is the assignment of t_i.

Illustration: Let us illustrate the generation of the weakest precondition $wp_{l_1}(\texttt{push}(p), \texttt{contains}(q))$ for the method $\texttt{push}(p)$ relative to the postcondition $\texttt{contains}(q)$, where l_1 is the location in the RA fragment depicted in Fig. 4. Here, l_1 is the location representing a bounded list containing a single element stored in the register x_1. The transition from ℓ_1 to ℓ_2 is the only transition from l_1 for the method $\texttt{push}$. It inserts a second element into the list, causing two elements to be stored into the registers x_1 and x_2. We first obtain $[\![\texttt{contains}(q)]\!]_{l_2}$ as $(q = x_1 \vee q = x_2)$. Using Eq. (1), we then derive the weakest precondition

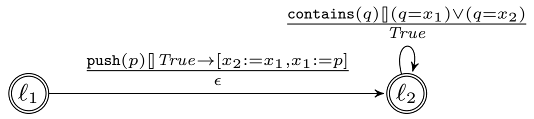
$\frac{\texttt{contains}(q) \,[\!]\, (q{=}x_1) \vee (q{=}x_2)}{True}$
$\frac{\texttt{push}(p) \,[\!]\, True \rightarrow [x_2{:=}x_1, x_1{:=}p]}{\epsilon}$
ℓ_1
ℓ_2

Fig. 4: The single transition for the method `push` from location ℓ_1, together with the transition for `contains` from location ℓ_2.

$wp_{l_1}(\texttt{push}(p), \texttt{contains}(q))$ as $(q = x_1 \vee q = x_2)[x_1/x_2, p/x_1]$, i.e., $(p = q) \vee (x_1 = q)$. $\square$

Step 2: Generating Location-Specific Preconditions: The weakest precondition $wp_l(m, Q)$ is not adequate as a precondition, since in general it mentions registers, while a precondition can only refer to the module state through observer calls. Therefore, in each location l, we generate *location-specific* preconditions $Pre_l(m, Q)$ such that $[\![Pre_l(m, Q)]\!]_l$ implies $wp_l(m, Q)$. To this end, define the set $\mathcal{O}$ as containing all possible parameterized method calls $obs(\overline{p})$ whose parameters $\overline{p}$ are taken from $\mathcal{V}_{contr}$. Next, let $\mathcal{C}_l$ be the set of formulas, which are either (i) of form $[\![obs(\overline{p})]\!]_l$ or of form $[\![\neg obs(\overline{p})]\!]_l$ for a parameterized observer call $obs(\overline{p})$ in $\mathcal{O}$, or (ii) a (nonnegated or negated) relation between variables in $\mathcal{V}_{contr}$. We generate $Pre_l(m, Q)$ as a disjunction of conjunctions of formulas in $\mathcal{C}_l$, where each disjunct is obtained as a minimal conjunction of formulas in $\mathcal{C}_l$ which implies $wp_l(m, Q)$. The generation of $Pre_l(m, Q)$ can be performed using a SAT/SMT-solver by observing that the validity of

$$(c_1 \wedge \cdots \wedge c_k) \rightarrow wp_l(m, Q)$$

is equivalent to unsatisfiability of

$$(c_1 \wedge \cdots \wedge c_k) \wedge \neg wp_l(m, Q),$$

implying that we can obtain minimal conjunctions $c_1 \wedge \cdots \wedge c_k$ with the above properties by asking a SAT/SMT-solver to produce minimal unsatisfiable subsets (MUS) of formulas in $\mathcal{C}_l \cup \{(\neg wp_l(m, Q))\}$. From each of these we obtain a conjunction of formulas in $\mathcal{C}_l$ by first removing $\neg wp_l(m, Q)$, and replacing each conjunct of form $[\![obs(\overline{p})]\!]_l$ (or $[\![\neg obs(\overline{p})]\!]_l$) by the corresponding parameterized observer method call $obs(\overline{p})$ (or $\neg obs(\overline{p})$). We discard conjunctions, such as $obs(\overline{p}) \wedge \neg obs(\overline{p})$, which are syntactical contradictions. Since the generation of minimal unsatisfiable subsets may not explicitly generate the empty set of conjuncts (which is equivalent to *False*), we finally add, for each non-parameterized observer $obs()$ for which $[\![obs()]\!]_l$ is *True*, the disjunct $\neg obs()$; by symmetry we add the disjunct $\neg obs()$ if $[\![obs()]\!]_l$ is *False*. These disjuncts are redundant in the the location-specific precondition at location l, but may be non-redundant in another location l' where $[\![obs()]\!]_{l'}$ is neither *True* nor *False*; in such a case they allow to form weaker global preconditions in Step 3. The result is our sought location-specific precondition $Pre_l(m, Q)$, structured as a disjunction of conjunctions over formulas in $\mathcal{O}$.

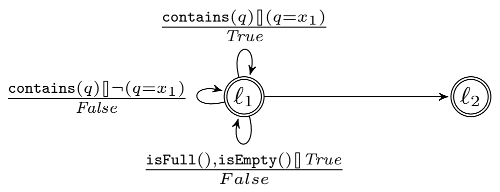
$\frac{\texttt{contains}(q) [\!] (q{=}x_1)}{True}$
$\frac{\texttt{contains}(q) [\!] \neg(q{=}x_1)}{False}$
ℓ_1
ℓ_2
$\frac{\texttt{isFull}(),\texttt{isEmpty}() [\!] True}{False}$

Fig. 5: Observers in location ℓ_1 in the automaton for `BoundedList`

Illustration: In Step 1, we obtained $wp_{l_1}(\texttt{push}(p), \texttt{contains}(q))$, the weakest precondition in location l_1, as $(p = q) \vee (x_1 = q)$. In Fig. 5, we show a fragment of the learned RA, showing calls to observers in location l_1. To construct $Pre_{l_1}(\texttt{push}(p), \texttt{contains}(q))$, we collect in $\mathcal{C}_l$ the guards for $\texttt{contains}(q)$ (i.e., $(q = x_1)$) and for $\neg\texttt{contains}(q)$ (i.e., $\neg(q = x_1)$) together with equalities and dis-equalities between occurring parameters. By interacting with a SAT/SMT solver, we identify the following minimal unsatisfiable subsets:

(i) $\{(p = q), \neg wp_{l_1}(\texttt{push}(p), \texttt{contains}(q))\}$
(ii) $\{(q = x_1), \neg wp_{l_1}(\texttt{push}(p), \texttt{contains}(q))\}$

which, after removing the negated weakest preconditions, yields the following two minimal disjuncts to be used in the precondition: (i) $(p = q)$, and (ii) $\texttt{contains}(q)$. Since none of these disjuncts entails the other, we use both when forming the formula in DNF, as $((p = q) \vee \texttt{contains}(q))$. As the final step, we consider unparameterized observer calls that always return *True* or *False* in location l_1. Considering that in l_1, the list contains one item, these are $\neg\texttt{isEmpty}()$ and $\neg\texttt{isFull}()$. Therefore, we add the two disjuncts $\texttt{isEmpty}()$ and $\texttt{isFull}()$. By making them antecedents in an implication, we can then write $Pre_{l_1}(\texttt{push}(p), \texttt{contains}(q))$ in the following way:

$$(\neg\texttt{isFull}() \wedge \neg\texttt{isEmpty}()) \rightarrow ((p = q) \vee \texttt{contains}(q)).$$

□

Step 3: Generating Global Preconditions: After obtaining location-specific preconditions, we can finally obtain a *location-agnostic* precondition $Pre(m, Q)$ as the conjunction

$$Pre(m, Q) = \bigwedge_{l \in L} Pre_l(m, Q) \tag{2}$$

over location-specific preconditions for all locations. The so obtained formula for $Pre(m, Q)$ is then simplified to a formula which is equivalent in each reachable location of the RA. The simplification transforms it into disjunctive normal form (DNF), and then pruning disjuncts that are either infeasible, i.e., evaluating to false in each location (this can be determined by inspecting the RA for the module), or redundant, i.e., entailed by some other disjunct.

Illustration: In Step 2, we obtained the following location-specific preconditions for postcondition $\texttt{contains}(q)$ while synthesizing contracts for method `push`:

location ℓ_0 (empty list): $\{(\neg\texttt{isFull}() \wedge \texttt{isEmpty}()) \rightarrow (p = q)\}$
location ℓ_n (full list): $\{(\texttt{isFull}() \wedge \neg\texttt{isEmpty}()) \rightarrow \texttt{contains}(q)\}$
other locations ℓ_i : $\{(\neg\texttt{isFull}() \wedge \neg\texttt{isEmpty}()) \rightarrow ((p = q) \vee \texttt{contains}(q))\}$

Then, taking the conjunction of the above preconditions and applying the simplification techniques described in Step 3, we obtain the global precondition as follows:

$$((p = q) \wedge \neg\texttt{isFull}()) \quad \vee \quad \texttt{contains}(q)$$

which is the sought precondition for our final contract. □

4.3 Correctness and Optimality

In this section, we state and prove that our technique generates valid contracts (Theorem 1) which, under some conditions, are also maximal (Theorem 2).

Theorem 1 (Contract Validity). *If our method synthesizes a contract of form $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$ for an RA $\mathcal{A}$, then this contract is valid for $\mathcal{A}$.*

Proof: The theorem follows by observing that the steps our methods produce results with the desired properties:

Step 1: For each location l, the generated weakest precondition $wp_l(m, Q)$ has the property to guarantee that a method call of form $m(p_1, \ldots, p_n)$ in location l is guaranteed to terminate and result in a state where Q evaluates to true. This follows by standard techniques for computing weakest preconditions.

Step 2: For each location l, the location-specific precondition $Pre_l(m, Q)$ generated in Step 2 has the property that $[\![Pre_l(m, Q)]\!]_l \rightarrow wp_l(m, Q)$. This follows from the observation that $[\![C_i]\!]_l \rightarrow wp_l(m, Q)$ for each disjunct C_i in $[\![Pre_l(m, Q)]\!]_l$.

Step 3: Since $Pre_l(m, Q)$ is a conjunct of $Pre(m, Q)$, it follows that $Pre(m, Q)$ entails $Pre_l(m, Q)$ for any l, hence $[\![Pre(m, Q)]\!]_l \rightarrow [\![Pre_l(m, Q)]\!]_l$. Thus, if $[\![Pre(m, Q)]\!]_l$ is true in l, then a method call of form $m(p_1, \ldots, p_n)$ in l is guaranteed to terminate and result in a state where Q evaluates to true. Since l is arbitrary, the theorem follows. □

We say that an RA is *fully reachable* if for each location l and valuation μ of the registers $\mathcal{X}(l)$ of l, the state $\langle l, \mu \rangle$ is reachable.

Theorem 2 (Synthesis of Maximal Contracts). *Let $\mathcal{A}$ be a fully reachable RA, let m be a method, let the condition Q and set of variables $\mathcal{V}_{contr}$ be the input to our contract generation. If the condition R is such that its parameters are in $\mathcal{V}_{contr}$ and the contract $\{R\}\ m(p_1, \ldots, p_n)\ \{Q\}$ is valid for $\mathcal{A}$, then our method synthesizes a contract of form $\{P\}\ m(p_1, \ldots, p_n)\ \{Q\}$ such that $R \Rightarrow P$.*

Proof: Let R be a condition as above. Put R in DNF. Assume that $c_1 \wedge \cdots \wedge c_k$ is a disjunct of R. Consider a location l of $\mathcal{A}$. Since $\{R\}\ m(p_1, \ldots, p_n)\ \{Q\}$ is valid for $\mathcal{A}$, the corresponding condition $[\![c_1]\!]_l \wedge \cdots \wedge [\![c_k]\!]_l$ guarantees that calling $m(p_1, \ldots, p_n)$ in l is guaranteed to terminate and result in a state in which Q holds. Since $wp_l(m(p_1, \ldots, p_n), Q)$ is the weakest formula with such a property, it follows that $[\![c_1]\!]_l \wedge \cdots \wedge [\![c_k]\!]_l$ implies $wp_l(m(p_1, \ldots, p_n), Q)$. If none of $[\![c_1]\!]_l$, $\ldots$, $[\![c_k]\!]_l$ is *False*, our MUS generation will then find a subset of $[\![c_1]\!]_l, \ldots, [\![c_k]\!]_l$ which implies $wp_l(m(p_1, \ldots, p_n), Q)$, and generate the conjunction of the corresponding subset of $c_1, \ldots, c_k$ as a disjunct of $Pre_l(m, Q)$. If some $[\![c_i]\!]_l$ is *False*, then c_i will be added as a disjunct of $Pre_l(m, Q)$. In both cases, the result is that $Pre_l(m, Q)$ is entailed by R. Since P is obtained as the conjunction of the different $Pre_l(m, Q)$ for $l \in L$, this implies that also P is entailed by R. $\square$

The condition that $\mathcal{A}$ be fully reachable in Theorem 2 shows that our technique may generate unnecessarily strong preconditions if some states are not reachable in $\mathcal{A}$. This deficiency can be addressed by adding a procedure for generating invariant, which for each location l generates a characterization Inv_l of the valuations μ such that $\langle l, \mu \rangle$ is reachable. The formulas Inv_l are then used in Step 2, but generating minimal disjuncts $c_1 \wedge \cdots \wedge c_k$ such that

$$(c_1 \wedge \cdots \wedge c_k \wedge Inv_l) \rightarrow wp_l(m, Q)$$

is valid. We leave this extension as future work.

5 Implementation

We implement the strategies outlined in Section 4 in a Python tool called `CoGent`, abbreviation of **Co**ntract **Gen**era**t**or. We build `CoGent` in integration with z3 SAT/SMT solver [10] for checking SAT/UNSAT of logical entailments and identifying minimal unsatisfiable subsets. For this purpose, we use z3 Python library, `z3py` [12], as the constraint solver. In addition, we have used the Python library `Sympy` [24] for simplification of Boolean expressions to conversion to DNF.

In our work, we first learn the RA model of the target API using the tool `RAlib` [8, 9]. `RAlib` utilizes a given test harness tailored to the target API in order to learn the automaton. The test harness maps each method from the API to a symbol for learning the model. Next, we operate `CoGent` by giving inputs an XML representation of the automaton model and the target mutator for which we are interested in synthesizing contracts. The tool automatically identifies the observers (following the observer semantics) present in the API and generates pre and postconditions for the mutator. These conditions are quantifier-free first-order logic expressed in terms of Boolean valuations of observers and relation between input parameters. Thus the tools `RAlib` and `CoGent` in combine offer a comprehensive solution to synthesizing contracts for the mutators from an API.

Fig. 6 shows the architecture of our tool where each step described in this paper is represented as a Python module (depicted as a box). The module `Driver` runs the contract synthesis engine by operating modules for performing steps

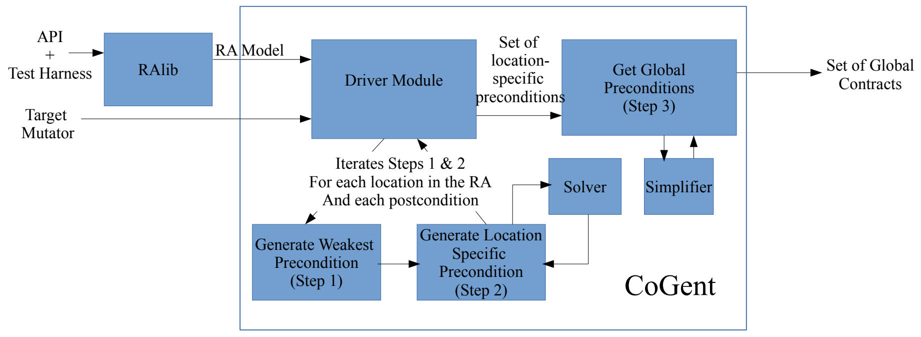
API + Test Harness
RAlib
RA Model
Target Mutator
Driver Module
Set of location-specific preconditions
Get Global Preconditions (Step 3)
Set of Global Contracts
Iterates Steps 1 & 2 For each location in the RA And each postcondition
Solver
Simplifier
Generate Weakest Precondition (Step 1)
Generate Location Specific Precondition (Step 2)
CoGent

Fig. 6: Architecture of Contract Synthesis Engine

1 and 2 in rounds for every location in the automaton and each possible postcondition. While in step 2, the module `Solver` utilizes `z3py` API for checking SAT/UNSAT and eventually deriving MUSes that yield the set of preconditions. Once the module `Driver` accumulates all location-specific preconditions for a mutator, it delegates the task of synthesizing global contracts to a module that merges the preconditions wrt. each postcondition. Additionally, it simplifies the merged contract with the help of `Simplifier`, which inherits some of the functions provided by `Sympy`.

6 Evaluation

We evaluate our contract generation tool by synthesizing contracts for some of the modules from Java SEv8, and the Contiki-NG OS. We generate contracts for the mutators from those modules using the supplied Boolean observers including `isFull` and `isEmpty` methods for handling size bounds. The maximum size for each data structure is set to 3. Table 1 outlines the details of our tool evaluation. For each module, the number of non-whitespace, non-comment lines of code is mentioned within brackets. The average running time (in seconds) for model learning and synthesizing contract for a mutator are recorded in columns 4 and 5, respectively, using `RAlib` and `CoGent` tools. In `RAlib`, the maximum number of attempts to find counterexamples is set to 1000 per hypothesis. Column 6 shows the number of locations in the automaton, and column 7 indicates the total number of contracts generated by the tool. The final column specifies maximum number of disjuncts obtained after simplifying the preconditions for the contracts generated by our tool for each module. Following we illustrate a few contracts generated for two mutators from Contiki-NG list module.

Contiki-NG, a widely used open-source OS for IoT, includes a critical list module, which has unique characteristics compared to typical list implementations. This module is designed to be highly resource-constrained, where the API allocates a memory block by releasing it if it has been pre-allocated. Addition-

Table 1: Interfaces for Evaluating Our Approach to Synthesizing Contract

Modules	**Mutators**	**Observers**	**Runtime(s)**		**#**	**#**	**max.**
			`RAlib`	`CoGent`	**locs.**	**cont.**	**disj.**
Contiki-NG List (45)	`insert`(e_1, e_2), `pop`(),`push`(e_1), `add`(e), `remove`(e)	`contains`(e), `isFull`(), `isEmpty`()	51.2	21.02	52	19	3
HashMap (1916)	`put`(k, v), `remove`(k)	`containsKey`(k), `containsValue`(v), `isEmpty`(), `isFull`()	3.33	12.07	15	14	7
Stack (93)	`push`(e), `pop`()	`isEmpty`(), `isFull`(), `contains`()	1.6	0.6	21	8	3
PriorityQueue (704)	`add`(e), `remove`(e), `poll`()	`isEmpty`(), `isFull`(), `contains`()	5.4	11.7	53	21	5
BoundedList (43)	`push`(e), `pop`(), `insert`(e_1, e_2)	`isEmpty`(), `isFull`(), `contains`()	15.74	0.97	21	12	3

ally, the list can function as both a stack and a queue, but storing a block in either way requires removing it first if it already exists in the list.

To evaluate this module, we create a Java class that simulates the behaviour of the Contiki-NG `list` module, treating memory blocks as integer elements, and generate contracts for the mutators. In the following, we discuss the contracts generated for two specific mutators: `add` and `insert`, which establish the aforementioned behaviour. The `add` method takes an input element through p, removes it if it already exists in the list, and then appends the element at the end. On the other hand, the `insert` method receives two parameters: p_1 and p_2. It removes p_2 if it is present in the list and inserts it again after p_1.

Here are two of the contracts generated for the `add` and `insert` methods:

i $\{\texttt{isEmpty}() \lor (\texttt{contains}(p) \land \neg\texttt{isFull}())\}\ \texttt{add}(p)\ \{\neg\texttt{isFull}()\}$
ii $\{\texttt{isEmpty}() \lor (\texttt{contains}(p_2) \land \neg\texttt{isFull}())\}\ \texttt{insert}(p_1, p_2)\ \{\neg\texttt{isFull}()\}$

Contract (i) for method `add` demonstrates that adding an element that is already present in the list will not result in the list becoming full. This is because the method removes the element before adding it again. Similarly, contract (ii) shows that the list cannot become full if the parameter p_2 is already present in the list.
Contract Validation: Next, we validate the synthesized contracts for the mutators listed in Table 1 leveraging symbolic execution [17], a program verification technique that explores different execution paths to test the validity of the contracts. Symbolic execution treats inputs as symbols representing arbitrary values and systematically explores feasible code paths with symbolic input values.

To validate contract for a mutator, we generate an arbitrary pre-state that can be reached after a bounded-length sequence of calls to mutators with symbolic parameters. Symbolic execution is then performed on the targeted mutator, under the assumption correspond to the precondition from synthesized contract. The postcondition is treated as an assertion checked after symbolic execution to identify any execution paths that fail to satisfy the postcondition for certain parameter values. If the postcondition remains valid throughout symbolic execution, the contract is considered to be valid for all module states. We utilize the Symbolic(Java) PathFinder tool (SPF) [21] to facilitate contract validation. Using the above setup, we successfully validated all contracts obtained through our proposed method, confirming that none of them are invalid. For a detailed implementation for contract validation, we encourage to refer to [14].

7 Related Work

We give an overview of the most related areas of research. For a broader survey of existing contract synthesis approaches, we refer the reader to [2].

Our work can be seen as an approach to *precondition inference:* given a method m with a given postcondition Q, produce a precondition P which guarantees that Q will hold when the method returns. Data-driven approaches to this problem (e.g., [22]) start from a set of *features*, i.e., predicates over m's inputs; they collect "good" test inputs (causing Q to be satisfied) as well as "bad" test inputs (causing Q to be falsified), which induce *feature vectors* (valuations of the features) for "good" and for "bad" inputs. A classification algorithm can then be used to separate "good" from "bad" inputs, producing a precondition. Padhi et al. [20] augment this technique by the ability to learn new features, when the existing ones are not sufficient to separate "good" from "bad" inputs. Astorga et al. [4, 5] further build on this technique to be able to give guarantees relative to a given test input generator: a precondition is *safe* if the test generator cannot find a test input that satisfies the precondition and violates the postcondition; it is *maximal* if it includes all inputs found by the test generator that satisfy the postcondition. Our method is data-driven as well, as active automata learning is a black-box method and works by executing test cases. Our method differs from existing inference methods in the intermediate step of constructing a register automaton, and is, thus, able to discover which states of a system are reachable.

Molina et al. [19] use an analogous technique for generating postconditions for a given precondition, in which the method is executed with an exhaustive set of inputs, and postconditions are generated from the observed outputs using a genetic algorithm. Dynamic methods have also been used to infer program invariants. Ernst et al. [13] developed the Daikon system, which infers likely invariants by observing program executions. The obtained invariants are restricted to conjunctive Boolean expressions. The approach has later also been extended to generate likely program contracts. At the moment, it is not clear whether our method can be extended to synthesise postconditions, although this is an interesting avenue of future research.

There are also several white-box approaches to synthesize contracts. Alpuente et al. [1] apply a symbolic execution engine, which explores program paths reachable for given a precondition P. For each path, the engine produces a path condition and symbolic values of program variables, from which corresponding postconditions are synthesized. Singleton et al. [23] present an algorithm, based on symbolic execution, to extract human-readable concise contracts from strongest postconditions. Alshnakat et al. [2] use solvers for constrained Horn clauses (i.e., model checkers) to generate program contracts that are sufficient to verify given properties of a program. It remains to be investigated how our approach compares, in terms of the required runtime and readability of contracts, to white-box approaches.

8 Conclusion

We have presented a novel approach to synthesizing method contracts for stateful software modules, specifically those implementing data structures like stacks, queues, etc. Assuming that the modules are equipped with observer methods for querying the module state, and mutators for modifying it, our technique synthesizes contracts for the mutators, where pre- and postconditions are expressed as Boolean combinations of observer calls together with equalities between parameters to observers and mutators. Our proposed technique first learns a model of the module's behaviour, utilizing existing algorithms for active learning of register automata. On the basis of the learned model, our technique automatically synthesizes preconditions for any given postcondition. We prove that, under some assumptions, the obtained preconditions are the weakest possible. We have developed a tool called `CoGent` based on our approach, which generates contracts for mutators from a given register automaton where the contracts cover reachable behaviours (module locations). Our implementation provides evidence that this approach can successfully synthesize contracts for various stateful Java modules. As additional evidence, we validate obtained contracts using symbolic execution.

In future work, we plan to extend our approach to handle non-Boolean observers and inequalities between input parameters and registers during the model learning phase. This extension will enable the inference of preconditions in a more expressive language. In addition, we will enhance contract synthesis with location-specific invariant generation, to handle some cases in which invariants about registers are needed to prevent the synthesis of unnecessarily strong preconditions (see Section 4.3).

Acknowledgments

This research was partially funded by the Swedish Foundation for Strategic Research through projects aSSIsT and WebSec, the Swedish Research Council (Vetenskapsrådet), and the Knut and Alice Wallenberg Foundation through project UPDATE. We also thank the SEFM 2023 reviewers for comments and questions that have improved the presentation of our work.

References

1. M. Alpuente, D. Pardo, and A. Villanueva, "Abstract contract synthesis and verification in the symbolic K framework," *Fundam. Informaticae*, vol. 177, no. 3-4, pp. 235–273, 2020. [Online]. Available: https://doi.org/10.3233/FI-2020-1989
2. A. Alshnakat, D. Gurov, C. Lidström, and P. Rümmer, *Constraint-Based Contract Inference for Deductive Verification.* Springer International Publishing, 2020, pp. 149–176.
3. D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.
4. A. Astorga, P. Madhusudan, S. Saha, S. Wang, and T. Xie, "Learning stateful preconditions modulo a test generator," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 775–787. [Online]. Available: https://doi.org/10.1145/3314221.3314641
5. A. Astorga, S. Saha, A. Dinkins, F. Wang, P. Madhusudan, and T. Xie, "Synthesizing contracts correct modulo a test generator," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021. [Online]. Available: https://doi.org/10.1145/3485481
6. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, *ACSL: ANSI/ISO C Specification Language*, http://frama-c.com/acsl.html.
7. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *Int. J. Softw. Tools Technol. Transf.*, vol. 7, no. 3, pp. 212–232, 2005. [Online]. Available: https://doi.org/10.1007/s10009-004-0167-4
8. S. Cassel, F. Howar, and B. Jonsson, "RALib: A LearnLib extension for inferring EFSMs," in *DIFTS 2015*, 2015. [Online]. Available: https://www.faculty.ece.vt.edu/chaowang/difts2015/papers/paper_5.pdf
9. S. Cassel, F. Howar, B. Jonsson, and B. Steffen, "Active learning for extended finite state machines," *Formal Asp. Comput.*, vol. 28, no. 2, pp. 233–263, 2016.
10. L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
11. E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT Numerical Mathematics*, vol. 8, no. 3, pp. 174–186, 1968. [Online]. Available: https://doi.org/10.1007/BF01933419
12. A. Dutcher and N. Bjorner, "z3-solver 4.12.2.0," https://pypi.org/project/z3-solver/, 2023.
13. 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," vol. 69, no. 1-3, pp. 35–45, 2007.
14. S. Ghosal, B. Jonsson, and P. Rümmer, "An active learning approach to synthesizing program contracts," Jul. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8169860
15. M. Isberner, F. Howar, and B. Steffen, "The open-source LearnLib - A framework for active automata learning," in *Proc. CAV 2015*, ser. LNCS, vol. 9206. Springer, 2015, pp. 487–495.

16. ——, “The TTT algorithm: A redundancy-free approach to active automata learning,” in *Runtime Verification: 5th International Conference, RV 2014, Proceedings*, ser. LNCS, vol. 8734. Springer, Sep. 2014, pp. 307–322. [Online]. Available: https://doi.org/10.1007/978-3-319-11164-3_26
17. J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: https://doi.org/10.1145/360248.360252
18. B. Meyer, “Applying ”design by contract”,” *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992. [Online]. Available: https://doi.org/10.1109/2.161279
19. F. Molina, P. Ponzio, N. Aguirre, and M. F. Frias, “Evospex: An evolutionary algorithm for learning postconditions,” in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 2021, pp. 1223–1235. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00112
20. S. Padhi, R. Sharma, and T. D. Millstein, “Data-driven precondition inference with learned features,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds. ACM, 2016, pp. 42–56. [Online]. Available: https://doi.org/10.1145/2908080.2908099
21. C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, “Combining unit-level symbolic execution and system-level concrete execution for testing nasa software,” in *Proceedings of the 2008 International Symposium on Software Testing and Analysis ISSTA*, 2008, pp. 15–26. [Online]. Available: https://doi.org/10.1145/1390630.1390635
22. S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta, “Dynamic inference of likely data preconditions over predicates by tree learning,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, B. G. Ryder and A. Zeller, Eds. ACM, 2008, pp. 295–306. [Online]. Available: https://doi.org/10.1145/1390630.1390666
23. J. L. Singleton, G. T. Leavens, H. Rajan, and D. R. Cok, “Inferring concise specifications of APIs,” *CoRR*, vol. abs/1905.06847, 2019. [Online]. Available: http://arxiv.org/abs/1905.06847
24. S. D. Team, “Sympy 1.12,” https://www.sympy.org/en/index.html, 2023.