Program Extrapolation with Jennisys

K. Rustan M. Leino
Microsoft Research
Redmond, WA, USA
leino@microsoft.com

Aleksandar Milicevic
Massachusetts Institute of Technology (MIT)
Cambridge, MA, USA
aleks@csail.mit.edu

Abstract

The desired behavior of a program can be described using an abstract model. Compiling such a model into executable code requires advanced compilation techniques known as synthesis. This paper presents an object-based language, called Jennisys, where programming is done by introducing an abstract model, defining a concrete data representation for the model, and then being aided by automatic synthesis to produce executable code. The paper also presents a synthesis technique for the language. The technique is built on an automatic program verifier that, via an underlying SMT solver, is capable of providing concrete models to failed verifications. The technique proceeds by obtaining sample input/output values from concrete models and then extrapolating programs from the sample points. The synthesis aims to produce code with assignments, branching structure, and possibly recursive calls. It is the first to synthesize code that creates and uses objects in dynamic data structures or aggregate objects. A prototype of the language and synthesis technique has been implemented.

Categories and Subject Descriptors D.1.2 [*PROGRAMMING TECHNIQUES*]: Automatic Programming

General Terms program synthesis, programming language design, program verification

Keywords abstract specifications, concrete representations, coupling invariants, preconditions, Jennisys, Dafny

0. Introduction

One important approach to ensuring program correctness is to raise the level of abstraction provided by programming languages. If a language lends itself to clean descriptions of solutions in the problem domain, then a programmer may be more likely to get programs correct. Two desiderata in this approach, which may seem to be at odds with each other, are (D0) allowing higher-level descriptions of programs in a general-purpose programming language and (D1) allowing efficient run-time representations of programs [22]. In this paper, we present a language framework that combines these two. The language is called Jennisys, and because it allows program designs to be recorded ahead of their concrete implementations, the language slogan is “This is where programs begin”.

Most programming languages provide some delineation between the public specification of a procedure, type, or module and the private implementation thereof. In some cases, a public specification consists of just a type signature; in other cases, it may include a behavioral contract [5, 18]. Jennisys takes the delineation a step further, dividing every program component (or class, if you will) into three parts, which allows a separation between data-structure definitions and code.

The first part of a Jennisys component is the *public interface* (cf. Fig. 0, the `interface` declaration). It defines an abstract model of the component, given in terms of variables whose types are often mathematical structures, like sets and sequences. The public interface also defines the component's operations and the behavioral effects of these, typically given in terms of simple code snippets that act on the model variables. The model variables and the code acting on these describe the component, but are not compiled as part of the run-time manifestation of the program.

The second part describes the data structure used to represent the component at run time (cf. Fig. 1, the `datamodel` declaration). More specifically, it declares concrete variables (object fields, if you will) that are part of each instance of the component, gives an account of which other component instances are part of the representation (this is called the *frame*), and specifies an *invariant* that both constrains the concrete variables and frame and couples these with the model variables in the public interface.

The third part of a Jennisys component is responsible for the executable code that will implement the component operations (e.g. `code IntSet{}`). The vision is for the language to provide the programmer with a variety of ways to produce the code, including automatic code synthesis (which

```
interface IntSet {
  var elems: set[int]

  constructor Singleton(x: int)
    elems := {x}

  constructor Dupleton(x: int, y: int)
    requires x ≠ y
    elems := {x y}

  method Find(x: int) returns (ret: bool)
    ret := x ∈ elems
}
```

Figure 0. A Jennisys public *interface* `IntSet` that abstractly defines an integer set data structure.

```
datamodel IntSet {
  var data: int
  var left: IntSet
  var right: IntSet

  frame left * right

  invariant
    elems = {data} +
            (left ≠ null ? left.elems : {}) +
            (right ≠ null ? right.elems : {})
    left ≠ null  ⟹
        (∀ e • e ∈ left.elems ⟹ e < data)
    right ≠ null ⟹
        (∀ e • e ∈ right.elems ⟹ data < e)
}
```

Figure 1. A concrete *data structure*, namely a binary tree for the `IntSet` interface. Model variable `elems` is used to describe the behavior of the operations, but is itself not compiled into executable code.

is our focus in this paper), code-generation hints, program sketches [28], and, as a last resort, old-fashioned manual coding.

Jennisys is general purpose, addressing (D0). Its public interfaces let programmers write clean descriptions whose correctness can more easily be ascertained by human scrutiny. The variety of ways to obtain code aims to speed up code production and maintenance. The data-structure description addresses (D1) by letting programmers use their insights into defining good data structures.

Jennisys is still a prototype. In this paper, we focus on the automatic code synthesis. In particular, we contribute a technique that from abstract variables, abstract code, concrete variables, and a coupling invariant (in other words, from the **interface** and **datamodel** of a component) synthesizes loopless structured programs, where each "if" branch contains assignments to modifiable fields (one assignment per field) and possibly some method calls. The synthesis technique is most readily applicable to constructors, but its class of applications extends beyond that; for example, we show we can synthesize code for some recursive methods for traversing or computing some properties of complex data structures.

In a nutshell, our technique uses a program verifier to obtain sample input/output values that satisfy the given specifications. The sample values are then extrapolated into code for all input values. Frequently, the synthesized code will contain necessary branching structure, will allocate new instances of other components, and will call methods on those components in order to change their state. As far as we know, this is the first code synthesizer to reason about pointers to objects, let alone synthesize code that allocates and uses objects in data structures.

1. Examples

In this section, we give examples that illustrate the use of Jennisys and the code it synthesizes.

Figure 0 shows the public interface of a Jennisys component that we will use as a running example, `IntSet`.[0] Abstractly, an `IntSet` is an integer set, which we define by model variable `elems`. The constructors create a set of size 1 or 2, respectively, and method `Find` returns whether or not a given integer is part of the set. An operation can define a *precondition* (keyword **requires**), which obligates callers to invoke the operation only when the condition is met. The effect of an operation is given by assignments (as in Fig. 0) or relations (constraints) on the pre- and post-states (see Fig. 16).

A concrete data structure for `IntSet` is described using the **datamodel** declaration. It contains three kinds of declarations—**var**, **frame**, and **invariant**—which we explain next.

The variable declarations (e.g. **var** data: **int**) introduce the familiar fields of a binary-tree node. Unlike the model variables in the public interface, these concrete variables will be present in the run-time representation of `IntSet` components.

The **frame** declaration says that the memory locations used to represent an `IntSet` include not just the `IntSet` object itself, but also those memory locations that are used to represent the `IntSet` components `left` and `right`. The star, inspired by the notation of separation logic [21], says that the sets of memory locations used by `left` and `right` are disjoint. The frame declaration is necessary for the verification of candidate synthesized programs and tells the synthesis engine which parts of the underlying state a method may mutate.

[0] In this paper, we sometimes stray from the concrete syntax of our Jennisys prototype in order to make programs easier to read. Most notably, we replace the ASCII syntax of some operators by common mathematical notation. The actual Jennisys programs are available online in the Jennisys distribution.

The **invariant** declaration defines a relation between the model variables and the concrete variables, as in a *coupling invariant* (aka an *abstraction invariant* or *retrieve relation*, see, e.g., [1, 2, 14]). It also constrains the values of the concrete representation, as in a *class invariant* [18].

From Figs. 0 and 1, Jennisys automatically synthesizes code for the three operations. We give an excerpt of that code in Fig. 2. The target language is Dafny [16], which for us has the advantage that we can use the Dafny verifier to double check the correctness of the synthesized code. We also use Dafny during the synthesis itself. Dafny compiles to the .NET virtual machine, so there is in principle no reason why Jennisys could not compile to any Java-like language.

Some interesting things to note about the synthesized code are the **if** statements in Dupleton and Find. Also, note the dynamic allocation of objects on line 50, the call of another constructor on line 51 (to respect the abstraction boundary of the non-**this** object gensym85), the use of ghost variables on lines 2, 46, 53, 56, ... (these are needed only during verification, not at run time), and the recursive calls to Find on lines 83, 84, 88, and 92.

Note, although the operations in the public IntSet interface in Fig. 0 can only construct sets with cardinality 1 or 2 (because our Jennisys prototype is currently not up to the task of synthesizing code for a Union method), the data representation we define in Fig. 1 allows arbitrarily large sets. Indeed, the synthesized code for Find will work for any concrete data structure satisfying the invariant.

2. Dynamic Synthesis

This section focuses on the algorithm for program synthesis behind Jennisys. We call this algorithm *dynamic synthesis*, because it combines ideas from both concrete and symbolic execution, in a way that is similar to what concolic testing [6, 25, 31] does. In contrast to concolic testing, however, declarative specifications are being executed, rather than traditional imperative code.

We first describe how Dafny [16], a program verifier for functional correctness, can be used to execute first-order declarative specifications of Jennisys. Dafny is implemented on top of Boogie [4], an intermediate verification language, which via an SMT solver, namely Z3 [7], attempts to automatically discharge verification conditions.

Executing a specification gives only a single valid input/output pair, that is, a pair of concrete instances of the program heap (one for pre-state and one for post-state) for which the specification holds. In order to synthesize a program that is correct for all possible cases (i.e. all possible pre-states) this is clearly not enough. To this end, we next present an algorithm for systematic state exploration and program extrapolation from concrete instances. Since the problem of synthesis is undecidable, the algorithm does not always succeed, but when it does, the synthesized program is provably correct (it can be automatically verified against the

```
1   class IntSet {
2     ghost var Repr: set<object>;
3     ghost var elems: set<int>;
4     var data: int, left: IntSet, right: IntSet;

27    function Valid(): bool
```

(omitted, is defined to return **true** when the invariants of all reachable objects hold)

```
43    method Dupleton(x: int, y: int)
44      modifies this;
45      requires x ≠ y;
46      ensures Valid() ∧ fresh(Repr - {this});
47      ensures elems = {x, y};
48    {
49      if (x < y) {
50        var gensym85 := new IntSet;
51        gensym85.Singleton(y);
52        this.data := x;
53        this.elems := {x, y};
54        this.left := null;
55        this.right := gensym85;
56        this.Repr := {this} + this.right.Repr;
57        assert gensym85.Valid();
58      } else {
```

(the other case is symmetric)

```
74    } }
75
76    method Find(x: int) returns (ret: bool)
77      requires Valid();
78      ensures Valid() ∧ fresh(Repr - old(Repr));
79      ensures ret = (x ∈ elems);
80      decreases Repr;
81    {
82      if (this.left ≠ null ∧ this.right ≠ null) {
83        var x_27 := this.left.Find(x);
84        var x_28 := this.right.Find(x);
85        ret := (x = this.data ∨ x_27) ∨ x_28;
86      } else {
87        if (this.left ≠ null) {
88          var x_29 := this.left.Find(x);
89          ret := x = this.data ∨ x_29;
90        } else {
91          if (this.right ≠ null) {
92            var x_30 := this.right.Find(x);
93            ret := x = this.data ∨ x_30;
94          } else {
95            ret := x = this.data;
96    } } } }

120 }
```

Figure 2. Excerpts of the Dafny code that Jennisys synthesizes for the IntSet example. For brevity, the figure combines some lines. Dafny's ghost variables are not present during the run-time execution of Dafny programs, but are needed to verify the correctness of the synthesized program.

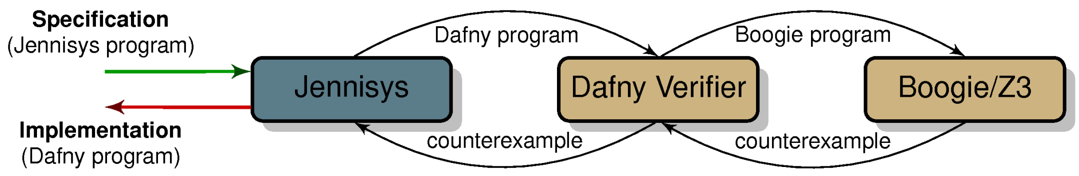


Figure 3. *Architecture* of the Jennisys tool. The Jennisys synthesizer relies on Dafny, a program verifier for functional correctness implemented on top of Boogie/Z3, to (1) execute Jennisys specifications (i.e., obtain concrete program heaps that satisfy such a specification), and (2) verify the correctness of programs extrapolated from those concrete heaps. This process is done iteratively until a correct program is synthesized or the search heuristic terminates.

original specification using Dafny). The overall architecture of the Jennisys tool is depicted in Fig. 3.

2.1 Concrete Specification Execution with Dafny

By "executing a method specification" we mean "finding *arbitrary* pre- and post-states that satisfy that specification"[1]. Dafny, even though designed for program verification, is suitable for this task. The basic idea is to tell Dafny to assume (using the **assume** keyword) the pre-condition, the post-condition, and all the invariants, then ask it to derive false from there (as in Fig. 4, which shows a Dafny code snipped used to execute the specification of the `IntSet.Dupleton` method). If Dafny succeeds, the pre- and post-conditions are mutually inconsistent, so any attempt to synthesize code for such a specification would be futile. Otherwise, Dafny returns a counterexample where all the assumed constraints hold is returned, so concrete values for the pre- and post-states can be directly extracted from it.

2.2 Symbolic and Concrete Execution Combined

Assigning concrete values (constants) obtained by executing a specification to *output variables* is unlikely to result in a program that is correct for inputs other than the one discovered by the execution of that specification. The goal is, therefore, to try and generalize from a concrete instance and find *symbolic* assignments for output variables. Furthermore, even though more general than constants, such symbolic assignments might be correct only for certain program scenarios represented by the concrete instance used. When that is the case, a logical condition (guard) must be inferred to characterize those particular scenarios.

To solve the problem of finding a guard and a set of symbolic assignments, the specification is first **partially evaluated** with respect to the previously obtained concrete instance. This process yields a specification that is simpler and more specific to the current instance. This simplified specification is then **symbolically executed** to arrive at a set of symbolic expressions that can be used, depending on the type, as potential guards or variable assignments.

[1] Note that this is slightly different from some previous work where executing specifications is part of the runtime system (e.g. [19, 23]). In those systems, the pre-state is always explicitly known (it is the state of the running program before the specification is executed), so the goal there is to find a valid post-state for *a given* pre-state.

```
class IntSet {
  ghost var elems: set<int>;
  var data: int;
  var left: SetNode, right: SetNode;

  function Valid(): bool {
    -- all invariants inlined
  }

  method Dupleton() modifies this; {
    var x: int, y: int;
    assume a ≠ b ∧ elems = {a, b};
    assume Valid();
    assert false;
  }
}
```

Figure 4. Translation of `IntSet.Dupleton` into Dafny for specification execution. Since Jennisys and Dafny share the same language (modulo the exact syntax), this translation is fairly straightforward.

If it can be verified that the chosen symbolic assignments are correct given the inferred guard, one branch of the target program is successfully synthesized. To discover the rest of the program, a new program specification is created by adding the negation of the inferred guard as an additional pre-condition. The synthesis process is then recursively repeated for the new specification to discover the 'else' counterpart of the previously synthesized branch. This process allows for synthesis of programs in the form of an arbitrarily long *if-then-elseif-then-elseif-...-else* structure.

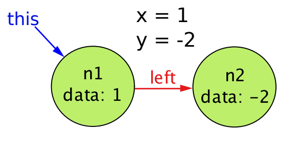
The question of termination immediately comes to mind; we discuss this question in Sec. 3.6.

3. Synthesis Algorithm in Depth

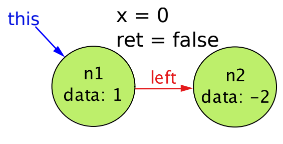
3.1 Partial Specification Evaluation

Let us assume throughout this section that the initial execution of the specification of `Dupleton` yielded the instance shown in Fig. 5(a).

A method specification can be unfolded for a given concrete instance by means of inlining invariants of every



(a) `Dupleton`



(b) `Find`

Figure 5. Concrete instances automatically generated for methods `Dupleton` and `Find`.

heap object in that instance. Unfolding the specification of `Dupleton` for the instance from Fig. 5(a) gives:

```
x ≠ y ∧ n1.elems = {x y}

n1.elems = {n1.data} +
           (n1.left ≠ null ? n1.left.elems : {}) +
           (n1.right ≠ null ? n1.right.elems : {})
n1.left ≠ null ⟹
    (∀ e • e in n1.left.elems ⟹ e < n1.data)
n1.right ≠ null ⟹
    (∀ e • e in n1.right.elems ⟹ n1.data < e)

n2.elems = {n2.data} +
           (n2.left ≠ null ? n2.left.elems : {}) +
           (n2.right ≠ null ? n2.right.elems : {})
n2.left ≠ null ⟹
   (∀ e • e in n2.left.elems ⟹ e < n2.data)
n2.right ≠ null
   (∀ e • e in n2.right.elems ⟹ n2.data < e)
```

This expression as a whole must evaluate to true, because the instance was generated so that the specification holds for it. The insight is, however, that some subexpressions of the specification need not be relevant for the particular instance at hand (e.g., a consequent of an implication whose antecedent is false). For example, in this concrete instance, `n1.right`, `n2.left`, and `n2.right` are all **`null`**, so with that in mind, the previous constraint can easily be simplified to arrive at what we will refer to as a *heap expression*:

```
x ≠ y ∧ n1.elems = {x y}

n1.elems = {n1.data} + n1.left.elems
∀ e • e in n1.left.elems ⟹ e < n1.data

n2.elems = {n2.data}
```

We call this notion of simplification *partial evaluation* and define it formally in Fig. 6[2]. The basic idea is to drop all disjunction terms that when fully evaluated (using the *eval*[3] function) give false, since they are likely to be irrelevant for the current instance.

The *apply* function is used to reconstruct symbolic expressions along the way. Its significance is that it additionally performs some well-known simplifications. Besides short-circuiting boolean expressions, it implements several rules specifically designed for the task of synthesis. The most interesting example would be the simplification rules for operations over sequences, as depicted in Fig. 7; in particular, they enable decomposition of specifications involving sequences into smaller bits which are often simpler to synthesize code from.

[2] For brevity, the instance parameter is not explicitly used in the definition in Fig. 6; it is instead assumed to be the "current" instance.

[3] The *eval* function, given a concrete instance evaluates an expression to a constant. This is a well-known notion of evaluation, so we do not give a formal definition here.

3.2 Symbolic Specification Execution

After obtaining a heap expression for the current instance (by computing $\mathcal{E}(e)$, where e is the unfolded version of the original specification), a *database of premises* is created. All premises are boolean expressions, e.g., `a = 5` or `a > b`, but not `a + b`. The initial set of premises includes all the conjuncts of the heap expression:

```
x ≠ y;
n1.elems = {x y};
n1.elems = {n1.data} + n1.left.elems;
∀ e • e in n1.left.elems ⟹ e < n1.data;
n2.elems = {n2.data};
```

as well as $v = eval(v)$ mappings for all variables:

```
this = n1;
x = 1;
y = -2;
n1.data = 1;    n1.left = n2;    n1.right = null
n2.data = -2;   n2.left = null;  n2.right = null
```

Using the *inference rules* defined in Fig. 8, new premises are derived from existing ones and are added to the database. This process is repeated until either a fixpoint or a predefined maximum number of iterations is reached.

The main purpose of the inference rules from Fig. 8 is to decompose and simplify expressions over the built-in data structures. For example, from a specification like $x \in e_1 + e_2$, and a concrete instance in which x is not in the sequence e_2, $x \in e_1$ can be safely derived. These rules derive expressions specific to the current instance, and thus help infer appropriate guards and symbolic assignments. Some rules are independent of the concrete instance, e.g., $x \in [e_0,\ e_1,\ \cdots,\ e_{n-1}] \vdash x \in [e_0] + [e_1,\ \cdots,\ e_{n-1}]$; their purpose is mainly to enable rules of the previous kind to get instantiated more often.

3.3 Choosing Correct Assignments for Output Variables

At the end of the previous step, the database might (and typically does) contain multiple assignments for each variable. Jennisys automatically rules some of them out, and uses a heuristic to rank the remaining ones. In order for an assignment to be considered valid, its right hand side must contain only references to either pre-state or unmodifiable variables. Between the valid assignments, Jennisys prefers those that contain symbolic, rather than constant values.

$\mathcal{E} : Expr \rightarrow Expr$

		rewriting rules
$\mathcal{E}(Const)$	$\equiv$	$Const$
$\mathcal{E}(Var)$	$\equiv$	Var
$\mathcal{E}(\lvert e \rvert)$	$\equiv$	$apply(\lvert\rvert,\ \mathcal{E}(e))$
$\mathcal{E}([e_0, e_1, \ldots, e_{n-1}])$	$\equiv$	**List.map** $\mathcal{E}\ [e_0, e_1, \ldots, e_{n-1}]$
$\mathcal{E}(\{e_0, e_1, \ldots, e_{n-1}\})$	$\equiv$	**Set.map** $\mathcal{E}\ \{e_0, e_1, \ldots, e_{n-1}\}$
$\mathcal{E}(lst[idx])$	$\equiv$	$apply([],\ \mathcal{E}(lst),\ \mathcal{E}(idx))$
$\mathcal{E}(e_1\ \rho\ e_2)$	$\equiv$	$apply(\rho,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\rho\ -$ relational operator: $=,\ \neq,\ <,\ \leq,\ >,\ \geq,\ \in,\ \notin$
$\mathcal{E}(e_1\ \alpha\ e_2)$	$\equiv$	$apply(\alpha,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\alpha\ -$ arithmetic operator: $+,\ -,\ *,\ /,\ \%$
$\mathcal{E}(\forall v \bullet e)$	$\equiv$	$\forall v \bullet e$
		simplification of logic expressions
$\mathcal{E}(c\,?\,t : e)$	$\equiv$	**if** $eval(c)$ **then** $\mathcal{E}(t)$ **else** $\mathcal{E}(e)$
$\mathcal{E}(e_1 \wedge e_2)$	$\equiv$	$apply(\wedge,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
$\mathcal{E}(e_1 \vee e_2)$	$\equiv$	**match** $eval(e_1),\ eval(e_2)$ **with**
		$\mid$ true, true $\rightarrow apply(\vee,\ \mathcal{E}(e_1),\ \mathcal{E}(e_2))$
		$\mid$ true, false $\rightarrow \mathcal{E}(e_1)$
		$\mid$ false, true $\rightarrow \mathcal{E}(e_2)$
		$\mid$ false, false $\rightarrow False$
$\mathcal{E}(e_1 \implies e_2)$	$\equiv$	$\mathcal{E}(\neg e_1 \vee e_2)$
$\mathcal{E}(e_1 \iff e_2)$	$\equiv$	$\mathcal{E}((e_1 \wedge e_2) \vee (\neg e_1 \wedge \neg e_2))$
$\mathcal{E}(\neg e)$	$\equiv$	$apply(\neg,\ \mathcal{E}(e))$

helper functions:

$eval$	$: Expr \rightarrow Const$	–	evaluates an expression to a constant wrt the current instance
$apply$	$: Op \rightarrow Expr$ list $\rightarrow Expr$	–	applies a given operator to given operands

Figure 6. *Partial expression evaluation* function ($\mathcal{E}$): partially evaluates a given expression with respect to a concrete instance, making it simpler and more specific to that instance.

$apply : Op \rightarrow Expr$ list $\rightarrow Expr$

		Simplifications for the || operator
$apply(\lvert\rvert,\ l_1 + l_2)$	$\equiv$	$apply(\lvert\rvert,\ l_1) + apply(\lvert\rvert,\ l_2)$
$apply(\lvert\rvert,\ [e_0,\ \ldots,\ e_{n-1}])$	$\equiv$	n
		Simplifications for the [] operator
$apply([],\ [e_0,\ \ldots,\ e_i,\ \ldots,\ e_{n-1}],\ i)$	$\equiv$	e_i
$apply([],\ [e_0,\ \ldots,\ e_{k-1}] + l,\ i)$	$\equiv$	**if** $i < k$ **then** e_i **else** $apply([],\ l,\ i - k)$

Figure 7. Simplifications of the sequence length and sequence select expressions performed by the *apply* function.

			inference rules for $\in$
$x \in []$	$\vdash$	$False$	(1)
$x \in [e]$	$\vdash$	$x = e$	(2)
$x \in [e_0, e_1, \ldots, e_{n-1}]$	$\vdash$	$x \in [e_0] + [e_1, \ldots, e_{n-1}]$	(3)
$x \in \{\}$	$\vdash$	$False$	(4)
$x \in \{e\}$	$\vdash$	$x = e$	(5)
$x \in \{e_0, e_1, \ldots, e_{n-1}\}$	$\vdash$	$x \in \{e_0\} + \{e_1, \ldots, e_{n-1}\}$	(6)
$x \in e_1 + e_2$			
when $eval(x \in e_1) \wedge eval(x \notin e_2)$	$\vdash$	$x \in e_1$	(7)
when $eval(x \notin e_1) \wedge eval(x \in e_2)$	$\vdash$	$x \in e_2$	(8)
else	$\vdash$	$x \in e_1 \vee x \in e_2$	(9)
			inference rules for $\forall$
$\forall x \in [e_0,\ \ldots,\ e_{n-1}] \bullet p$	$\vdash$	$p[x \rightsquigarrow e_0] \wedge \ldots \wedge p[x \rightsquigarrow e_{n-1}]$	(10)
$\forall x \in \{e_0,\ \ldots,\ e_{n-1}\} \bullet p$	$\vdash$	$p[x \rightsquigarrow e_0] \wedge \ldots \wedge p[x \rightsquigarrow e_{n-1}]$	(11)
$\forall x \in e_1 + e_2 \bullet p$	$\vdash$	$(\forall x \in e_1 \bullet p) \wedge (\forall x \in e_2 \bullet p)$	(12)

Figure 8. Inference rules for *symbolic execution.*

From the initial database for the `Dupleton` example, just by applying transitivity of equality, the following *candidate solution* is quickly discovered (other assignments exists in the database, but they all contain constant values):

```
n1.elems := {x y};      n2.elems := {y};
n1.data  := x;          n2.data  := y;
n1.left  := n2;         n2.left  := null;
n1.right := null;       n2.right := null;
```

As expected, this solution does not verify against the original specification of the `Dupleton` method. Knowing the properties of binary search trees, it is easy for us to conclude that the solution we just discovered is valid only for cases where `y < x` holds. In the next subsection we show how, when needed, Jennisys automatically infers such logical conditions (guards).

When no guard is needed (i.e., the candidate solution verifies at this point), the solution is simply returned to the top-level synthesis function (see Sec. 3.5) where the algorithm finishes, since the last 'else' branch of the outer *if-then-elif-...-else* structure has just been found.

3.4 Inference of Guards

The main insight for successful guard inference is that an appropriate guard is likely to be a logical property of the current instance. Therefore, a guard is likely to consist of one or more premises from the database.

Going back to the example, the `y < x` condition was indeed derived during the execution of the fixpoint algorithm. Namely, from

```
n1.left = n2
n2.elems = n2
∀ e • e in n1.left.elems ⟹ e < n1.data
```

just by applying transitivity of equality the following premise is derived:

```
∀ e in {n2.data} • e < n1.data
```

Applying rule 11 leads to `n2.data < n1.data`, from which `y < x` immediately follows (since both `n2.data` = `y` and `n1.data` = `x` are already in the database).

Jennisys selects a candidate guard by going through the database and looking for expressions that involve only unmodifiable variables and constants (expressions without constants are again ranked higher). When multiple such expressions exist, a conjunction of all of them is used first. If a candidate solution verifies under the assumption of a selected guard, the guard is minimized by iteratively trying to remove one clause at a time. For example, during the synthesis of the `Dupleton` method, `x ≠ y ∧ y < x` was selected as a guard first, and was next minimized to `y < x`.

3.5 Top-level Algorithm

The top-level synthesis function, `synth`, is given in Fig. 9. At the very beginning (line 2), it invokes `synth_branch` to find a solution for the current instance only (exactly by following the procedure described so far). If no verified solution is found (line 4), Jennisys gives up.

If both guard and a solution were found (line 6), the guard is negated and appended to the list of pre-conditions to ensure that all subsequent concrete instances obtained by executing the specification fall outside this of branch. The whole process is then repeated to find a solution for the *else* branch.

Finally, if a solution was found for which a guard was not needed (line 5), a solution for the entire program is discovered (not just the current instance!). That is true because a solution is just proven unconditionally correct for the portion of the program space not covered by the previously synthesized branches. This solution represents the last *else* branch of the *if-then-elif-...-else* structure that our approach synthesizes.

```
let Solution = FlatSol | IfThenElse(Guard, FlatSol, Solution)
let rec synth (m: Method): Solution =
  let guardOpt,flatSolOpt = synth_branch m
  match flatSolOpt, guardOpt with
  | None, _                    -> None
  | Some(flatSol), None        -> Some(flatSol)
  | Some(flatSol), Some(guard) ->
      match synth (AddPrecondition m (not guard)) with
      | None -> None
      | Some(solElse) -> Some(IfThenElse(guard, flatSol, solElse))
```

Figure 9. Top level algorithm in pseudo F#

3.6 Termination

The synthesis algorithm terminates when one of the following conditions is met: (0) a candidate solution is found and it verifies without needing a guard (synthesis succeeds), (1) a candidate solution is found, but it doesn't verify and a guard cannot be inferred (synthesis fails), or (2) no candidate solution is found (synthesis fails).

An important question is whether the algorithm is guaranteed to always terminate. The only way for the synth function not to terminate is if it is possible to forever keep generating new concrete instances and each time finding a guarded solution. The way we generate new instances guarantees that at each step the remainder of the search space is getting smaller, because every new instance is guaranteed to be outside of the previously discovered classes of programs (characterized by previously discovered guards). It can happen, however, that at each step the inferred guard is over-constrained (e.g., it does not allow any instance other than the current one). In that case, if the search space is unbounded (that is, there are infinitely many different instances for the program under analysis), the algorithm potentially never terminates. To mitigate this, Jennisys always prefers solutions and guards with no constants so that at every step the remainder of the search space is shrunk as much as possible. In practice, this means that the algorithm is likely to either terminate with a solution or fail quickly.

4. Synthesizing Recursive Methods

The synthesis algorithm described so far was designed to support constructors in the form of a single (but of arbitrary length) *if-then-elseif-...-else* structure, where the only allowed statements are assignments to output variables. In this section, we show how we extended the algorithm to allow method calls (including recursion) in the assignment statements. Allowing recursion somewhat makes up for the lack of looping constructs.

Two modifications to the synthesis algorithm are needed: (0) after building the initial set of premises, **parameterized** expressions corresponding to method specifications are added to the database; and (1) the inference engine for symbolic execution is modified so that it allows matching with **unification**.

Allowing parameterized expressions means allowing expressions to have placeholders (or variables, if you will), which can be substituted by any other expressions of the same type. Concretely, expressions corresponding to method specifications will have such placeholders for the method parameters and the method receiver object. For example, such an expression for the IntSet.Find method would look like:

$$\texttt{\$this.Find(\$x) = \$x} \in \texttt{\$this.elems} \qquad (0)$$

To be able to instantiate inference rules against parameterized expression, a form of unification is needed. In our prototype implementation, we used a simple syntactic unification algorithm, which currently enables us to synthesize only read-only methods (such as IntSet.Find). Replacing this algorithm with a suitable form of semantic unification would potentially enable synthesis of more complex recursive methods (such as binary search tree insertion).

To illustrate this, consider the IntSet.Find method. Let us assume that after its specification is executed for the first time, the instance from Fig. 5(b) is discovered. The initial set of premises looks almost the same as before (since the instance is almost the same), with a difference of the first line (the pre-condition from the previous example) being replaced with the post-condition of the Find method (ret = x ∈ **this**.elems) and a parameterized specification for the Find method. The derivation then goes as follows[4]:

```
ret = x ∈ this.elems
  → ret = x ∈ {this.data} + this.left.elems        (1)
  → ret = x ∈ {this.data} ∨ x ∈ this.left.elems    (2)
  → ret = (x = this.data) ∨ x ∈ this.left.elems    (3)
  → ret = (x = this.data) ∨ this.left.Find(x)      (4)
```

Equations 2 and 3 are derived directly by applying rules 9 and 5 (from Fig. 8) respectively. Equation 4 is next derived by *matching up* x ∈ **this**.left.elems (from equation 3) and \$x ∈ \$this.elems (from premise 0, i.e., the right-hand side of the parameterized specification of the Find method). To establish that match, the following two unifications are needed: \$this ⇝ this.left and \$x ⇝ x. Finally, with those two unifications and premise 0, x ∈ **this**.left.elems can be replaced with **this**.left.Find(x), which directly derives equation 4.

The remainder of the process stays the same. The guard for this instance (**this**.left ≠ **null**) is easily inferred (note that it is okay now to use **this**.left in the guard, because

[4] Note that the derivation would be slightly different if, for example, in the concrete instance it happens that the set contains the input value x. In that case, at the second step either rule 7 or 8 would be instantiated instead of rule 9, meaning that the disjunction would be simplified so that only one side remains, depending on where the value x is actually found.

this is unmodifiable in this case) and the process continues the same way to synthesize the rest of the program. The final program is shown in Fig. 2.

5. Boilerplate Code to Aid Verification

As noted earlier, our synthesis algorithm requires a fully automated program verifier. The algorithm described in Sec. 3 and 4, however, is verifier agnostic — any automated tool for functional program verification (capable of generating concrete counterexamples) can be used.

Full functional verification of object-oriented programs with pointers, memory allocation, and recursive calls is not an easy task, and, in general, cannot be done automatically. Jennisys uses the Dafny program verifier, which is fully automated, but for successful verification (of a program already correct) it often requires some extra input from the user (in the form of extra code or constraints, but not interactive sessions during the verification process). Luckily, certain Dafny specification idioms are known to be helpful in such cases. We describe here how Jennisys uses these specifications idioms to aid verification.

A common idiom in Dafny is to have a 'ghost' field (typically named `Repr`) to hold the set of all the constituent objects of **this** (i.e., the objects used to represent the data structure of **this**, aka the *footprint* of **this**). In the `IntSet` example, the `Repr` field holds all objects of the `IntSet` (that is, all nodes reachable from the current node by following the `left` and `right` pointers). The `Repr` field is then conveniently used to specify common frame properties like: (0) all objects except **this** that a constructor makes part of the footprint are newly allocated — `fresh(Repr-{this})` (e.g., Fig. 2, line 46), and (1) all objects in a method's post-state footprint that were not in the method's pre-state footprint are newly allocated — `fresh(Repr-old(Repr))` (e.g., Fig. 2, line 78). Furthermore, we use the `Repr` field as a *termination measure* to prove termination of recursive methods; we use it idiomatically in the form of `decreases Repr` (e.g., Fig. 2, line 80) since a recursion is often invoked on an object with a smaller footprint.

Invariants are required to hold in the pre- and post-states of each Jennisys method. To supply such information to Dafny, a standard idiom is to encapsulate the definition of the invariant in a function (in our case called `Valid`), and then assert that it holds in both pre- and postcondition (e.g., Fig. 2, lines 77 and 78). Generated from the Jennisys component specification, the `Valid` function says that the representation and coupling invariants defined in the Jennisys model hold, that `Repr` contains all reachable objects, and that these reachable objects are themselves valid (by the same definition). Since this is a recursive definition, Jennisys can be configured to unroll it a given number of times, or to generate a recursive Dafny function (using the `decreases Repr` trick to prove its termination), or to do both (which, in our experience, is often useful). For an example, see how the `Valid` function is generated for the singly-list example (Fig. 13).

6. Experiments and Evaluation

To evaluate our synthesis algorithm, we show that Jennisys can successfully and reasonably quickly synthesize constructors for several complex data structures, as well as some read-only recursive methods, solely from abstract first-order specifications. To the best of our knowledge, no other approach generates code with such dynamic features from declarative specifications.

We wrote abstract specifications in Jennisys for a number of constructors and methods for the following data structures: binary search tree (`IntSet`), binary heap (`BHeap`), singly-linked list (`List`), and doubly-linked list (`DList`). We also specified several simple mathematical operations to show how Jennisys is capable of handling convoluted declarative specifications. Jennisys programs for these experiments are given in Figs. 0, 10, 12, 14, and 16 respectively, and corresponding synthesized programs are given in Figs. 2, 11, 13, 15, and 17.

Table 0 shows the results of the experiments. For each program, we show whether it is a constructor or a method (*type*), how many branches it has (*#branches*), how many times the verifier was invoked (*#Dafny*), and the total synthesis time in seconds (time). The *#Dafny* column is the total number of times Dafny was run for all different purposes, including executing a specification, verifying a synthesized program, various auxiliary verification tasks (e.g., during the minimization of guards), etc. All experiments were done on an Intel$^{®}$ Core^{TM}2 Duo CPU @ 2.40GHz computer, with 4GB of RAM, running 32-bit Windows 7.

Most of the programs were successfully synthesized within 25 seconds. Some of the programs, whose solutions required 4 branches, had to run the program verifier up to 23 times, and therefore the entire synthesis process took longer, up to 65 seconds.

Jennisys was able to handle the full declarativeness of the specifications we wrote. For example, the specification for `List.Size` (Fig. 12) is simply the length of the abstract `list` field used to model the contents of this data structure. Other than the coupling invariant between the abstract and concrete state (best programming practices would recommend this invariant to be written anyway), no other hints about the program structure had to be given. The synthesized code is a recursive method (Fig. 13).

Another example of Jennisys being able to handle declarative specifications are the `Min` methods in the `Math` component (Fig. 16). The specification of `Min2` (minimum of two integers) is written in a direct, almost imperative way, using two implications which could be translated to an imperative language based on syntactic rules only. Writing a specification for the `Min4` method (minimum of four) in the same style, however, would be much less convenient, and would require explicit enumeration of possible orderings between the four input arguments. Instead, we wrote it succinctly in a more declarative fashion, simply saying that the result must be equal to one of the four inputs, and that it must be less than or equal to each one of them. Synthesizing code from such a specification becomes much harder and an attempt to translate it based on syntactic rules would likely fail.

Note how the `IntSet` and `BHeap` examples demonstrate a key feature of a programming language supported by synthesis. The data model for `IntSet` in Fig. 1 says that the concrete data structure should be a binary search tree. Suppose the `IntSet` developer later decides that a binary-heap representation would be a better choice. This change is accomplished by simply replacing the invariant with the one in Fig. 10. The necessary code changes are left for Jennisys to take care of; in fact, Jennisys will then synthesize the code shown in Fig. 11.

Currently, Jennisys can synthesize only constructors that create objects of a fixed (known in advance) size. In other words, our algorithm can, for example, synthesize code that constructs a binary heap of 2 objects, or 3 objects, but not of all objects in a given list (because such an operation would require a loop). On the other hand, for a constructor with a fixed number of input parameters, Jennisys can automatically partition the input space (which is possibly infinite) into a finite number of parts, for each part synthesizing straight-line code that implements the constructor for that part of the input space.

7. Related Work

The idea of *automatic code generation* dates back to the first Autocoder [8] systems from the 1950s which offered an automatic translation from a high-level symbolic language into actual (machine-level) object code. Soon thereafter, the goal of *automatic programming*, i.e., automatic synthesis of programs from even higher-level specifications, was born and has been a dream for more than four decades. The idea that software engineering would be a better place if programmers could spend their time editing specifications, rather than trying to maintain optimized programs, is argued convincingly in a paper that tried to predict the future [3].

Pioneering efforts in the synthesis area around 1970 used theorem provers to verify the existence of an output for every input and then synthesized executable programs from the ingredients of these proofs [9, 17]. More encompassing development systems with synthesis included the 1970s PSI program synthesis system [10] and the 1980s Programmer's Apprentice project [22]. These ambitious systems tried to aid programmers by engaging in a dialog about the program to be developed, offering advice, keeping track of details, and synthesizing code. The systems made use of a significant knowledge base of the domains and template scenarios (so-called *clichés*) of the programs to be developed, and PSI also included a major natural-language component. In comparison, the abstract models one can define in Jennisys look much more like programs.

	type	*#branches*	*#Dafny*	*t (s)*
Binary Search Tree (see Figs. 0, 1 and 2)				
`IntSet.Singleton`	constr	1	2	5.4
`IntSet.Dupleton`	constr	2	5	15.5
`IntSet.Find`	method	4	17	64.8
Binary Heap (see Figs. 10 and 11)				
`BHeap.Singleton`	constr	1	2	5.2
`BHeap.Dupleton`	constr	2	5	19.3
`BHeap.Tripleton`	constr	3	13	61.6
`BHeap.Find`	method	3	17	51.9
Singly-Linked List (see Figs. 12 and 13)				
`List.Singleton`	constr	1	2	4.9
`List.Dupleton`	constr	1	2	5.2
`List.Elems`	method	2	6	14.6
`List.Get`	method	3	10	23.6
`List.Find`	method	2	7	16.8
`List.Size`	method	2	6	14.5
Doubly-Linked List (see Figs. 14 and 15)				
`DList.Singleton`	constr	1	2	4.8
`DList.Dupleton`	constr	1	2	5.0
`DList.Elems`	method	2	6	14.6
`DList.Get`	method	3	10	23.5
`DList.Find`	method	2	7	16.7
`DList.Size`	method	2	6	14.4
Simple Math Functions (see Figs. 16 and 17)				
`Math.Min2`	method	2	5	10.4
`Math.Min3Sum`	method	3	13	26.4
`Math.Min4`	method	4	23	45.7
`Math.Abs`	method	2	6	12.3

Table 0. Total synthesis time for each of the benchmarks (*t*), number of branches in the synthesized program (*#branches*), and the total number of times the verifier was invoked during synthesis (*#Dafny*).

Developed in the late 1980s, the comprehensive KIDS system provided a number of tools to support algorithm design and program transformations [27]. Besides major design decisions like semantically instantiating algorithm templates, the operations performed with KIDS are correctness-preserving transformations—*refinement steps*—that can take an algorithm description into an efficient program.

The Jennisys language is in many ways similar to one step of a refinement process, where Jennisys offers synthesis as one way of obtaining the refined program. Monahan has suggested defining components in three parts (`spec`/`abstr`/`impl`) [20], which is also what Jennisys does. Jennisys also shares in the vision of the language SETL [24], which sought to provide ways to first describe programs cleanly and then provide them with efficient data representations.

The construction of programs from examples is a powerful idea that has been explored from the 1970s. For example, THESYS synthesis system generated LISP programs [30] and QBE generated SQL queries [32]. Queries by Example became a competitive feature of the Paradox relational database system in the 1980s and 1990s, and techniques with similar goals are being explored in the context of spreadsheets today [12]. Jennisys also extrapolates programs from examples, but the examples are not supplied by users but are instead sample points from specifications supplied by users. An advantage of having specifications is that one can then verify the synthesized program, as opposed to just knowing it is correct for the examples provided.

Interest in program synthesis seems to be on the rise again, possibly in part due to the success of SMT solvers in program verification and other applications. Kuncak et al. are exploring features like generalized assignments in a mainstream programming language, backed up by automatic synthesis procedures [15]. The PINS [29] system takes a program and a template and synthesizes an inversion of the given program.

The technique of program sketching lets programmers supply some ingredients of a program (i.e. a program sketch) while a tool worries about the details to find a correct way to combine the ingredients [28]. The notion of correctness is taken from another correct (but presumably inefficient) implementation of the same program, that has to be supplied by the user. Storyboard programming [26] improves on that idea by letting the users draw a series of input/output examples instead of providing an alternative correct implementation (it still requires a sketch, though). Similarly, instead of a sketch, the Brahma tool [11, 13] takes a library of components to be used as building blocks and either an explicit set of input/output pairs or a specification describing the relation between inputs and outputs, and synthesizes a loop-free program (currently focused on bit-vector manipulations) from the given components. In comparison, Jennisys does not require any input from the user other than a specification in the form of pre- and post-conditions, it targets object-oriented programs with dynamic allocation, but is unable to synthesize as wide a class of programs as the storyboard programming.

The key advances in Jennisys are twofold. As for **what** our synthesis algorithm achieves, it is able to generate programs that create dynamic data structures with pointers, and it does this from an abstract (declarative) description of the operations (in the form of pre- and post-conditions) and a link to the concrete data structure. We have not seen this kind of synthesis done before in a class-based setting. As for **how** our synthesis algorithm makes a technical advance, it uses a combination of symbolic execution and unification to produce candidate program snippets, and then uses a detailed counterexample-producing program verifier to determine the applicable scope of each such snippet.

8. Conclusion

In this paper, we have contributed a language design that promotes writing down an abstract model of each component, gives control over the data structure used to implement a component, and opens the door for synthesis techniques to fill in the code. The paper also contributes a synthesis technique for the language, which operates in the context of an infinite state space, dynamic object allocation, and object references. Finally, the paper contributes a prototype implementation of the language and synthesis technique. The prototype is available as open source[5], and experiments with it are encouraging.

Still, much work lies ahead. We are interested in exploring the limits of the approach to coding based solely on synthesis from interface specifications and data-model invariants. A next step in this direction is to extend our synthesis engine to support mutating methods. Another is to explore different domains of programs that can be automatically synthesized purely from specifications. But, as we mentioned in the introduction, our vision is also to blend the idealistic synthesis-only approach with ways that give programmers the ability to supply code-generation hints, like in program sketching [28].

Acknowledgments

We would like to thank the various referees for their thoughtful comments on drafts of this paper.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[3] R. Balzer, T. E. Cheatham, Jr., and C. Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16(11):39–45, 1983.

[4] 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*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[5] 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. *J. STTT*, 7(3):212–232, 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[7] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[8] R. Goldfinger. The ibm type 705 autocoder. In *Papers presented at the February 7-9, 1956, joint ACM-AIEE-IRE western computer conference*, AIEE-IRE '56 (Western), pages 49–51, New York, NY, USA, 1956. ACM. doi: 10.1145/1455410.

[5] `http://boogie.codeplex.com`

1455427. URL http://doi.acm.org/10.1145/1455410.1455427.

[9] C. Green. Application of theorem proving to problem solving. In *IJCAI 1969*, pages 219–240. William Kaufmann, 1969.

[10] C. Green. The design of the PSI program synthesis system. In *ICSE*, pages 4–18. IEEE Computer Society, 1976.

[11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.

[12] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI 2011*, pages 317–328. ACM, 2011.

[13] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[14] C. B. Jones. *Systematic Software Development using VDM*. Series in Computer Science. Prentice-Hall International, second edition, 1990.

[15] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI 2010*, pages 316–329. ACM, 2010.

[16] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[17] Z. Manna and R. J. Waldinger. Towards automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[18] B. Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[19] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. Unifying execution of imperative and declarative code. In *ICSE*, pages 511–520, 2011.

[20] R. Monahan. *Data Refinement in Object-Oriented Verification*. PhD thesis, Dublin City University, 2010.

[21] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[22] C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, 1988.

[23] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, pages 552–576, 2010.

[24] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.

[25] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[26] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *ESEC/FSE 2011*, pages 289–299, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6.

[27] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9): 1024–1043, 1990.

[28] A. Solar-Lezama, L. Tancau, R. Bodík, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS 2006*, pages 404–415. ACM, 2006.

[29] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI 2011*, pages 492–503. ACM, 2011. ISBN 978-1-4503-0663-8.

[30] P. D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.

[31] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *EDCC*, pages 281–292, 2005.

[32] M. M. Zloof. Query by example. In *AFIPS National Computer Conference 1975*, pages 431–438. AFIPS Press, 1975.

A. Code Listings for the BHeap Example

```
// exactly the same as the interface for IntSet
interface BHeap {
  var elems: set[int]

  constructor Singleton(x: int)
    elems := {x}

  constructor Dupleton(a: int, b: int)
    requires a ≠ b
    elems := {a b}

  constructor Tripleton(x: int, y: int, z: int)
    requires x ≠ y ∧ y ≠ z ∧ z ≠ x
    elems := {x y z}

  method Find(n: int) returns (ret: bool)
    ret := n ∈ elems
}

datamodel BHeap {
  var data: int
  var left: BHeap
  var right: BHeap

  frame
    left * right

  invariant
    elems = {data} + (left ≠ null ? left.elems : {})
               + (right ≠ null ? right.elems : {})
    left ≠ null  ⟹ ∀ e  •  e in left.elems ⟹ e < data
    right ≠ null ⟹ ∀ e  •  e in right.elems ⟹ e < data
    left = null ⟹ right = null
    left ≠ null ∧ right = null ⟹ left.elems = {left.data}
}
```

Figure 10. Binary heap specified abstractly (declaratively) in Jennisys. Note that the interface specification is exactly the same as for the binary search tree (IntSet from Fig. 0); only the datamodel is different to impose a different concrete representation.

```
class BHeap {
  ghost var Repr: set<object>;
  ghost var elems: set<int>;

  var data: int;
  var left: BHeap;
  var right: BHeap;

  function Valid_repr(): bool
    reads *;
  {
    this in Repr ∧
    null ∉ Repr ∧
    (left ≠ null ⟹ left in Repr ∧
     left.Repr ≤ Repr ∧ this ∉ left.Repr) ∧
    (right ≠ null ⟹ right in Repr ∧
     right.Repr ≤ Repr ∧ this ∉ right.Repr)
  }

  function Valid_self(): bool
    reads *;
  {
    Valid_repr() ∧
    (elems = ({data} +
              (if left ≠ null then left.elems else {})) +
              (if right ≠ null then right.elems else {})) ∧
    (left ≠ null ⟹ (∀ e • e in left.elems ⟹ e < data)) ∧
    (right ≠ null ⟹ (∀ e • e in right.elems ⟹ e < data))
  }

  function Valid(): bool
    reads *;
    decreases Repr;
  {
    this.Valid_self() ∧
    (left ≠ null ⟹ left.Valid()) ∧
    (right ≠ null ⟹ right.Valid()) ∧
    (left ≠ null ⟹ left.Valid_self()) ∧
    (right ≠ null ⟹ right.Valid_self()) ∧
    (left ≠ null ∧
     left.left ≠ null ⟹ left.left.Valid_self()) ∧
    (left ≠ null ∧
     left.right ≠ null ⟹ left.right.Valid_self()) ∧
    (right ≠ null ∧
     right.left ≠ null ⟹ right.left.Valid_self()) ∧
    (right ≠ null ∧
     right.right ≠ null ⟹ right.right.Valid_self())
  }
```

```
method Singleton(x: int)
  modifies this;
  ensures fresh(Repr - {this});
  ensures Valid();
  ensures elems = {x};
{
  this.data := x;
  this.elems := {x};
  this.left := null;
  this.right := null;
  // repr stuff
  this.Repr := {this};
}

method Dupleton(a: int, b: int)
  modifies this;
  requires a ≠ b;
  ensures fresh(Repr - {this});
  ensures Valid();
  ensures elems = {a, b};
{
  if (b < a) {
    var gensym71 := new BHeap;
    var gensym73 := new BHeap;
    this.data := a;
    this.elems := {b, a};
    this.left := gensym73;
    this.right := gensym71;
    gensym71.data := b;
    gensym71.elems := {b};
    gensym71.left := null;
    gensym71.right := null;
    gensym73.data := b;
    gensym73.elems := {b};
    gensym73.left := null;
    gensym73.right := null;
    // repr stuff
    gensym71.Repr := {gensym71};
    gensym73.Repr := {gensym73};
    this.Repr := ({this} + {gensym73}) + {gensym71};
  } else {
    var gensym71 := new BHeap;
    var gensym73 := new BHeap;
    this.data := b;
    this.elems := {a, b};
    this.left := gensym73;
    this.right := gensym71;
    gensym71.data := a;
    gensym71.elems := {a};
    gensym71.left := null;
    gensym71.right := null;
    gensym73.data := a;
    gensym73.elems := {a};
    gensym73.left := null;
    gensym73.right := null;
    // repr stuff
    gensym71.Repr := {gensym71};
    gensym73.Repr := {gensym73};
    this.Repr := ({this} + {gensym73}) + {gensym71};
  }}

method Tripleton(x: int, y: int, z: int)
  modifies this;
  requires x ≠ y ∧ y ≠ z ∧ z ≠ x;
  ensures fresh(Repr - {this});
  ensures Valid();
  ensures elems = {x, y, z};
{
  if (z < y ∧ x < y) {
    var gensym75 := new BHeap;
    var gensym77 := new BHeap;
    this.data := y;
    this.elems := {x, z, y};
    this.left := gensym77;
    this.right := gensym75;
    gensym75.data := x;
    gensym75.elems := {x};
    gensym75.left := null;
    gensym75.right := null;
    gensym77.data := z;
    gensym77.elems := {z};
    gensym77.left := null;
    gensym77.right := null;
    // repr stuff
    gensym75.Repr := {gensym75};
    gensym77.Repr := {gensym77};
    this.Repr := ({this} + {gensym77}) + {gensym75};
  } else {
    if (x < z) {
      var gensym75 := new BHeap;
      var gensym77 := new BHeap;
      this.data := z;
      this.elems := {x, y, z};
      this.left := gensym77;
      this.right := gensym75;
      gensym75.data := x;
      gensym75.elems := {x};
      gensym75.left := null;
      gensym75.right := null;
      gensym77.data := y;
      gensym77.elems := {y};
      gensym77.left := null;
      gensym77.right := null;
      // repr stuff
      gensym75.Repr := {gensym75};
      gensym77.Repr := {gensym77};
      this.Repr := ({this} + {gensym77}) + {gensym75};
    } else {
      var gensym75 := new BHeap;
      var gensym77 := new BHeap;
      this.data := x;
      this.elems := {z, y, x};
      this.left := gensym77;
      this.right := gensym75;
      gensym75.data := y;
      gensym75.elems := {y};
      gensym75.left := null;
      gensym75.right := null;
      gensym77.data := z;
      gensym77.elems := {z};
```

```
        gensym77.left := null;
        gensym77.right := null;
        // repr stuff
        gensym75.Repr := {gensym75};
        gensym77.Repr := {gensym77};
        this.Repr := ({this} + {gensym77}) + {gensym75};
      }
    }
  }

  method Find(n: int) returns (ret: bool)
    requires Valid();
    ensures fresh(Repr - old(Repr));
    ensures Valid();
    ensures ret = (n in elems);
    decreases Repr;
  {
    if (this.left = null) {
      ret := n = this.data;
    } else {
      if (this.right ≠ null) {
        var x_10 := this.left.Find(n);
        var x_11 := this.right.Find(n);
        ret := (n = this.data ∨ x_10) ∨ x_11;
      } else {
        var x_12 := this.left.Find(n);
        ret := n = this.data ∨ x_12;
      }
    }
  }
}
```

Figure 11. Imperative Dafny code that Jennisys synthesizes for the abstract `BHeap` program from Fig. 10.

B. Code Listings for the List Example

```
interface List[T] {
  var list: seq[T]

  invariant
    |list| > 0

  constructor Singleton(t: T)
    list := [t]

  constructor Dupleton(p: T, q: T)
    list := [p q]

  method Elems() returns (ret: seq[T])
    ret := list

  method Get(idx: int) returns (ret: T)
    requires 0 ≤ idx ∧ idx < |list|
    ret := list[idx]

  method Find(n: T) returns (ret: bool)
    ret := n ∈ list

  method Size() returns (ret: int)
    ret := |list|
}

datamodel List[T] {
  var data: T
  var next: List[T]

  frame next

  invariant
    next = null  ⟹ list = [data]
    next ≠ null ⟹ list = [data] + next.list
}
```

Figure 12. Singly-linked list specified abstractly (declaratively) in Jennisys.

```
class List<T> {
  ghost var Repr: set<object>;
  ghost var list: seq<T>;

  var data: T;
  var next: List<T>;

  function Valid_repr(): bool
    reads *;
  {
    this in Repr ∧
    null ∉ Repr ∧
    (next ≠ null ⟹ next in Repr ∧
     next.Repr ≤ Repr ∧ this ∉ next.Repr)
  }

  function Valid_self(): bool
    reads *;
  {
    Valid_repr() ∧
    (next = null ⟺ list = [data] ∧ list[0] = data) ∧
    (next ≠ null ⟹ list = [data] + next.list) ∧
    (|list| > 0)
  }

  function Valid(): bool
    reads *;
    decreases Repr;
  {
    this.Valid_self() ∧
    (next ≠ null ⟹ next.Valid()) ∧
    (next ≠ null ⟹ next.Valid_self() ∧
       next.next ≠ null ⟹ next.next.Valid_self())
  }

  method Singleton(t: T)
    modifies this;
    ensures fresh(Repr - {this});
    ensures Valid();
    ensures list = [t];
  {
    this.data := t;
    this.list := [t];
    this.next := null;
    // repr stuff
    this.Repr := {this};
  }
```

```
method Dupleton(p: T, q: T)
  modifies this;
  ensures fresh(Repr - {this});
  ensures Valid();
  ensures list = [p, q];
{
  var gensym71 := new Node<T>;
  gensym71.data := q;
  gensym71.list := [q];
  gensym71.next := null;
  this.data := p;
  this.list := [p, q];
  this.next := gensym71;
  // repr stuff
  gensym71.Repr := {gensym71};
  this.Repr := {this} + this.next.Repr;
}

method Elems() returns (ret: seq<T>)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = list;
  decreases Repr;
{
  if (this.next = null) {
    ret := [this.data];
  } else {
    var x_7 := this.next.Elems();
    ret := [this.data] + x_7;
  }
}

method Get(idx: int) returns (ret: T)
  requires Valid();
  requires 0 ≤ idx;
  requires idx < |list|;
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = list[idx];
  decreases Repr;
{
  if (this.next = null) {
    ret := this.data;
  } else {
    if (idx = 0) {
      ret := this.data;
    } else {
      var x_6 := this.next.Get(idx - 1);
      ret := x_6;
    }
  }
}
```

```
method Find(n: T) returns (ret: bool)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = (n in list);
  decreases Repr;
{
  if (this.next = null) {
    ret := n = this.data;
  } else {
    var x_5 := this.next.Find(n);
    ret := n = this.data ∨ x_5;
  }
}

method Size() returns (ret: int)
  requires Valid();
  ensures fresh(Repr - old(Repr));
  ensures Valid();
  ensures ret = |list|;
  decreases Repr;
{
  if (this.next = null) {
    ret := 1;
  } else {
    var x_8 := this.next.Size();
    ret := 1 + x_8;
  }
}
}
```

Figure 13. Imperative Dafny code that Jennisys synthesizes for the abstract `List` program from Fig. 12.

C. Code Listings for the DList Example

```
// exactly the same as the interface for List
interface DList[T] {
  var list: seq[T]

  invariant
    |list| > 0

  constructor Init(t: T)
    list := [t]

  constructor Double(p: T, q: T)
    list := [p q]

  method Elems() returns (ret: seq[T])
    ret := list

  method Get(idx: int) returns (ret: T)
    requires 0 ≤ idx ∧ idx < |list|
    ret := list[idx]

  method Find(n: T) returns (ret: bool)
    ret := n ∈ list

  method Size() returns (ret: int)
    ret := |list|
}

datamodel DList[T] {
  var data: T
  var next: DList[T]
  var prev: DList[T]

  frame
    next

  invariant
    next = null ⟹ list = [data]
    next ≠ null ⟹ (list = [data] + next.list
                    ∧  next.prev = this)
    prev ≠ null ⟹ prev.next = this
}
```

Figure 14. Doubly-linked list specified abstractly (declaratively) in Jennisys. Note that the interface specification is exactly the same as for the singly-linked list (`List` from Fig. 12); only the datamodel is different to impose a different concrete representation.

```
class DList<T> {
  ghost var Repr: set<object>;
  ghost var list: seq<T>;

  var data: T;
  var next: DList<T>;
  var prev: DList<T>;

  function Valid_repr(): bool
    reads *;
  {
    this in Repr ∧
    null ∉ Repr ∧
    (next ≠ null ⟹ next in Repr
                   ∧ next.Repr ≤ Repr
                   ∧ this ∉ next.Repr)
  }

  function Valid_self(): bool
    reads *;
  {
    Valid_repr() ∧
    (next = null ⟹ (list = [data] ∧ list[0] = data)
                   ∧ |list| = 1) ∧
    (next ≠ null ⟹ list = [data] + next.list
                   ∧ next.prev = this) ∧
    (prev ≠ null ⟹ prev.next = this) ∧
    (|list| > 0)
  }

  function Valid(): bool
    reads *;
    decreases Repr;
  {
    this.Valid_self() ∧
    (next ≠ null ⟹ next.Valid()) ∧
    (next ≠ null ⟹ next.Valid_self()) ∧
    (next ≠ null ∧
     next.next ≠ null ⟹ next.next.Valid_self())
  }

  method Singleton(t: T)
    modifies this;
    ensures fresh(Repr - {this});
    ensures Valid();
    ensures list = [t];
    ensures list[0] = t;
    ensures |list| = 1;
  {
    this.data := t;
    this.list := [t];
    this.next := null;
    this.prev := null;
    // repr stuff
    this.Repr := {this};
  }
```

```
method Double(p: T, q: T)
  modifies this;
  ensures fresh(Repr - {this});
  ensures Valid();
  ensures list = [p, q];
  ensures list[0] = p;
  ensures list[1] = q;
  ensures |list| = 2;
{
  var gensym71 := new DList<T>;
  this.data := p;
  this.list := [p, q];
  this.next := gensym71;
  this.prev := null;
  gensym71.data := q;
  gensym71.list := [q];
  gensym71.next := null;
  gensym71.prev := this;
  // repr stuff
  this.Repr := {this} + {gensym71};
  gensym71.Repr := {gensym71};
}
```

```
  method Elems() returns (ret: seq<T>)
  {
    // same as List.Elems
  }

  method Get(idx: int) returns (ret: T)
  {
    // same as List.Get
  }

  method Find(n: T) returns (ret: bool)
  {
    // same as List.Find
  }

  method Size() returns (ret: int)
  {
    // same as List.Size
  }
}
```

Figure 15. Imperative Dafny code that Jennisys synthesizes for the abstract `DList` program from Fig. 14.

D. Code Listings for the Math Example

```
interface Math {
  method Min2(a: int, b: int) returns (ret: int)
    ensures a < b ⟹ ret = a
    ensures a ≥ b ⟹ ret = b

  method Min3Sum(a: int, b: int, c: int)
      returns (ret: int)
    ensures ret ∈ {a+b a+c b+c}
    ensures ∀ x • x ∈ {a+b a+c b+c} ⟹ ret ≤ x

  method Min4(a: int, b: int, c: int, d: int)
      returns (ret: int)
    ensures ret ∈ {a b c d}
    ensures ∀ x • x ∈ {a b c d} ⟹ ret ≤ x

  method Abs(a: int) returns (ret: int)
    ensures ret ∈ {a (-a)} ∧ ret ≥ 0
}

datamodel Math {}
```

Figure 16. Several math operations specified abstractly (declaratively) in Jennisys.

```
class Math {
  ghost var Repr: set<object>;

  function Valid_repr(): bool reads *;
  {
    this in Repr ∧ null ∉ Repr
  }

  function Valid_self(): bool reads *;
  {
    Valid_repr()
  }

  function Valid(): bool reads *;
  {
    this.Valid_self()
  }

  method Min2(a: int, b: int) returns (ret: int)
    requires Valid();
    ensures fresh(Repr - old(Repr));
    ensures Valid();
    ensures a < b ⟹ ret = a;
    ensures a ≥ b ⟹ ret = b;
  {
    if (a < b) {
      ret := a;
    } else {
      ret := b;
    }
  }

  method Min3Sum(a: int, b: int, c: int)
      returns (ret: int)
    requires Valid();
    ensures fresh(Repr - old(Repr));
    ensures Valid();
    ensures ret in {a+b, a+c, b+c};
    ensures ret ≤ a+b ∧ ret ≤ a+c ∧ ret ≤ b+c;
  {
    if (a+b ≤ a+c ∧ a+b ≤ b+c) {
      ret := a+b;
    } else {
      if (b+c ≤ a+b ∧ b+c ≤ a+c) {
        ret := b+c;
      } else {
        ret := a+c;
      }
    }
  }

  method Min4(a: int, b: int, c: int, d: int)
      returns (ret: int)
    requires Valid();
    ensures fresh(Repr - old(Repr));
    ensures Valid();
    ensures ret in {a, b, c, d};
    ensures ret ≤ a ∧ ret ≤ b ∧ ret ≤ c ∧ ret ≤ d;
  {
    if ((a ≤ b ∧ a ≤ c) ∧ a ≤ d) {
      ret := a;
    } else {
      if (d ≤ b ∧ d ≤ c) {
        ret := d;
      } else {
        if (c ≤ b) {
          ret := c;
        } else {
          ret := b;
        }
      }
    }
  }

  method Abs(a: int) returns (ret: int)
    requires Valid();
    ensures fresh(Repr - old(Repr));
    ensures Valid();
    ensures ret in {a, -a} ∧ ret ≥ 0;
  {
    if (-a ≥ 0) {
      ret := -a;
    } else {
      ret := a;
    }
  }
}
```

Figure 17. Imperative Dafny code that Jennisys synthesizes for the abstract Math program from Fig. 16.