Executing Specifications using Synthesis and Constraint Solving

Viktor Kuncak*[1], Etienne Kneuss[1], and Philippe Suter[1,2]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
{firstname.lastname}@epfl.ch, psuter@us.ibm.com

Abstract. Specifications are key to improving software reliability as well as documenting precisely the intended behavior of software. Writing specifications is still perceived as expensive. Of course, writing implementations is at least as expensive, but is hardly questioned because there is currently no real alternative. Our goal is to give specifications a more balanced role compared to implementations, enabling the developers to compile, execute, optimize, and verify against each other mixed code fragments containing both specifications and implementations. To make specification constructs executable we combine deductive synthesis with run-time constraint solving, in both cases leveraging modern SMT solvers. Our tool decomposes specifications into simpler fragments using a cost-driven deductive synthesis framework. It compiles as many fragments as possible into conventional functional code; it executes the remaining fragments by invoking our constraint solver that extends an SMT solver to handle recursive functions. Using this approach we were able to execute constraints that describe the desired properties of integers, sets, maps and algebraic data types.

1 Introduction

Specifications are currently second class citizens in software development. An implementation is obligatory; specification is optional. Our goal is to assign to specifications a more balanced role compared to implementations. For this to happen, we aim to allow developers to execute specifications, even if such execution is slower or less predictable than execution of imperative and functional code. We wish to permit developers to write mixed code fragments containing both specifications and implementations. They should be able to compile, execute, optimize, and verify such fragments against each other.

By *execution* of specifications we mean not only testing whether a constraint is true for known values of variables (as when checking e.g. assertions), but also computing a missing value so that the given constraint is satisfied. Such constraint solving functionality can also be thought as one way of automating

* This work is supported in part by the European Research Council (ERC) Project *Implicit Programming*

the remedial action in case of assertion violation [57]. However, we believe that such constructs should not be treated as a sort of exception mechanism, but as one of the main ways of describing the desired common behavior.

This paper presents our experience in developing techniques to make such constraint solving executable. Our current approach combines deductive synthesis with run-time constraint solving, in both cases leveraging modern SMT solvers. We have built a tool as part of the Leon verification system [9] that incorporates both techniques and allows us to experiment with their trade-offs. A version of the Leon platform is publicly available in source code form for further experiments at `http://lara.epfl.ch/w/leon`. The tool decomposes specifications into simpler fragments using a cost-driven deductive synthesis framework [32,37,40–42]. It compiles as many fragments as possible into conventional functional code; it executes the remaining fragments by invoking a constraint solver at runtime. The solver extends a conventional SMT solver with the ability to handle recursive functions, in a manner similar to our previous systems [38,39]. Using this approach we were able to execute constraints that describe the desired properties of integers, sets, maps and algebraic data types.

In general, the deductive synthesis framework allows us to recursively split challenging problems into tractable subproblems and compile some of the subproblems into conventional code. If a subproblem remains too challenging for synthesis, we keep its declarative specification and execute it using run-time constraint solving. It turns out that in certain interesting cases, the resulting partial program is well-defined for simple, frequent paths and only relies on run-time constraint solving for complex cases.

In the rest of this paper we outline our approach, including the functional language setup, the run-time constraint solving approach, and synthesis techniques. We then illustrate our initial experience with combining run-time constraint solving and synthesis, hinting at some future directions. We finish with a (necessary biased) survey of related work.

2 Examples

We illustrate the benefits of enabling declarative specifications through a series of examples. We show how these examples can be effectively handled by our system. We start by defining a List data-structure with an abstraction function content from list to a set, and an invariant predicate isSorted.

```
abstract class List
case class Cons(head: Int, tail: List) extends List
case object Nil extends List

def content(l: List): Set[Int] = l match {
  case Cons(h, t) ⇒ Set(h) ++ content(t)
  case Nil ⇒ Set()
}
```

```
def isSorted(l: List): Boolean = l match {
  case Cons(h1, t1 @ Cons(h2, t2)) ⇒ (h1 ≤ h2) && isSorted(t1)
  case _ ⇒ true
}
```

Thanks to the abstraction function and the invariant, we can concisely specify an insert operation for sorted lists using a constraint:

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  choose{ (x: List) ⇒ isSorted(x) && (content(x) == content(l) ++ Set(v)) }
}
```

Our deductive synthesis procedure is able to translate this constraint into the following complete implementation in under 9 seconds:

```
def insert(l: List, v: Int) = {
  require(isSorted(l))
  l match {
    case Cons(head, tail) ⇒
      if (v == head) {
        l
      } elseif (v < head) {
        Cons(v, l)
      } else {
        insert(t, v)
      }
    case Nil ⇒
      Cons(v, Nil)
  }
}
```

However, as the complexity of the constraints increases, the deductive procedure may run short of available time to translate a constraint into complete efficient implementations. As an example, we can currently observe this limitation of our system for a red-black tree benchmark. The following method describes the insertion into a red-black tree.[3]

```
def insert(t: Tree, v: Int) = {
  require(isRedBlack(t))
  choose{ (x: Tree) ⇒ isRedBlack(x) && (content(x) == content(t) ++ Set(v)) }
}
```

Instead of using synthesis (for which this example may present a challenge), we can rely on the run-time constraint solving to execute the constraint. In such scenario, the run-time waits until the argument `t` and the value `v` are known, and finds a new tree value `x` such that the constraint holds. Thanks to our constraint solver, which has a support recursive functions and also leverages the Z3 SMT solver, this approach works well for small red-black trees. It is

[3] We omit here the definition of the tree invariant for brevity, which is rather complex [15, 52], but still rather natural to describe using recursive functions.

therefore extremely useful for prototyping and testing and we have previously explored it as a stand-alone technique for constraint programming in Scala [39]. However, the complexity of reasoning symbolically about complex trees makes this approach inadequate for large concrete inputs.

Fortunately, thanks to the nature of our deductive synthesis framework, we can combine synthesis and run-time constraint solving. We illustrate this using an example of a *red-black tree with a cache*. Such a tree contains a red-black tree, but also redundantly stores one of its elements.

```
case class CTree(cache: Int, data: Tree)
```

The specification of the invariant inv formalizes the desired property: the cache value must be contained in the tree unless the tree is empty.

```
def inv(ct: CTree) = {
  isRedBlack(ct.data) &&
  (ct.cache ∈ content(ct.data)) || (ct.data == Empty)
}
```

The contains operation tests membership in the tree.

```
def contains(ct: CTree, v: Int): Boolean = {
  require(inv(ct))
  choose{ (x: Boolean) ⇒ x == (v ∈ content(ct)) }
}
```

While not being able to fully translate it, the deductive synthesis procedure decomposes the problem and partially synthesizes the constraint. One of its possible results is the following *partial* implementation that combines actual code and a sub-constraint:

```
def contains(ct: CTree, v: Int): Boolean = ct.data match {
  case n: Node ⇒
    if (ct.cache == v) {
      true
    } else {
      choose { (x: Boolean) ⇒ x == (v ∈ content(n)) }
    }
  case Empty ⇒
    false
}
```

We notice that this partial implementation makes use of the cache in accordance with the invariant. The code accurately reflects the fact that the cache may not be trusted if the tree is empty. The remaining constraint is in fact a simpler problem that only relates to standard red-black trees. Our system can then compile the resulting code, where the fast path is compiled as the usual Scala code, and the choose construct is compiled using the run-time solving approach. In the sequel we give details both for our run-time solving approach and the compile-time deductive synthesis transformation framework. We then discuss our very first experience with combining these two approaches.

3 Language

We next present a simple functional language that we use to explore the ability to do verification as well as to execute and compile constraints.

3.1 Implementation Language

As the implementation language we consider a Turing-complete Scala fragment. For the purpose of this paper we assume that programs consist of a set of side-effect-free deterministic mutually recursive functions that manipulate countable data types including integers, n-tuples, algebraic data types, finite sets, and finite maps. We focus on functional code. Our implementation does support localized imperative features; for more details see [9]. We assume that recursive functions are terminating when their specified preconditions are met; our tool applies several techniques to establish termination of recursive functions.

3.2 Function Contracts

Following Scala's contract notation [51], we specify functions in the implementation language using preconditions (**require**) and postconditions (**ensuring**). The declaration

```
def f(x:A) : B = {
  body
} ensuring((res:B) ⇒ post(res))
```

indicates that the result computed by f should satisfy the specification post. Here res⇒post(res) is a lambda expression in which res is a bound variable; the **ensuring** operator binds res to the result of evaluating body. The expression post is itself a general expression in the implementation language, and can invoke recursive functions itself.

3.3 Key Concept: Constraints

Constraints are lambda expressions returning a Boolean value, precisely of the kind used after **ensuring** clauses. To express that a constraint should be solved for a given value, we introduce the construct choose. The expression

```
choose((res:B) ⇒ C(res))
```

should evaluate to a value of type B that satisfies the constraint C. For example, an implicit way to indicate that we expect that the value y is even and to compute y/2 is the following:

```
choose((res:Int) ⇒ res + res == y)
```

The above expression evaluates to y/2 whenever y is even. Note that C typically contains, in addition to the variable res, variables denoting other values in scope (in the above, example, the variable y). We call such variables *parameters* of the constraint.

4 Solving Constraints at Run-Time

We next describe the baseline approach that we use to execute constraints at run-time. This approach is general, as it works for essentially all computable functions on countable domains. On the other hand, it can be inefficient. The subsequent section will describe our synthesis techniques, which can replace such general-purpose constraint solving in a number of cases of interest.

4.1 Model-Generating SMT Solver

The main work horse of our run-time approach is an SMT solver, concretely, Z3 [16]. What is crucial for our application is that Z3 supports model generation: it not only detects unsatisfiable formulas, but in case a formula has a model, can compute and return one model. Other important aspects of Z3 are that it has good performance, supports algebraic data types and arrays [17], supports incremental solving, and has a good API, which we used to build a Scala layer to conveniently access its functionality [38].

4.2 Fair Unfolding of Recursive Functions

Although SMT solvers are very expressive, they do not directly support recursive functions. We therefore developed our own procedure for handling recursive function definitions. Given a deterministic recursive functions f viewed as a relation, assume that f is defined using the fixed point of a higher-order functional H, which implies the formula D:

$$D \quad \equiv \quad \forall x.\ f(x) = H(x, f)$$

The constraints we solve have the form $C \wedge D_k$, where both C and D_k are quantifier-free formulas that we map precisely into the language of an SMT solver.

We use an algorithm for fair unfolding of recursive definitions [69] to reduce the formula $C \wedge D$ to a series of over-approximations and under-approximations. From an execution point of view, such approximations describe all the executions up to certain depth. From a logical perspective, unfolding is a particular form of universal quantifier instantiation which generates a ground consequence D_k of the definition D. If $C \wedge D_k$ is unsatisfiable, so is $C \wedge D$. If $C \wedge D_k$ is satisfiable for the model $x = a$ then we can simply check whether the executable expression evaluates to **true**. In our implementation we have an additional option: we can use the SMT solver itself, to check whether a model of $C \wedge D_k$ depends on the values of partly interpreted functions denoted by f. To this extent, we instrument the logical representation of unfolding up to a given depth using propositional variables that can prevent the execution from depending on uninterpreted values of functions. We call the value of these variables the *control literals* B.

Figure 1 shows the pseudo-code of the resulting algorithm for solving constraints involving recursive functions. It is defined in terms of two subroutines,

```
def solve(C, D) {
  (C, D_0, B_0) = unrollStep(C, D, ∅, 0)
  k = 0
  while(true) {
    decide(C ∧ D_k ∧ ⋀_{b∈B_k} b) match {
      case "SAT" ⇒ return "SAT"
      case "UNSAT" ⇒ decide(C ∧ D_k) match {
        case "UNSAT" ⇒ return "UNSAT"
        case "SAT" ⇒ (C, D_{k+1}, B_{k+1}) = unrollStep(C, D, B_k, k) }}
    k += 1
  }
}
```

Fig. 1. Pseudo-code of the solving algorithm. The decision procedure for the base theory is invoked through the calls to decide.

decide, which invokes the underlying SMT solver, and unrollStep. The fair nature of the unrolling step guarantees that all uninterpreted function values present in the formula are eventually unfolded, if needs be.

The formula without the control literals can be seen as an *over-approximation* of the formula with the semantics of the program, in that it accepts all the same models plus some models in which the interpretation of some invocations is incorrect. The formula with the control literals is an *under-approximation*, in the sense that it accepts only the models that do not rely on the guarded invocations. This explains why the UNSAT answer can be trusted in the first case and the SAT case in the latter.

This algorithm is the basis of the original Leon as a constraint solver and verifier for functional programs [9, 68, 69].

4.3 Executing Choose Construct at Run-Time

During the compilation of programs with choose constructs, we collect a symbolic representation of the constraints used. The actual call to choose is then substituted with a call back to the Leon system indicating both which constraint it refers to, but also propagating the run-time inputs.

During execution, these inputs are converted from concrete JVM objects back to their Leon representations, and finally substituted within the constraint. By construction, the resulting formula's free variables are all output variables.

Given a model for this formula, we translate the Leon representation of output values back to concrete objects which are then returned.

4.4 Evaluation

We evaluated the performance of the run-time constraint solving algorithm on two data structures with invariants: sorted lists and red-black trees. For each

data structure, we implemented a declarative version of the *add* and *remove* operations. Thanks to the abstraction and predicate functions, the specifications of both operations are very concise and self-explanatory. We illustrate this by providing the corresponding code for red-black trees:

```
def add(t: Tree, e: Int): Tree = choose {
  (res: Tree) ⇒ content(res) == content(t) ++ Set(e) && isRedBlackTree(res)
}

def remove(t: Tree, e: Int): Tree = choose {
  (res: Tree) ⇒ content(res) == content(t) -- Set(e) && isRedBlackTree(res)
}
```

Solving is relatively efficient for small data structures: it finds models in under 400ms for lists and trees up to size 4. However, the necessary solving time increases exponentially with the size and goes as high as 35 seconds for synthesizing insertion into a red-black tree of size 10.

5 Synthesizing Functional Code from Constraints

In this section, we give an overview of our framework for deductive synthesis. The goal of the approach is to derive correct programs by successive steps. Each step is validated independently, and the framework ensures that composing steps results in global correctness.

5.1 Synthesis Problems and Solutions

A synthesis problem, or constraint, is fundamentally a relation between inputs and outputs. We represent this, together with contextual information, as a quadruple

$$[\![\bar{a} \;\; \langle \Pi \rhd \phi \rangle \;\; \bar{x}]\!]$$

where:

- $\bar{a}$ denotes the set of *input variables*,
- $\bar{x}$ denotes the set of *output variables*,
- ϕ is the *synthesis predicate*, and
- Π is the *path condition* to the synthesis problem.

The free variables of ϕ must be a subset of $\bar{a} \cup \bar{x}$. The path condition denotes a property that holds for input at the program point where synthesis is to be performed, and the free variables of Π should therefore be a subset of $\bar{a}$.

As an example, consider the following call to choose:

```
def f(a : Int) : Int = {
  if(a ≥ 0) {
    choose((x : Int) ⇒ x ≥ 0 && a + x ≤ 5)
  } else ...
}
```

The representation of the corresponding synthesis problem is:

$$[\![a \ \langle a \geq 0 \rhd x \geq 0 \wedge a + x \leq 5 \rangle \ x]\!] \tag{1}$$

We represent a solution to a synthesis problem as a pair $\langle P | \bar{T} \rangle$ where:

- P is the *precondition*, and
- $\bar{T}$ is the *program term*.

The free variables of both P and $\bar{T}$ must range over $\bar{a}$. The intuition is that, whenever the path condition and the precondition are satisfied, evaluating $\phi[\bar{x} \mapsto \bar{T}]$ should evaluate to $\top$ (true), i.e. $\bar{T}$ are realizers for a solution to $\bar{x}$ in ϕ given the inputs $\bar{a}$. Furthermore, for a solution to be as general as possible, the precondition must be as weak as possible.

Formally, for such a pair to be a solution to a synthesis problem, denoted as

$$[\![\bar{a} \ \langle \Pi \rhd \phi \rangle \ \bar{x}]\!] \vdash \langle P | \bar{T} \rangle$$

the following two properties must hold:

- *Relation refinement:* $\Pi \wedge P \models \phi[\bar{x} \mapsto \bar{T}]$
  This property states that whenever the path- and precondition hold, the program $\bar{T}$ can be used to generate values for the output variables $\bar{x}$ such that the predicate ϕ is satisfied.
- *Domain preservation:* $\Pi \wedge (\exists \bar{x} : \ \phi) \models P$
  This property states that the precondition P cannot exclude inputs for which an output exists.

As an example, a valid solution to the synthesis problem (1) is given by: $\langle a \leq 5 | 0 \rangle$. The precondition $a \leq 5$ characterizes exactly the input values for which a solution exists, and for all such values, the constant 0 is a valid solution term for x. The solution is in general not unique; alternative solutions for this particular problem include $\langle a \leq 5 | 5 - a \rangle$, or $\langle a \leq 5 | \mathsf{if}(a < 5) \ a + 1 \ \mathsf{else} \ 0 \rangle$.

5.2 Inference Rules

The correctness conditions described above characterize the validity of solutions to synthesis problems. We now show how to derive such solutions. We present our techniques as a set of *inference rules*. As a simple first example, consider the following rule:

$$\frac{[\![\bar{a} \ \langle \Pi \rhd \phi[x_0 \mapsto t] \rangle \ \bar{x}]\!] \vdash \langle P | \bar{T} \rangle \qquad x_0 \notin \mathrm{vars}(t)}{[\![\bar{a} \ \langle \Pi \rhd x_0 = t \wedge \phi \rangle \ x_0 \, , \bar{x}]\!] \vdash \langle P | \mathsf{val} \ \bar{x} := \bar{T}; \ (t \, , \bar{x}) \rangle}$$

As is usual with inference rules, on top are the premisses and below is the goal. This particular rule captures the intuition that, whenever a term of the form $x_0 = t$ appears as a top-level conjunct in a synthesis problem, the problem can be simplified by assigning to the output variable x_0 the term t. The rule specifies

both how the subproblem relates to the original one, and how its solution and precondition are used in the final program.

Another example is the following rule for decomposing disjunctions:

$$\frac{[\![\bar{a}\ \ \langle \Pi \rhd \phi_1 \rangle\ \ \bar{x}]\!] \vdash \langle P_1 | \bar{T}_1 \rangle \qquad [\![\bar{a}\ \ \langle \Pi \wedge \neg P_1 \rhd \phi_2 \rangle\ \ \bar{x}]\!] \vdash \langle P_2 | \bar{T}_2 \rangle}{[\![\bar{a}\ \ \langle \Pi \rhd \phi_1 \vee \phi_2 \rangle\ \ \bar{x}]\!] \vdash \langle P_1 \vee P_2 | \mathsf{if}(P_1)\ \{\bar{T}_1\}\ \mathsf{else}\ \{\bar{T}_2\} \rangle}$$

Here, the rule states that a disjunction can be handled by considering each disjunct in isolation, and combining the solutions as an if-then-else expression, where the branching condition is the precondition for the first problem. Note that in the second subproblem, we have added the literal $\neg P_1$ to the path condition. This reflects the knowledge than, in the final program, the subprogram for the second disjunct only executes if the first one cannot compute a solution.

In general, a synthesis problem is solved whenever a derivation can be found for which all output variables are assigned to a program term.

For certain well-defined classes of synthesis problems, we can design sets of inference rules which, together with a systematic application strategy, are guaranteed to result in successful derivation. We have shown in previous work such complete strategies for integer linear arithmetic, rational arithmetic, or term algebras [32, 40, 67]. We call these *synthesis procedures*, analogously to decision procedures.

As a final example, we now show how our framework can express solutions that take the form of recursive functions traversing data structures. The next rule captures one particular yet very common form of such a traversal for Lists.

$$\frac{\begin{array}{c}(\Pi_1 \wedge P) \implies \Pi_2 \qquad \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \implies \Pi_2[a_0 \mapsto t] \\ [\![\bar{a}\ \ \langle \Pi_2 \rhd \phi[a_0 \mapsto \mathsf{Nil}] \rangle\ \ \bar{x}]\!] \vdash \langle \top | \bar{T}_1 \rangle \\ [\![\bar{r}\,, h\,, t\,, \bar{a}\ \ \langle \Pi_2[a_0 \mapsto \mathsf{Cons}(h,t)] \wedge \phi[a_0 \mapsto t, \bar{x} \mapsto \bar{r}] \rhd \phi[a_0 \mapsto \mathsf{Cons}(h,t)] \rangle\ \ \bar{x}]\!] \vdash \langle \top | \bar{T}_2 \rangle\end{array}}{[\![a_0\,, \bar{a}\ \ \langle \Pi_1 \rhd \phi \rangle\ \ \bar{x}]\!] \vdash \langle P | \mathsf{rec}(a_0, \bar{a}) \rangle}$$

The goal of the rule is to derive a solution consisting of a single invocation of a (fresh) recursive function rec, of the following form:

```
def rec(a0, ā) = {
  require(Π2)
  a0 match {
    case Nil ⇒ T̄1
    case Cons(h, t) ⇒
      val r̄ = rec(t, ā)
      T̄2
  }
} ensuring(r̄ ⇒ φ[x̄ ↦ r̄])
```

The rule decomposes the problem into two cases, corresponding to the alternatives in the data type, and assumes that the solution takes the form of a *fold* function, fixing the recursive call.

6 Combining Synthesis and Runtime Constraint Solving

The deductive synthesis framework allows us to split a challenging problem into tractable sub problems. In the case where the subproblems remain too challenging, we keep their corresponding declarative specifications and compile them into the run-time invocation of the constraint. This result in a partially implemented program. In certain cases, the partial program is well-defined for simple, frequent paths, and only falls back to run-time solving for complex cases.

As an illustrative example, we give here the partial derivation of the function contains on CTrees from Section 2 using rules such as the ones described in Section 5.

We start with the synthesis problem:

$$[\![\mathsf{c}, \mathsf{d}, \mathsf{v} \;\; \langle \mathsf{inv}(\mathsf{CTree}(\mathsf{c}, \mathsf{d})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c}, \mathsf{d}))\rangle \;\; \mathsf{x}]\!]$$

A first step is to perform case analysis on d, the tree. This generates two subproblems, for the cases Empty and Node respectively. For Empty, we have:

$$[\![\mathsf{c}, \mathsf{v} \;\; \langle \mathsf{inv}(\mathsf{CTree}(\mathsf{c}, \mathsf{Empty})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c}, \mathsf{Empty}))\rangle \;\; \mathsf{x}]\!]$$

At this point, inv(CTree(c, Empty)) simplifies to $\top$ and content(CTree(c, Empty)) simplifies to $\emptyset$. The problem thus becomes:

$$[\![\mathsf{c}, \mathsf{v} \;\; \langle \top \rhd \mathsf{x} \iff \mathsf{v} \in \emptyset\rangle \;\; \mathsf{x}]\!]$$

which is solved by $\langle \top | \mathsf{false}\rangle$. For the Node branch, we have:

$$[\![\mathsf{c}, \mathsf{n}, \mathsf{v} \;\; \langle \mathsf{n} \neq \mathsf{Empty} \wedge \mathsf{inv}(\mathsf{CTree}(\mathsf{c}, \mathsf{n})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{c}, \mathsf{n}))\rangle \;\; \mathsf{x}]\!]$$

This is almost the original problem, with the additional contextual information that the tree is not empty. Given that we have two integer variables in scope, c and v (the cache and the value for which we are checking inclusion), a potential tactic is to perform case analysis on their equality. This yields two subproblems. For the equal case, we have one fewer variable:

$$[\![\mathsf{n}, \mathsf{v} \;\; \langle \mathsf{n} \neq \mathsf{Empty} \wedge \mathsf{inv}(\mathsf{CTree}(\mathsf{v}, \mathsf{n})) \rhd \mathsf{x} \iff \mathsf{v} \in \mathsf{content}(\mathsf{CTree}(\mathsf{v}, \mathsf{n}))\rangle \;\; \mathsf{x}]\!]$$

At this point, because inv(CTree(v,n)) implies that v∈CTree(v,n), the problem is solved with $\langle \top | \mathsf{true}\rangle$.

For the final subproblem, where d is a Node and the cache does not hold the value v, our system is currently not efficient enough to derive a solution. Therefore, it falls back to emitting a run-time invocation of choose. Combining the solutions for all subproblems, we obtain the partially synthesized function contains as shown in Section 2.

6.1 Discussion

It is our hope that the combination of two technologies, run-time constraint solving and synthesis, can make execution of specifications practical. It will be then

interesting to understand to what extend such complete specifications change the software development process. Writing data structures using constraints shows the productivity advantages of using constraints, because data structure invariants are reusable across all operations, whereas the remaining specification of each individual operation becomes extremely concise. The development process thus approaches the description of a data structure design from a textbook [15], which starts from basic invariants and uses them as guiding principle when presenting the implementation of data structure operations. We are thus hopeful that, among other results, we can soon enable automated generation of efficient unbounded data structures from high-level descriptions, analogously to recent breakthrough on cache coherence protocol generation [71].

An exciting future direction is to use run-time property verification techniques to efficiently combine code generated through speculative synthesis and constraint solving. In many synthesis approaches, due to a heavy use of example-driven techniques and the limitations of static verification, it is also quite possible that the automatically generated implementation is incorrect for some of the inputs. In such cases, techniques of run-time verification can be used to detect, with little overhead, the violation of specifications for given inputs. As a result, synthesis could be used to generate fast paths and likely code fragments, while ensuring the overall adherence to specification at run time.

In general, we are excited about future interplay between dynamic and static approaches to make specifications executable, which is related to partial evaluation and to techniques for compilation of declarative programming languages, as well as static optimization of run-time checks.

7 Related Work

We provide an overview of related work on executing constraints using general-purpose solvers at run-time, synthesizing constraints into conventional functional and imperative code, and combining these two extreme approaches by staging the computation between run-time and compile time.

7.1 Run-time Constraint Solving

This proceedings volume contains a notable approach and tool Boogaloo [56], which enables execution of a rich intermediate language Boogie [43]. The original purpose of Boogie is static verification [6]. The usual methods to verify Boogie programs generate conservative *sufficient* verification, which become unprovable if the invariants are not inductive. Tools such as Boogaloo complement verification-condition generation approaches and help developer distinguish errors in programs or specifications that would manifest at run-time from those errors that come from inductive statements not being strong enough. A run-time interpreter for the annotations of the Jahob verification system [79] can also execute certain limited form of specifications, but does not use symbolic execution techniques and treats quantifiers more conservatively than the approach

of Boogaloo. Our current Leon system works with a quantifier-free language; the developers write specifications using recursion instead of quantifiers. Our system allows developers to omit postconditions of defined functions and does not reporting spurious counterexamples. Therefore, it provides the users the advantages of both sound static verification and true counterexamples in a unified algorithm. As remarked, however, unfolding recursive functions can be viewed as a particular quantifier instantiation strategy.

Constraint solving is key for executing programs annotated with contracts because it enables the generation of concrete states that satisfy a given precondition. In our tool we use our approach of satisfiability modulo recursive (pure) functions. In a prior work we have focused on constraint programming using such system [39], embedding constraint programming with recursive functions and SMT solvers into the full Scala language and enabling ranked enumeration of solutions. The Boogaloo approach [56] uses symbolic execution where quantifiers are treated through a process that generalizes deterministic function unrolling to more general declarative constraints. Unfolding is also used in bounded model checking [7] and k-induction approaches [35]. Symbolic execution can also be performed at the level of bytecodes, as in the UDITA system that builds on Java Pathfinder and contains specialized techniques for generating non-isomorphic graph structures [26].

Functional logic programming [3] amalgamates the functional programming and logic programming paradigms into a single language. Functional logic languages, such as Curry [47] benefit from efficient demand-driven term reduction strategies proper to functional languages, as well as non-deterministic operations of logic languages, by using a technique called *narrowing*, a combination of term reduction and variable instantiation. Instantiation of unbound logic variables occur in constructive guessing steps, only to sustain computation when a reduction needs their values. The performance of non-deterministic computations depends on the evaluation strategy, which are formalized using definitional trees [2]. Applications using functional logic languages include programming of graphical and web user interfaces [28,29] as well as providing high-level APIs for accessing and manipulating databases [11]. The Oz language and the associated Mozart Programming System is another admirable combination of multiple paradigms [72], with applications in functional, concurrent, and logic programming. In particular, Oz supports a form of logical variables, and logic programming is enabled through unification. One limitation is that one cannot perform arithmetic operations with logical variables (which we have demonstrated in several of our examples), because unification only applies to constructor terms.

Monadic constraint programming [58] integrates constraint programming into purely functional languages by using monads to define solvers. The authors define monadic search trees, corresponding to a base search, that can be transformed by the use of *search transformers* in a composable fashion to increase performance. The Dminor language [8] introduces the idea of using an SMT solver to check subtyping relations between refinement types; in Dminor, all types are defined as logical predicates, and subtyping thus consists of proving

an implication between two such predicates. The authors show that an impressive number of common types (including for instance algebraic data types) can be encoded using this formalism. In this context, generating values satisfying a predicate is framed as the *type inhabitation* problem, and the authors introduce the expression elementof$T$ to that end. It is evaluated by invoking Z3 at run-time and is thus conceptually comparable to our find construct but without support for recursive function unfolding. We have previously found that recursive function unfolding works better as a mechanism for satisfiability checking than using quantified axiomatization of recursive functions [69]. In general, we believe that our examples are substantially more complex than the experiences with elementof in the context of Dminor.

The ScalaZ3 library [38], used in Leon, integrates invocations to Z3 into a programming language. Because it is implemented purely as a library, we were then not able to integrate user-defined recursive functions and data types into constraints, so the main application is to provide an embedded domain-specific language to access the constraint language of Z3 (but not to extend it). A similar approach has been taken by others to invoke the Yices SMT solver [20] from Haskell.[4]

7.2 Synthesis of Functions

Our approach blends deductive synthesis [45,46,61], which incorporates transformation of specifications, inductive reasoning, recursion schemes and termination checking, with modern SMT techniques and constraint solving for executable constraints. As one of our subroutines we include complete functional synthesis for integer linear arithmetic [42] and extend it with a first implementation of complete functional synthesis for algebraic data types [32, 67]. This gives us building blocks for synthesis of recursion-free code. To synthesize recursive code we build on and further advance the counterexample-guided approach to synthesis [64].

Deductive synthesis frameworks. Early work on synthesis [45, 46] focused on synthesis using expressive and undecidable logics, such as first-order logic and logic containing the induction principle.

Programming by refinement has been popularized as a manual activity [5,76]. Interactive tools have been developed to support such techniques in HOL [13]. A recent example of deductive synthesis and refinement is the Specware system from Kesterel [61]. We were not able to use the system first-hand due to its availability policy, but it appears to favor expressive power and control, whereas we favor automation.

A combination of automated and interactive development is analogous to the use of automation in interactive theorem provers, such as Isabelle [50]. However, whereas in verification it is typically the case that the program is available, the emphasis here is on constructing the program itself, starting from specifications.

[4] http://hackage.haskell.org/package/yices-easy

Work on synthesis from specifications [65] resolves some of these difficulties by decoupling the problem of inferring program control structure and the problem of synthesizing the computation along the control edges. The work leverages verification techniques that use both approximation and lattice theoretic search along with decision procedures, but appears to require more detailed information about the structure of the expected solution than our approach.

Synthesis with input/output examples. One of the first works that addressed synthesis with examples and put inductive synthesis on a firm theoretical foundation is the one by Summers [66]. Subsequent work presents extensions of the classical approach to induction of functional Lisp-programs [30, 36]. These extensions include synthesizing a set of equations (instead of just one), multiple recursive calls and systematic introduction of parameters. Our current system lifts several restrictions of previous approaches by supproting reasoning about arbitrary datatypes, supporting multiple parameters in concrete and symbolic I/O examples, and allowing nested recursive calls and user-defined declarations.

Inductive programming and programming by demonstration. Inductive (logic) programming that explores automatic synthesis of (usually recursive) programs from incomplete specifications, most often being input/output examples [24, 49], influenced our work. Recent work in the area of programming by demonstration has shown that synthesis from examples can be effective in a variety of domains, such as spreadsheets [60]. Advances in the field of SAT and SMT solvers inspired counter-example guided iterative synthesis [27, 64], which can derive input and output examples from specifications. Our tool uses and advances these techniques through two new counterexample-guided synthesis approaches.

Synthesis based on finitization techniques. Program sketching has demonstrated the practicality of program synthesis by focusing its use on particular domains [62–64]. The algorithms employed in sketching are typically focused on appropriately guided search over the syntax tree of the synthesized program. The tool we presented shows one way to move the ideas of sketching towards infinite domains. In this generalization we leverage reasoning about equations as much as SAT techniques.

Reactive synthesis. Synthesis of reactive systems generates programs that run forever and interact with the environment. However, known complete algorithms for reactive synthesis work with finite-state systems [55] or timed systems [4]. Such techniques have applications to control the behavior of hardware and embedded systems or concurrent programs [73]. These techniques usually take specifications in a fragment of temporal logic [54] and have resulted in tools that can synthesize useful hardware components [33, 34]. Recently such synthesis techniques have been extended to repair that preserves good behaviors [23], which is related to our notion of partial programs that have remaining choose statements.

Automated inference of program fixes and contracts. These areas share the common goal of inferring code and rely on specialized software synthesis techniques [53, 74, 75]. Inferred software fixes and contracts are usually snippets of code that are synthesized according to the information gathered about the analyzed program. The core of these techniques lies in the characterization of runtime behavior that is used to guide the generation of fixes and contracts. Such characterization is done by analyzing program state across the execution of tests; state can be defined using user-defined query operations [74, 75], and additional expressions extracted from the code [53]. Generation of program fixes and contracts is done using heuristically guided injection of (sequences of) routine calls into predefined code templates.

Our synthesis approach works with purely functional programs and does not depend on characterization of program behavior. It is more general in the sense that it focuses on synthesizing whole correct functions from scratch and does not depend on already existing code. Moreover, rather than using execution of tests to define starting points for synthesis and SMT solvers just to guide the search, our approach utilizes SMT solvers to guarantee correctness of generated programs and uses execution of tests to speedup the search. Coupling of flexible program generators and the Leon verifier provides more expressive power of the synthesis than filling of predefined code schemas.

7.3 Combining Run-Time and Compile-Time Approaches

We have argued that constraint solving generalizes run-time checking, and allows the underlying techniques to be applied in more scenarios than providing additional redundancy. The case of optimizing run-time checks also points out that there is a large potential for speedups in executing specifications: in the limit, a statically proved assertion can be eliminated, so its execution cost goes from traversing a significant portion of program state to zero. As is in general the case for compilation, such static pre-computation can be viewed as partial evaluation, and has been successfully applied for temporal finite-state properties [10].

Compilation and transformation of logic programs. Compilation of logic programs is important starting point for improving the baseline of compiled code. A potential inefficiency in the current approach (though still only a polynomial factor) is that the constraint solver and the underlying programming language use a different representation of values, so values need to be converted at the boundary of constraints and standard functional code. Techniques such as those employed in Warren's Abstract Machine (WAM) is relevant in this context [1]. Deeper optimizations and potentially exponential speedups can be obtained using tabling [14], program transformation [59] and partial evaluation [12, 25].

Specifications as a fallback to imperative code. The idea to use specifications as a fall-back mechanism for imperative code was adopted in [57]. Dynamic contract checking is applied and, upon violations, specifications can be *executed*. The technique ignores the erroneous state and computes output values for methods given

concrete input values and the method contract. The implementation uses a relational logic similar to Alloy [31] for specifications, and deploys the Kodkod model finder [70]. A related tight integration between Java and the Kodkod engine is presented in [48]. We expect that automated synthesis will allow the developers to use specifications alone in such scenarios, with a candidate implementation generated automatically.

Data structure repair. It is worth mentioning that this proceedings also contains new results [78] in the exciting area of *data structure repair*. This general approach is related to solving constraints at run-time. The assumption in data structure repair is that, even if a given data structure does not satisfy the desired property, it may provide a strong hint at the desired data structure. Therefore, it is reasonable to use the current data structure as the starting point for finding the value that satisfies the desired constraint, hoping that the correct data structure is close to the current one. Although such approach is slightly more natural in the context of imperative than functional code, it is relevant whenever the data manipulated is large enough. The first specification-driven approach for data structure repair is by Demsky and Rinard [18, 19] where the goal is to recover from corrupted data structures by transforming states that are erroneous with respect to integrity constraints into valid ones, performing local heuristic search. Subsequent work uses less custom constraint solvers instead [21, 22]. We believe that SMT solvers could also play a role in this domain. Researchers [77] have also used method contracts instead of data structure integrity constraints to be able to support rich behavioral specifications, which makes it also more relevant for our scenarios. While the primary goal in most works is run-time recovery of data structures, recent work [44] extends the technique for debugging purposes, by abstracting concrete repair actions to program statements performing the same actions. As in general for run-time constraint solving, we expect that data structure repair can be productively applied to implementations generated using "speculative synthesis" that generates a not necessarily correct implementation.

References

1. H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. S. Antoy. Definitional trees. In *ALP*, pages 143–157, 1992.
3. S. Antoy and M. Hanus. Functional logic programming. *CACM*, 53(4):74–85, 2010.
4. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II*, pages 1–20, 1995.
5. R.-J. Back and J. von Wright. *Refinement Calculus*. Springer, 1998.
6. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.
7. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
8. G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP*, pages 105–116, 2010.

9. R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Scala Workshop*, 2013.
10. E. Bodden, P. Lam, and L. J. Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Trans. Program. Lang. Syst.*, 34(2):7, 2012.
11. B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *PADL*, pages 316–332, 2008.
12. M. Bruynooghe, D. D. Schreye, and B. Krekels. Compiling control. *The Journal of Logic Programming*, 6(12):135 – 162, 1989.
13. M. Butler, J. Grundy, T. Langbacka, R. Ruksenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In *Formal Methods Pacific*, 1997.
14. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.
15. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001.
16. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
17. L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design*, Nov. 2009.
18. B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.
19. B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.
20. B. Dutertre and L. de Moura. The Yices SMT solver, 2006. `http://yices.csl.sri.com/tool-paper.pdf`.
21. B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *ICSE*, pages 855–858, 2008.
22. B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. Starc: static analysis for efficient repair of complex data. In *OOPSLA*, pages 387–404, 2007.
23. C. V. Essen and B. Jobstmann. Program repair without regret. In *CAV*, 2013.
24. P. Flener and D. Partridge. Inductive programming. *Autom. Softw. Eng.*, 8(2):131–137, 2001.
25. J. Gallagher and J. Peralta. Regular tree languages as an abstract domain in program specialisation. *Higher-Order and Symbolic Computation*, 14(2-3):143–172, 2001.
26. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *International Conference on Software Engineering (ICSE)*, 2010.
27. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
28. M. Hanus. Type-oriented construction of web user interfaces. In *PPDP*, pages 27–38, 2006.
29. M. Hanus and C. Kluß. Declarative programming of user interfaces. In *PADL*, pages 16–30, 2009.
30. M. Hofmann. IgorII - an analytical inductive functional programming system (tool demo). In *PEPM*, pages 29–32, 2010.
31. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
32. S. Jacobs, V. Kuncak, and P. Suter. Reductions for synthesis procedures. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

33. B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, 2006.
34. B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, 2007.
35. T. Kahsai and C. Tinelli. PKIND: A parallel k-induction based model checker. In *10th Int. Workshop Parallel and Distributed Methods in verifiCation (PDMC'11)*, 2011.
36. E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *JMLR*, 7:429–454, 2006.
37. E. Kneuss, V. Kuncak, I. Kuraj, and P. Suter. On integrating deductive synthesis and verification systems. Technical Report EPFL-REPORT-186043, EPFL, 2013.
38. A. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: Integrating SMT and programming. In *Computer-Aideded Deduction (CADE) Tool Demo*, 2011.
39. A. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.
40. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2010.
41. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Functional synthesis for linear arithmetic and sets. *Software Tools for Technology Transfer (STTT)*, TBD(TBD), 2012.
42. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Communications of the ACM*, 2012.
43. K. R. M. Leino. This is Boogie 2. Manuscript KRML 178, working draft 24 June 2008.
44. M. Z. Malik, J. H. Siddiqui, and S. Khurshid. Constraint-based program debugging using data structure repair. In *ICST*, pages 190–199, 2011.
45. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
46. Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.
47. E. Michael Hanus. Curry: An integrated functional logic language. `http://www.curry-language.org`, 2006. vers. 0.8.2.
48. A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.
49. S. Muggleton and L. D. Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.
50. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
51. M. Odersky. Contracts for Scala. In *Int. Conf. Runtime Verification*, 2010.
52. C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
53. Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Evidence-based automated program fixing. *CoRR*, abs/1102.1059, 2011.
54. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, 2006.
55. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM.
56. N. Polikarpova, C. A. Furia, and S. West. To run what no one has run before. In *Int. Conf. Runtime Verification*, 2013.
57. H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

58. T. Schrijvers, P. J. Stuckey, and P. Wadler. Monadic constraint programming. *J. Funct. Program.*, 19(6):663–697, 2009.
59. V. Senni and F. Fioravanti. Generation of test data structures using constraint logic programming. In *TAP*, pages 115–131, 2012.
60. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651, 2012.
61. D. R. Smith. Generating programs plus proofs by refinement. In *VSTTE*, pages 182–188, 2005.
62. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.
63. A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.
64. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.
65. S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. In *POPL*, 2010.
66. P. D. Summers. A methodology for LISP program construction from examples. *JACM*, 24(1):161–175, 1977.
67. P. Suter. *Programming with Specifications*. PhD thesis, EPFL, December 2012.
68. P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *ACM POPL*, 2010.
69. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symposium (SAS)*, 2011.
70. E. Torlak and D. Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007.
71. A. Udupa, A. Raghavan, J. Deshmukh, S. Mador-Haim, M. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *ACM Conference on Programming Language Design and Implementation*, 2013.
72. P. Van Roy. Logic programming in Oz with Mozart. In *ICLP*, 1999.
73. M. T. Vechev, E. Yahav, and G. Yorsh. Inferring synchronization under limited observability. In *TACAS*, 2009.
74. Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *ICSE*, 2011.
75. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 61–72, 2010.
76. N. Wirth. Program development by stepwise refinement (reprint). *Commun. ACM*, 26(1):70–74, 1983.
77. R. N. Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *ECOOP*, pages 577–598, 2010.
78. R. N. Zaeem, M. Z. Malik, and S. Khurshid. Repair abstractions for more efficient data structure repair. In *Int. Conf. Runtime Verification*, 2013.
79. K. Zee, V. Kuncak, M. Taylor, and M. Rinard. Runtime checking for program verification. In *Workshop on Runtime Verification*, volume 4839 of *LNCS*, 2007.