Specification Construction Using SMT Solvers

Paul C. Attie, Fadi Zaraket, Mohamad Noureddine, and Farah El-Hariri

American University of Beirut
Email: {pa07,fz11,man17,fsa20}@aub.edu.lb

Abstract. The problem of writing a *functional specification* which reflects the intentions of the developer has long been recognized as fundamental. We propose a method to construct (synthesize) a functional specification for a terminating program which takes a single input and produces a single output. The specification consists of a precondition and a postcondition, expressed as first-order logic wff's, including quantifiers. The user provides an underlying *type theory* for the variables used to write the specification, and also answers a series of queries, which make precise her intentions. Each query presents an example of the relevant data: input for precondition, input-output pair for postcondition. The developer states whether the precondition (postcondition, respectively) being written should hold or not (according to the users intentions). By grouping examples into *equivalence classes*, we reduce the number of queries to finite. After one query for each equivalence class has been answered, the method outputs a formula for the precondition (postcondition, respectively) that the user intends. By using various pruning strategies, and by constructing a formula using simpler formulae that have been previously constructed, we further reduce the number of queries to the point that our method is practical. Our method constructed, in reasonable time, accurate specifications for array search, binary heap, binary search, linked list, and trie.

1 Introduction

The task of writing a (formal) specification has long been recognized as crucial, cf. Brooks [1], and is addressed by many requirements elicitation methods, e.g., [2–6], The derivation of programs and their correctness proofs from formal specifications has been advocated by Dijkstra, Hoare, and others. Recent program synthesis techniques require formal specifications [7, 8] as input.

We present a method for writing a formal functional specification for a transformational, terminating program $\mathcal{P}$, which takes a single input and produces a single output. We assume that the user has an informal idea of how $\mathcal{P}$ should behave, and that (1) given an input, the user can judge whether the input is irrelevant (need not be processed correctly) or relevant (must be processed correctly), and (2) given an output corresponding to the input, the user can judge whether the output is correct w.r.t the input, or not. For example, for binary search, an ordered array is relevant and an unordered one is not. A correct output is the location of the sought value, or an indication of its absence. This assumption on user behavior is key to program synthesis techniques such as [9–11], which generate programs from user provided inputs and outputs.

Code snippet program synthesis techniques require the user to provide the specification as a starting point [8,12,13]. SQL query synthesis techniques require the user to provide a set of inputs and outputs [9]. Techniques for synthesizing recursive programs [10] and mobile automation scripts [11] additionally require a set of program building blocks. The techniques [11, 13] require in addition an interactive construction process where the user provides feedback. Techniques that build loop free programs [14] require basic components and user answers to SMT solver generated queries. Except for the techniques that start from a specification, the user is left with no notion of correctness except her judgment.

This paper presents a *novel method* that constructs an *accurate specification* in first order logic (including quantifiers) and that only requires the user to (1) provide a set of variable declarations that forms a type theory, (2) describe the variables as index, bound, or data variables w.r.t. array variables in a simple grammar, and (3) judge a sequence of variable valuations generated using an SMT solver. To our knowledge, our technique is the first to produce specifications with *quantified formulae*. Our method guarantees accurate specifications provided that the user makes all judgments correctly. Experiments with undergrad and graduate computer engineering students produced array search, binary heap, binary search, linked list, and trie specifications in reasonable time. These were more accurate than manually written specifications.

Overview. A single input-output pair is a *behavior*. We model the user's intuition as a (possibly infinite) set of *judgments* over inputs and behaviors, and we formalize this intuition as a specification $\mathcal{S}$, i.e., a precondition-postcondition pair (P, Q), expressed as first-order logic formulae. A precondition is evaluated over an input, and a postcondition is evaluated over a behavior. We produce a specification which is *accurate*, i.e., reflects the user's intentions:

1. The precondition holds for an input iff the user judges the input relevant.
2. The postcondition holds for a behavior if the user either judges the input irrelevant (dont care) or judges the output correct w.r.t. the input.

Figure 1(a) shows this relationship, for an accurate specification, between the user judgments of the input and output, and the required values for the precondition and postcondition, where '?' indicates a "dont care," i.e., the postcondition can be true or false. We use "accurate" rather than "correct" so as to not overload the term "correct", which usually means correct w.r.t. a specification.

We motivate with an example: sorting two integers x and y. The user judges all possible input values as relevant, and so *true* is an accurate precondition. A formula for the accurate postcondition is $Q \triangleq [(x_i = x_o \wedge y_i = y_o) \vee (x_i = y_o \wedge y_i = x_o)] \wedge x_o \leqslant y_o$, i.e., the final values of x, y are an ordered permutation of the initial values. Subscripts of i, o indicate the initial and final values, respectively. Consider the inaccurate $Q1 \triangleq (x_i = x_o \wedge y_i = y_o) \wedge x_o \leqslant y_o$. For $x_i = 1$, $y_i = 2$, $x_o = 1$, $y_o = 2$, we have $Q = tt$ and $Q1 = tt$, and for $x_i = 2$, $y_i = 1$, $x_o = 1$, $y_o = 2$, we have $Q = tt$ and $Q1 = \mathit{ff}$, where tt and ff denote true and false respectively. This example shows that an inaccurate postcondition may still match an accurate one in some cases, so sporadic checking of some behaviors

	Input is relevant	Input is irrelevant
Output is correct	P is true / Q is true	P is false / Q is ?
Output is incorrect	P is true / Q is false	P is false / Q is ?

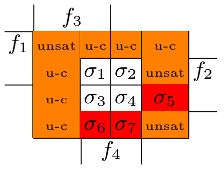
(a)

$$f_1 \triangleq 0 = i$$
$$f_2 \triangleq 0 \leqslant i$$
$$f_3 \triangleq i = |a| - 1$$
$$f_4 \triangleq i \leqslant |a| - 1$$

(b)

	f_3	f_3	$\neg f_3$	$\neg f_3$	
f_1	F_1	F_2	F_3	F_4	$\neg f_2$
f_1	F_5	F_6	F_7	F_8	f_2
$\neg f_1$	F_9	F_{10}	F_{11}	F_{12}	f_2
$\neg f_1$	F_{13}	F_{14}	F_{15}	F_{16}	$\neg f_2$
	$\neg f_4$	f_4	f_4	$\neg f_4$	

(c)



(d)

unsat valuation formula	conflict	similar
$F_1 = f_1 \wedge \neg f_2 \wedge f_3 \wedge \neg f_4$	$f_1, \neg f_2$	F_2, F_3, F_4
	$f_3, \neg f_4$	F_5, F_9, F_{13}
$F_8 = f_1 \wedge f_2 \wedge \neg f_3 \wedge \neg f_4$	$f_1, \neg f_4$	F_1, F_4, F_5
$F_{16} = \neg f_1 \wedge \neg f_2 \wedge \neg f_3 \wedge \neg f_4$	all	

(e)

valuation formula	i	a	
$F_6 = f_1 \wedge f_2 \wedge f_3 \wedge f_4$	0	$\langle 0 \rangle$	σ_1
$F_7 = f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4$	0	$\langle 0, 0 \rangle$	σ_2
$F_{10} = \neg f_1 \wedge f_2 \wedge f_3 \wedge f_4$	1	$\langle 0, 0 \rangle$	σ_3
$F_{11} = \neg f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4$	1	$\langle 0, 0, 0 \rangle$	σ_4
$F_{12} = \neg f_1 \wedge f_2 \wedge \neg f_3 \wedge \neg f_4$	3	$\langle 0, 0, 0 \rangle$	σ_5
$F_{14} = \neg f_1 \wedge \neg f_2 \wedge f_3 \wedge f_4$	-1	$\langle\rangle$	σ_6
$F_{15} = \neg f_1 \wedge \neg f_2 \wedge \neg f_3 \wedge f_4$	-1	$\langle 0 \rangle$	σ_7

(f)

Fig. 1: (a) User judgment and pre/post-condition values in an accurate specification. (b) formulae of `isvalid` from the `binary-heap` example (Figure 2). (c) Karnaugh map of 16 valuation formulae. (d) classification of $F_1 - F_{16}$. (e) unsatisfiable formulae. (f) reduction to 7 judgments and 10 solver calls (6 and 8 with pruning-Section 4.4).

is insufficient. However, we cannot check the infinite number (in general) of all behaviors. Hence we partition the set of all behaviors into a finite number of equivalence classes, and check one representative from each equivalence class.

The key idea is to use a theory (set of first order formulae) as "primitive building blocks" to write the formulae for precondition P and postcondition Q. For the above example, the relevant theory is the formulae expressing $=$ and $<$ between all pairs of x_i, x_o, y_i, y_o. This results in 12 formulae. Each possible assignment of values to these 12 formulae generates an equivalence class of the set of behaviors. Thus we reduce the problem from considering an infinite number of cases to considering a finite but large (2^{12}) number of cases. We show in the sequel how to reduce this to a reasonable number, using various pruning tactics.

Consider next the validity of an array index. Figure 1(b) shows a theory of four formulae that compare index i to 0 and $|a| - 1$, the bounds of array a ($|a|$ denotes the size of a). There are 16 Boolean valuations of these four formulae, as shown in the Karnaugh map in Figure 1(c). Each valuation determines an equivalence class of behaviors, namely those for which the formulae have the assigned values. We convert each valuation into the corresponding "valuation formula" in the obvious manner, as shown. This formula characterizes the same equivalence class of behaviors, namely those which satisfy it. We take a representative "valuation behavior" for each equivalence class and determine which of the valuation behaviors are correct. We then take the disjunction of the corresponding valuation formulae. Here, the assignments $\sigma_1, \sigma_2, \sigma_3$, and σ_4 (Figure 1(f)) are correct behaviors that correspond to $F_6 = f_1 \wedge f_2 \wedge f_3 \wedge f_4$, $F_7 = f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4$, $F_{10} = \neg f_1 \wedge f_2 \wedge f_3 \wedge f_4$, and $F_{11} = \neg f_1 \wedge f_2 \wedge \neg f_3 \wedge f_4$, respectively. The disjunction $F_6 \vee F_7 \vee F_{10} \vee F_{11}$ simplifies to $f_2 \wedge f_4 = 0 \leqslant i \wedge i \leqslant |a| - 1$ which is accurately the specification.

To convert the above into an effective procedure, we need to

1. generate a representative behavior from each valuation formula, and

2. classify each representative behavior as correct or incorrect.

We deal with (1) by submitting the valuation formula to an *SMT solver*, which, if satisfiable, will return a satisfying assignment. Since the assignment is over all the input and output values, we can interpret it as a behavior. We deal with (2) by invoking the *user as an oracle*: the user will interact with the algorithm, and will judge in turn the behavior of each valuation formula as correct or incorrect.

To turn the effective procedure into a practical and useful tool, we have to:

1. Give pruning techniques (Section 4), since the number of valuation formulae is exponential in the theory size.
2. Avoid using large and complex theories, as this makes the number of valuation formulae impractical. Thus Section 6 presents a means for constructing theories *incrementally and hierarchically* using simpler specifications as building blocks. This also helps us generate succinct and readable formulae.
3. The above method uses only Boolean operators to construct formulae. Section 6.1 presents a method that also uses quantifiers.
4. We must restrict the SMT solver queries to those which state-of-the-art solvers can decide. Section 5 shows how we do this.

Our method constructs a specification that can be used (1) early in the software development cycle to synthesize an implementation of the intended program, (2) later to formally verify the correctness of an existing implementation, and (3) periodically by using the generated test inputs that satisfy the precondition as test cases against the implementation. It has the following advantages.

Advantage 1: Our method helps users with basic logic skills write accurate specifications. The user must only provide data types and intended use of the variables as index, bound, or data variables, and judge whether an assignment of those variables is correct or not. The program that we are writing a specification for can be represented as a Java method. The variables used in the specification are given as parameters of the method. The constructed specification is in first order logic; e.g., expressible in Java Modeling Language and Alloy.

Advantage 2: Our method can build specifications that are in a decidable fragment of first-order logic. This is very useful for formal verification and synthesis of programs, and also for assertion-based testing techniques [15–17].

Advantage 3: The SMT solver generated behaviors, along with the user judgments, can be used as test cases in a unit testing environment such as JUnit, for the end result program. The resulting test suite has high coverage since each test case represents a unique equivalence class. Reciprocally, existing test suites can be used to answer user queries, so the user is only burdened with queries whose equivalence classes are not represented in the test suite.

Advantage 4: Like interactive techniques that construct proofs [18, 19] and bridge the gap between verification engineers and automated proof techniques, our method also bridges the gap between programmers and logic specifications.

2 Motivating Examples

The example in Figure 2 shows the construction of a binary heap specification. The user starts by declaring the array `a` that constitutes the binary heap. The user declares `isvalid` to specify a valid index in the array `a`. The type

```
input: class binary-heap {
 int [] a;
 constants {1,2};

 theory isvalid (int i) {
   grammar {(a,i,index);}}

 theory isleft (int i, int left) {
  predicate isvalid;
  grammar {(a,i,index),(a,left,index);}}

 theory isright (int i, int right) {
  predicate isvalid;
  grammar {(a,i,index),(a,right,index);}}

 theory isnotheap {
  // quantified variables i and j will
  //   be injected as indices of a
  predicate isleft, isright, isvalid;
  grammar {(a,a,<=);}}}
```

```
query: i=-1,j=1,a=[0 0]
user answer: not correct because of i
...
query: i=0,j=1,a=[2 1]
user answer: correct (isnotheap)
...
```

```
Specifications:
isvalid: (0<= i) & (i<= a.size - 1)
isleft: isvalid(i)& isvalid(left)& left=2*i+1
isright:isvalid(i)& isvalid(right)& right=2*i+2
isnotheap: exists i,j. isvalid(i)& isvalid(j)&
 (isleft(i,j)| isright(i,j))& not (a[i]<=a[j])
```

Fig. 2: Binary heap theories, ConstructFormula queries, and specifications

theory of `isvalid` includes the members of the binary heap class and the integer `i`. The grammar declares `i` as index of array `a`. The user runs our method, answers 6 queries, and gets the formula for `isvalid`, as shown in Figure 1. The user then declares the `isleft` predicate with `i` and `left` as indices of array `a` and declares the intent to use `isvalid` from the previous hierarchy level. The user answers 8 queries and gets the specification that constrains `i` and `left` to be valid indices of `a` and `left` to be `2*i+1`. Similarly the user constructs the `isright` predicate. Finally, the user declares a theory for the binary heap specification, and declares the intent to use the predicates from previous levels in the hierarchy. The user answers up to 14 queries marking the assignments that are not heaps as "correct" and those that are heaps as "incorrect" and gets the `isnotheap` specification. Figure 2 also shows the constructed specifications and example queries that the user answered appropriately for `isnotheap`. The `isheap` specification is the negation. The user constructed the negated formula since she is aware that the method generates an existentially quantified formula.

Similarly, we produce the specification $(0 \leqslant \ell \leqslant r \leqslant |a|-1) \wedge \big((rv \neq -1 \wedge e = a[rv] \wedge eina(a,\ell,r,e)) \vee (rv = -1 \wedge \neg eina(a,\ell,r,e))\big)$ for searching an array, where $eina(a,\ell,r,e) \triangleq \exists i.isvalid(\ell) \wedge isvalid(r) \wedge \ell \leqslant r \wedge \ell \leqslant i \wedge i \leqslant r \wedge a[i] = e$. $eina(a,\ell,r,e)$ was produced at the previous level in the hierarchy. ℓ, r are the left and right bounds of the section of a to be searched. rv is the index of a location of e within this section, or -1 if e is not present.

3 The Specification and Formula Construction Problems

We write specifications for terminating sequential programs with a fixed set $x_1, \ldots, x_n$ of program variables, which take values from universes $U_1, \ldots, U_n$, respectively. We use many-sorted first-order logic.We assume the usual definitions for first-order language, well-formed formula (we just say "formula"), first-order structure, truth in a structure, etc. The variables of our first-order language $\mathcal{L}$ include $x_1^i, \ldots, x_n^i$, which represent, respectively, the initial values of $x_1, \ldots, x_n$, and $x_1^o, \ldots, x_n^o$, which represent, respectively, the final values of $x_1, \ldots, x_n$. These are the only variables in $\mathcal{L}$ which occur free. All other variables occur bound.

Definition 1 (Input state, output state, behavior). *An* input state $\sigma_i : \langle x^i_1, \ldots, x^i_n \rangle \to U_1 \times \cdots \times U_n$ *is an assignment that maps each* x^i_j*,* $j \in [1:n]$*, to a value in* U_j*, and likewise for an* output state $\sigma_o : \langle x^o_1, \ldots, x^o_n \rangle \to U_1 \times \cdots \times U_n$*. A* behavior $\sigma = (\sigma_i, \sigma_o)$ *is a pair consisting of an input and an output state.*

A formula is interpreted in a many-sorted structure (I, σ), with universes $U_1, \ldots, U_n$. I provides the interpretation for the functions and predicate symbols. σ is a behavior, and provides the interpretation for the $x^i_1, \ldots, x^i_n, x^o_1, \ldots, x^o_n$.

Definition 2 (Satisfaction of a formula). *Let* f *be a formula. Write* $(I, \sigma) \models f$ *iff* f *is true in the structure* (I, σ)*, according to usual Tarskian semantics. We omit* I*, as it is fixed, and write* $\sigma \models f$*. Write* $\sigma.f$ *for the truth value of* f *in* (I, σ)*.* $[f]$ *denotes* $\{\sigma \mid \sigma \models f\}$*, i.e., the set of behaviors where* f *holds.*

Definition 3 (Specification, satisfaction of a specification). *A* specification $\mathcal{S} = (P, Q)$*, consists of:* P *which represents the precondition, and is restricted to contain only* $x^i_1, \ldots, x^i_n$*, and* Q*, which represents the postcondition. A behavior* $\sigma = (\sigma_i, \sigma_o)$ satisfies *a specification* $\mathcal{S} = (P, Q)$ *iff* $\sigma.(P \Rightarrow Q) = true$*. Write* $\sigma \models \mathcal{S}$ *in this case, and* $\sigma \not\models \mathcal{S}$ *otherwise. Also define* $[\mathcal{S}] \triangleq \{\sigma \mid \sigma \models \mathcal{S}\}$*.*

Let Σ be the set of behaviors. We partition Σ into:

- $\texttt{good} = \{\sigma \mid \sigma.P = \mathit{true}$ and $\sigma.Q = \mathit{true}\}$, the set of good behaviors; the precondition holds before and the postcondition holds after.
- $\texttt{bad} = \{\sigma \mid \sigma.P = \mathit{true}$ and $\sigma.Q = \mathit{false}\}$, the set of bad behaviors; the precondition holds before and the postcondition does not hold after.
- $\texttt{dontCare} = \{\sigma \mid \sigma.P = \mathit{false}\}$, the set of don't care behaviors; the precondition does not hold before, and the postcondition is unrestricted.

A partition $(\texttt{good}, \texttt{bad}, \texttt{dontCare})$ of Σ is *feasible* iff (1) for every input state σ_i, there do not exist two output states $\sigma_o, \sigma_{o'}$ such that $(\sigma_i, \sigma_o) \in \texttt{good} \cup \texttt{bad}$ and $(\sigma_i, \sigma_{o'}) \in \texttt{dontCare}$); and (2) for every input state σ_i there exists an output state σ_o such that $(\sigma_i, \sigma_o) \in \texttt{good} \cup \texttt{dontCare}$). Clause (1) means that the precondition is not both true ($(\sigma_i, \sigma_o) \in \texttt{good} \cup \texttt{bad}$) and false ($(\sigma_i, \sigma_{o'}) \in \texttt{dontCare}$) when evaluated on input σ_i. Clause (2) means that for every input there is at least one acceptable output. In the sequel, we consider only feasible partitions of Σ.

We construct a specification that accurately reflects the intentions of the user, which are given by a feasible partition $(\texttt{good}, \texttt{bad}, \texttt{dontCare})$ of Σ. The user provides information about this partition by answering queries about the location of given example inputs and behaviors within this partition.

Definition 4 (Specification construction problem). *A specification* $\mathcal{S}$ *is* accurate *w.r.t. a feasible partition* $(\texttt{good}, \texttt{bad}, \texttt{dontCare})$ *of* Σ *iff* $[\mathcal{S}] = \texttt{good} \cup \texttt{dontCare}$*. The* specification construction problem *is to write a specification that is accurate w.r.t. a given* $(\texttt{good}, \texttt{bad}, \texttt{dontCare})$*.*

Definition 5 (Formula construction problem). *A wff* $\mathcal{F}$ *is* accurate *with respect to a partition* $(\mathit{vtt}, \mathit{vff})$ *of* Σ *iff* $[\mathcal{F}] = \mathit{vtt}$*. The* formula construction problem *is to write a wff that is accurate w.r.t. a given* $(\mathit{vtt}, \mathit{vff})$*.*

We reduce specification construction to formula construction, as follows. Construct a formula P that is accurate w.r.t. $(\mathtt{good} \cup \mathtt{bad}, \mathtt{dontCare})$. Also construct a formula Q that is accurate w.r.t. $(\mathtt{good} \cup \phi, \mathtt{bad} \cup \psi)$, where (ϕ, ψ) is an arbitrary partition of $\mathtt{dontCare}$, which can be chosen for convenience of expressing Q. From the definitions of $\sigma \models \mathcal{S}$ and $[\mathcal{S}]$ given above, we obtain $[\mathcal{S}] = \mathtt{good} \cup \mathtt{dontCare}$, and so $\mathcal{S}$ is accurate with respect to $(\mathtt{good}, \mathtt{bad}, \mathtt{dontCare})$.

4 Construction of Formulae via Boolean Operations

We present below algorithm ConstructFormula, which constructs a formula in disjunctive normal form, where the literals are assignments to "more elementary" formulae, drawn from either the underlying type theory τ, or a "vocabulary" ν, as discussed below. These are theories (sets of wffs) in first order logic. τ is the type theory for the program variables $x_1, \ldots, x_n$. ν is a set of formulae that are constructed by prior applications of ConstructFormula, so that we can use ConstructFormula incrementally to build up complex formulae. We deal only with finite theories here. We show in Section 5.1 how to extend our method to infinite theories, i.e., countably infinite sets of wff's. Given a finite theory, ConstructFormula iterates through all possible valuations of the formulae, checks if the valuation can be satisfied, and presents a satisfying assignment (if any), to the user. If user judges this assignment in *vtt*, then ConstructFormula converts the valuation into a conjunctive clause and disjoins it into the formula being constructed. If user judges this assignment in *vff*, then ConstructFormula does not alter the formula being constructed. In the worst case, the number of user queries is $2^{|\varepsilon|}$ where ε is the type theory or vocabulary being used.

Let F be a finite set of first order wff's. $V : F \rightarrow \{tt, \mathit{ff}\}$ is a *Boolean valuation of* F, i.e., a mapping that assigns to each $f \in F$ a truth-value. Write $F \mapsto \{tt, \mathit{ff}\}$ for the set of valuations of F. Define $\mathit{fm}(V) \triangleq (\bigwedge f \in F : f \equiv V.f)$, i.e., $\mathit{fm}(V)$ is the formula which asserts that each $f \in F$ has the truth value assigned to it by V. We call $\mathit{fm}(V)$ a *valuation formula.* Define $[V] \triangleq \{\sigma \mid \sigma \in \Sigma \text{ and (for all } f \in F : \sigma.f = V.f)\}$, i.e., $[V]$ is the set of all behaviors that assign the same values to the formulae in F that V does. Note that $[V] = [\mathit{fm}(V)]$. Define $F_\sim = \{\langle \sigma, \sigma' \rangle \mid \text{(for all } f \in F : \sigma.f = \sigma'.f)\}$. $F_\sim$ then is the equivalence relation on Σ that considers two behaviors equivalent iff they assign the same values to the formulae in F. Define $\Sigma/F_\sim = \{[V] \mid V \in F \mapsto \{tt, \mathit{ff}\}\}$. $\Sigma/F_\sim$ is the partition of Σ induced by $F_\sim$. We assume the standard definitions for one partition of Σ being finer (coarser) than another, and write $\Sigma/E \leqslant \Sigma/E'$ when Σ/E is finer than Σ/E'.

4.1 Type theory

The type theory τ defines the universes $U_1, \ldots, U_n$ for the free variables $x^i_1, \ldots, x^i_n$ and $x^o_1, \ldots, x^o_n$ and provides axioms for their operations. It represents the "finest granularity of expression" that we have. We construct a formula $\mathcal{F}$ in disjunctive normal form: $\mathcal{F} = \bigvee_i (\bigwedge_j F_{ij})$. Hence $[\mathcal{F}] = \bigcup_i (\bigcap_j [F_{ij}])$. Assume that the F_{ij} are formulae (or their negations) from τ. Hence $[F_{ij}]$ is a union of equivalence classes of Σ/τ, namely all those classes where F_{ij} holds. Hence $\bigcup_i (\bigcap_j [F_{ij}])$ is also a union of equivalence classes of Σ/τ, since Σ/τ is a partition of Σ. If some F_{ij} are not formulae of τ, but are the result of prior applications of ConstructFormula, then this still holds, by induction on the number of applications of ConstructFormula. Hence $[\mathcal{F}]$ is a union of equivalence classes of Σ/τ.

A solution $\mathcal{F}$ to the formula construction problem satisfies $[\mathcal{F}] = vtt$. This implies that vtt is a union of equivalence classes of Σ/τ. However, vtt is arbitrary, since it is defined by the user: the user provides information about vtt by answering membership queries for vtt. Hence, for a solution to the formula construction problem to exist, we *must* assume the following axiom, which we take for given in the sequel:

$$\Sigma/\tau \leqslant \{vtt, vff\} \qquad \text{Axiom-T}$$

Proposition 1. *Let* $\mathcal{F} = (\bigvee V_\tau : V_\tau \in \tau \mapsto \{tt, ff\} \wedge [V_\tau] \subseteq vtt : fm(V_\tau))$*. Then* $[\mathcal{F}] = vtt$*, and so* $\mathcal{F}$ *is a solution to the formula construction problem.*

Often, τ will contain a large number of formulae, and since the number of user queries is $O(2^{|\tau|})$, this may be impractical. We reduce the number of user queries by using, in place of τ, a *vocabulary.*

4.2 Vocabulary

A vocabulary is concerned with the particular concepts that go into a specification. Consider a specification for searching an array a for a value e. This uses the sub-concept "e occurs in a". Given a formula for this, e.g., $(\exists i : 0 \leqslant i < |a| : a[i] = e)$, it is easier and faster to write an accurate specification than it would be using only basic type theory for the integers. It also requires fewer sub-formulae, resulting in fewer user queries.

Starting from type theory τ, we write formulae for simple concepts. We use this "first level" vocabulary to write more complex concepts, constituting a "second level" vocabulary. So level-by-level we build up more complex formulae.

We showed above that for any formula $\mathcal{F}$ written by ConstructFormula, $[\mathcal{F}]$ is a union of equivalence classes of Σ/τ. Since all the formulae in a vocabulary ν are either formulae of τ or are written by ConstructFormula, we conclude that for all $f \in \nu$: $[f]$ is a union of equivalence classes of Σ/τ. This implies that $\Sigma/\tau \leqslant \Sigma/\nu$. Since Σ/ν is coarser than Σ/τ, it is possible that Σ/ν is not finer than $\{vtt, vff\}$. This means that the vocabulary ν cannot express the intended partition $\{vtt, vff\}$, since some key concept is not expressed as a formula of ν. For example, consider array sorting, and a ν which includes ordering of output elements, but not equality of input and output elements. A formula $\mathcal{F}$ constructed from ν can express that the output array is ordered, but cannot express that the output array is a permutation of the input. To be able to express $\{vtt, vff\}$ using a formula constructed from ν, we require that ν is *adequate*:

Definition 6 (Adequacy). ν *is* adequate *iff* $\Sigma/\nu \leqslant \{vtt, vff\}$

that is, for each valuation V_ν of ν, $[V_\nu]$ is wholly contained in vtt or in vff. Unlike the situation for τ, we cannot take $\Sigma/\nu \leqslant \{vtt, vff\}$ as an axiom, since ν can contain arbitrarily coarse formulae, i.e., formulae f with large $[f]$. In practice, we want the coarsest formulae possible, since this reduces the number of user queries. The risk is that Σ/ν is too coarse, violating $\Sigma/\nu \leqslant \{vtt, vff\}$, and has to be corrected. A heuristic for adequacy of ν is that ν contains formulae for every concept in an informal description of the problem, e.g., for both ordering and permutation in the case of array sorting.

The full paper presents MakeAdequate($\nu, \varepsilon, vtt, vff$), which checks adequacy of ν. Here ε is a lower level adequate theory (type theory or a lower level vocabulary) from which ν is constructed, i.e., formulae of ν are Boolean combinations of formulae of ε. MakeAdequate makes $2^{|\varepsilon|}$ queries to the user. MakeAdequate either verifies adequacy of ν, or returns a set of "correction formulae" whose addition to ν makes it adequate.

Assume in the sequel that ν is adequate. Then the union of all $[V_\nu]$ contained in vtt is exactly vtt: $vtt = (\bigcup V_\nu : V_\nu \in \nu \mapsto \{tt, ff\} \wedge [V_\nu] \subseteq vtt : [V_\nu])$. Hence:

Proposition 2. *Let $\mathcal{F} = (\bigvee V_\nu : V_\nu \in \nu \mapsto \{tt, ff\} \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$. Then $[\mathcal{F}] = vtt$, and so $\mathcal{F}$ is a solution to the formula construction problem.*

4.3 The ConstructFormula Algorithm

Algorithm ConstructFormula, given in Figure 3, takes as input the partition $\{vtt, vff\}$ that the user intends, and a vocabulary ν that is adequate w.r.t. this partition. It evaluates $\mathcal{F} = (\bigvee V_\nu : V_\nu \in \nu \mapsto \{tt, ff\} \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$. We start with $\mathcal{F}$ set to false, and iterate through the valuations V_ν. For each V_ν, we submit $fm(V_\nu)$ to a Satisfiability-modulo-theory (SMT) solver, e.g., Z3 [20]. An SMT solver takes as input a formula in a defined theory under first order logic. The SMT solver produces one of these three possible outcomes: (1) it exhausts its computational resources, (2) it returns a satisfying assignment for $fm(V_\nu)$, or (3) it returns that $fm(V_\nu)$ is unsatisfiable.

In case (1), ConstructFormula terminates with failure. Feedback from the failed attempt can be used to modify the problem, e.g., by changing ν, and then re-attempting. In case (2), a satisfying assignment is a behavior $\sigma \in \Sigma$. We present σ to the user, who judges whether $\sigma \in vtt$ or $\sigma \in vff$. If the user responds "in vtt", then, by Definition 6, we have $[V_\nu] \subseteq vtt$. Hence we update $\mathcal{F}$ by disjoining $fm(V_\nu)$ to it, as indicated by $\mathcal{F} := \mathcal{F} \frown \text{“} \vee \text{”} \frown fm(V_\nu)$ in Figure 3, where $\frown$ denotes string concatenation, i.e., we are constructing the text of the formula $\mathcal{F}$ as a concatenation of disjuncts. In case (3), we conclude, by Definition 6, that $[V_\nu] \subseteq vff$, so we do not alter $\mathcal{F}$. We annotate the pseudocode in Figure 3 with a loop invariant and some Hoare-style annotations. Theorem 1 below follows (proof in Appendix A).

Theorem 1 (Correctness of ConstructFormula). *Assume that (1) ν is* adequate *for $\{vtt, vff\}$ and (2) that no invocation of the SMT solver by ConstructFormula fails, and (3) the user responds accurately to all queries. Then ConstructFormula (ν, vtt, vff) returns formula $\mathcal{F}$ such that $[\mathcal{F}] = vtt$.*

4.4 Complexity and optimization

The running time of ConstructFormula is at most $2^{|\nu|}$ calls to the SMT solver. It also makes at most $2^{|\nu|}$ queries to the user. We discuss next optimizations that reduce the number of SMT and user queries.

```
ConstructFormula($\nu$, $vtt$, $vff$)
1.  { Precondition: $\{vtt, vff\}$ partitions $\Sigma$ and $\Sigma/\nu \leqslant \{vtt, vff\}$ }
2.  $\mathcal{F} := false$; $\varphi := \nu \mapsto \{tt, ff\}$
3.  { Invariant: $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \in (\nu \mapsto \{tt, ff\}) - \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ }
4.  while $\varphi \neq \emptyset$
5.      select some valuation $V_\nu \in \varphi$;
6.      $\varphi := \varphi - \{V_\nu\}$;
7.      submit $fm(V_\nu)$ to an SMT solver;
8.      if the solver fails then return("failure");                          return with failure
9.      if $fm(V_\nu)$ is satisfiable then
10.         let $\sigma_v$ be the returned satisfying assignment;
11.         query the developer: is $\sigma_v$ in $vtt$ or in $vff$?
12.         if developer answers $\sigma_v \in vtt$ then $\mathcal{F} := \mathcal{F} \frown$ " $\vee$ " $\frown fm(V_\nu)$;
13.         else $\varphi := \varphi -$ partialAssignment($\sigma_v$);                 see Section 4.4
14.     else                                                                 solver returned unsat
15.         $\varphi := \varphi - unsat$ where $unsat$ is the unsat core valuations;   see Section 4.4
16. endwhile;
17. { Postcondition: $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \in \nu \mapsto \{tt, ff\} \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ }
18. simplify $\mathcal{F}$ using ABC [21] and ESPRESSO [22];
19. return($\mathcal{F}$);
```

Fig. 3: ConstructFormula(ν, vtt, vff)

Unsat-core elimination When $fm(V_\nu)$ is found to be unsatisfiable, we obtain the unsat core from the SMT solver, and eliminate from consideration all V_ν that are extensions of the unsat core, since these are all unsatisfiable. This happens automatically, without user involvement; Line 15 in Figure 3. Figures 1(e,f) show the unsat classification of F_1 and the six eliminated unsat core (u-c) formula valuations, illustrated in orange color.

Partial-assignment elimination The user can eliminate many valuations at once. When the user judges a behavior to be in vff, she can select a subset of the variables assigned as the real reason for the choice ("V" option in the tool). The partial assignment selected by the user may leave some formulae in ν without a well-defined truth value. Consider assignment σ_6 that was judged incorrect by the user in the index validity specification of Figure 1. The user can use the "V" option in the tool to select the negative value of i as the reason for the judgment. The partial assignment selected by the user evaluates f_1 and f_2 to *false* and leaves f_3 and f_4 undefined since they depend on $|a|$. Hence we classify all valuation formulae corresponding to $\neg f_1 \wedge \neg f_2$ (namely F_{14}, F_{15}, and F_{16}) as in vff. Hence we eliminated two queries to the solver and the user.

Alternatively, the user can directly select $\neg f_1, \neg f_2$ as offending, using the "B" option in the tool. All extensions of $\neg f_1 \wedge \neg f_2$ are eliminated. "V" and "B" eliminations are represented by line $\varphi := \varphi -$ partialAssignment(σ_v) in Figure 3.

5 Decidability and Finiteness Considerations

ConstructFormula may fail if the SMT solver fails on some call, i.e., fails to decide satisfiability of some $fm(V_\nu)$ formula. Restricting the syntax of $fm(V_\nu)$ formulae to decidable fragments of first order logic avoids this problem. Examples are fragments solvable in a finite domain, such as equality, monadic, and quantifier free theories [23]. Also, Bradley et. al. [24] take a formula in a restricted syntactic form (array property theory) and translate it to an equivalent formula in the combined theory of equality with uninterpreted functions of finite index (EUF), which is decidable. Our method uses a grammar to construct an EUF theory, which we use as a vocabulary. The resulting valuation formulae are in EUF. A grammar with list selectors allows us to use the results of Furia [25] for specifications over sequences. Such theories are enough to express specifications such as sortedness and injectivity. We use the following grammar:

– Type theory τ expressed as a set of variables X and a map from X to scalar and array types
– Set of literal constants L such as $0, 1, \mathit{true}$, and false,
– Presburger and index operations alphabet $\Sigma = \{\mathit{index}, \mathit{bound}, =, \leqslant, +, -, *, []\}$,
– Formula generation rules $G \subseteq X \times (X \cup L) \times 2^{\Sigma}$, denoting allowed operations. The user provides the rules in a simple form as a set of tuples, e.g., `{(a,i,index),(a,e,=)}` denoting `i` is an index to `a` and `e` is data w.r.t `a`,
– Bound K giving the maximum allowed number of operations in a clause

Our implementation traverses this grammar and builds the vocabulary ν as the set of all Boolean formulae with up to K operators. All SMT queries based on valuations of ν are then in a decidable fragment of first-order logic. ν can serve as the "first level" vocabulary, and we can proceed by using ConstructFormula(ν,vtt,vff) to write formulae for the "second level" vocabulary, etc.

5.1 Reduction to finite theories

In the worst case, ConstructFormula makes a query for each of the $2^{|\nu|}$ valuations of ν. Hence ν must be finite if ConstructFormula is to terminate. Infinite theories arise when unbounded structures such as arrays are used in the specification. We illustrate a reduction from an infinite theory ε to a finite one, when ε is restricted to consist of a finite number of *scalar formulae* $g_1(\bar{z}, |a|), \ldots, g_m(\bar{z}, |a|)$, and a finite number of *indexed formula set expressions* $\{f_1(\bar{z}, a, \bar{i}) \mid r_1(|a|, \bar{i})\}, \ldots, \{f_n(\bar{z}, a, \bar{i}) \mid r_n(|a|, \bar{i})\}$, which represent infinite sets of formulae. The range predicate $r(|a|, \bar{i})$ must be monotonic in $|a|$: for $b' \geqslant b$, $\{\bar{v} \mid r(\bar{v}, b)\} \subseteq \{\bar{v} \mid r(\bar{v}, b')\}$. The key idea of the reduction is to find a finite theory that is large enough so that every valuation of the infinite theory ε can be "represented" in the finite theory. Consider the theory $\ell = r$, $\ell < r$, $\ell \geqslant 0$, $\ell \leqslant |a| - 1$, $r \geqslant 0$, $r \leqslant |a| - 1$, $\{a[i] = e$ for all i such that $0 \leqslant i < |a|\}$, and the valuation that assigns tt to all of $\ell \geqslant 0$, $\ell < r$, and $r \leqslant |a| - 1$. This is possible only if $|a| \geqslant 2$. Hence we can reduce ε to a finite theory in which $|a| \geqslant 2$. Appendix C, and also the full paper, present the reduction formally.

6 Construction of General Formulae and Vocabularies

We generalize ConstructFormula to introduce existential quantifiers at the beginning of a formula $\mathcal{F}$ that it is constructing. Universal quantifiers can be obtained by negation. We also show how to hierarchically construct vocabularies, paying attention to how quantified formulae in a vocabulary are handled, and how to restrict the SMT queries to limit quantifier alternation.

```
ConstructQuantifiedFormula($\nu$, $vtt$, $vff$)
1. Decide on the existentially quantified variables $\bar{x}$
2. Extend $\nu$ with all clauses comparing variables in $\bar{x}$ to all index & bound terms
3. Call the resulting vocabulary $\nu'$
4. Invoke ConstructFormula($\nu'$, $vtt$, $vff$) and let $\mathcal{F}$ be the result
5. Return $\exists\,\bar{x}(\mathcal{F})$
```

Fig. 4: Quantified formula construction ConstructQuantifiedFormula(ν, vtt, vff)

6.1 Constructing a formula with leading existential quantification

Algorithm ConstructQuantifiedFormula (Figure 4) constructs formulae of the form $\exists\bar{x}(f)$, where $\bar{x}$ is a finite list of variables. The satisfying assignments (behaviors) of f, with the quantified variables $\bar{x}$ projected out, (relational projection), are the satisfying assignments of $\exists\,\bar{x}(f)$. Hence, we construct a formula $\mathcal{F}$ using ConstructFormula, and then project out the $\bar{x}$ by placing $\exists\,\bar{x}$ before $\mathcal{F}$. In effect, we "hide" the quantified variables, see e.g., Lamport [26, Sections 4.3, 8.8]. Consider using ConstructFormula to express: value e occurs in array a between indices ℓ and r inclusive. We inject a quantified variable i, and apply ConstructFormula to obtain $0 \leqslant \ell \wedge \ell \leqslant r \wedge r < |a| \wedge \ell \leqslant i \wedge i \leqslant r \wedge a[i] = e$. Since i is flagged as quantified, we project i out by prepending an $\exists\,i$ in front, to obtain $\exists\,i : 0 \leqslant \ell \wedge \ell \leqslant r \wedge r < |a| \wedge \ell \leqslant i \wedge i \leqslant r \wedge a[i] = e$.

6.2 Hierarchical vocabulary construction

Let $\mathcal{F}_{k-1}$ be a formula generated at vocabulary level $k-1$. We wish to use $\mathcal{F}_{k-1}$ to write a formula $\mathcal{F}_k$ at vocabulary level k. We have two approaches:

Plug-in. Include $\mathcal{F}_{k-1}$ as is in the vocabulary ν given to ConstructFormula(ν, vtt, vff) when used to construct $\mathcal{F}_k$.

Boolean abstraction. Introduce a free Boolean variable b to abstract $\mathcal{F}_{k-1}$, and to reduce quantifier alternation. Add b to the vocabulary ν given to ConstructFormula(ν, vtt, vff) when used to write $\mathcal{F}_k$, but modify ConstructFormula to make the following check. Suppose the SMT solver generates an assignment that assigns *false* to b but causes $\mathcal{F}_{k-1}$ to evaluate to *true*. This assignment is inconsistent with the definition of b and is therefore automatically placed in vff. Consider $\mathcal{F}_{k-1} = (\exists i : 0 \leqslant \ell \leqslant i \leqslant r < |a| : a[i] = e)$, and let $eina$ be the introduced Boolean variable, so that $eina = \mathcal{F}_{k-1}$. Consider the assignment: $eina = false, \ell = 3, r = 6, a[5] = 7, e = 7, |a| = 9$. $eina$ should be true here, since $\mathcal{F}_{k-1}$ is. Hence the assignment is inconsistent, and is automatically discarded.

7 Implementation and Experiments

We conducted experiments to write six specifications with our method and the supporting tool. The implementation of ConstructFormula is available online[1], along with documentation, instructions, and logs for the experiments. We used Z3 as the SMT solver [20], and ESPRESSO [22] and ABC [21] as logic synthesis tools to simplify the specification.

[1] `http://webfea.fea.aub.edu.lb/fadi/dkwk/doku.php?id=speccheck`

Table 1: Results of user experiments with `sc`

Spec	Valuations	SMT Queries		User Queries		Users	Time(min)		MAX SMT	Correct	Manual	
		Min	Max	Min	Max		Min	Max	Time(ms)		correct	time(min)
ASRCH(3)	4,144	77	337	18	251	8	6	44.15	118	6/8	2/5	10-15
ORDR (2)	128	15	40	11	36	6	3	6.77	13.2	5/6	1/3	20
BSRCH(2)	24	10	16	6	12	4	4	10	6.4	4/4	2/5	10
LLIST(1)	2^{38}	33	45	11	22	2	5	8	34	failure	2/5	15
LLIST(4)	176	48	124	32	47	2	12	26	242	2/2	2/5	15
BHEAP(4)	80	18	54	12	41	3	16	30	253	3/3	N/A	N/A
TRIE(5)	224	32	76	20	52	2	25	42	452	2/2	N/A	N/A

Eight computer engineering students (4 undergraduate and 4 graduate) and two logic designers were trained with simple examples; e.g., $x < y \wedge y < z$, and then used our method to write the specifications. Six selected students from a junior class on Algorithms wrote the specifications manually.

The "Spec" column in Table 1 lists each specification with the number of hierarchical levels used to write it. ASRCH is array search, ORDR is array ordered, BSRCH is binary search, LLIST is consistent node in a double linked list, BHEAP is binary heap, and TRIE is the trie (root indexed tree) structure. The users used previously generated specifications such as ORDR and ASRCH to write new specifications such as BSRCH. The columns in Table 1 report the number of valuations, number of SMT and user queries, minimum and maximum time, number of correct specifications, and compare the results against the group who wrote the specifications manually.

All the specifications written with the tool were more accurate than those written manually and took reasonably better and comparable time. Even-though users were warned not to use indirection in array indexing (`next[prev[i]]`) since that leads the SMT solver to fail [24], two users (Row LLIST(1)) did and tried to write the consistency of a node in a linked list directly without the hierarchical method. The SMT solver failed on one of the formula valuations. The two other users (Row LLIST(4)) used additional variables and previously generated predicates (`isvalid(i)`, `isnext(i,n)`) to write the specification successfully. The mistakes in the specifications written with the tool were underspecifications due to the use of the partial assignment pruning technique. Reportedly formulae with quantifiers, such as the intermediary `eina` and the ORDR properties were much easier to construct with the tool than to write manually.

8 Related work and Conclusions

The methods in [2, 3] work by writing the specification and then attempting to verify if it is accurate using animation and execution. In contrast, we write the specification from behaviors so that it is accurate by construction. A method of checking software cost reduction (SCR) specifications for consistency is presented in [4]. Zeller in [27] discusses writing specifications as models discovered from existing software artifacts of relevance to the desired functionality. In none of the above is there an analogue to our construction of preconditions and postconditions as formulae of first order logic.

The work in [9] constructs an SQL query given a database with table descriptions and a set of input output examples provided by the user. It uses a restricted SQL grammar built after a user survey, generates a set of SQL queries fitting the input output examples, and selects the one with the fewest conditions.

The works in [8, 12] start from a specification relating inputs to outputs and generate program implementations for (a) algebraic data types and arrays, and (b) linear arithmetic and sets, respectively. The works in [7, 13] use generic type information, parametric polymorphism, test cases, and existing pre/post-conditions to write code snippets that compute expressions that match a type at a point in code. The building blocks come from program elements in the scope. The work in [13] is interactive; it uses generic type information with a resolution based algorithm to present candidate code fragments to the user who refines them. In [7] weights derived from a code corpus rank the candidates.

The work in [11] uses natural language processing techniques to identify entities and components in the code, uses techniques similar to [7] to synthesize smartphone automation scripts in a proprietary intermediate language, and interacts with the user who provides feedback in a natural language about the entities that need to be modified and the relations between them, until the generated script is satisfactory. Escher [10] synthesizes a recursive program interactively from a set of program components and a set of input-output examples provided by the user. It does so by searching the possible recursive programs that can be generated using the provided components in two alternating manners: (1) forward search, and (2) conditional inference, and measures its conversion against a goal graph generated from the user provided inputs and outputs.

In contrast, (1) we construct specifications, i.e., quantified formulas in FOL, that are needed by some of the above techniques, (2) our method is hierarchical, i.e., does not require initial basic components, and (3) our user judges the correctness of concrete behaviors, and not the quality of the synthesized program.

In [14], an oracle-based method computes a loop free program that requires a distinguishing constraint and an I/O behavior constraint. The method either synthesizes a program or claims the provided components are insufficient. We differ in that we build specifications in first order logic with quantifiers, we do not require a distinguishing constraint; we compute adequacy with respect to the type theory and correct ν when it is not adequate.

The SPECIFIER [5] tool constructs formal specifications of data types and programs from informal descriptions, but uses schemas, analogy, and difference-based reasoning, rather than I/O behaviors. Larch [6] improves confidence in the specification's accuracy by verifying claims about the specification.

Conclusions Our method constructs a formal specification in first order logic (including quantifiers) given (1) a type theory, (2) a grammar, and (3) a sequence of user judgments of behaviors. User experiments demonstrated the benefits of our approach. Our method guarantees success for theories that satisfy both of the syntactic restrictions in Section 5, i.e., grammar and reduction (ε). Future theoretical work consists of extending this class of theory, and also applying the ideas presented here to program verification.

References

1. Brooks, F.: No silver bullet. In: IFIP 10'th World Computing Conference. (1986)
2. Heimdahl, M., Whalen, M., Thompson, J.: Nimbus: a tool for specification centered development. In: Requirements Engineering Conference. (2003)
3. Hazel, D., Strooper, P., Traynor, O.: Requirements engineering and verification using specification animation. In: Proc. Automated Software Engineering. (1998)
4. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM TOSEM **5** (July 1996)
5. Miriyala, K., Harandi, M.: Automatic derivation of formal software specifications from informal descriptions. IEEE TSE **17**(10) (oct 1991) 1126 –1142
6. Garland, S., Guttag, J., Horning, J.: Debugging larch shared language specifications. IEEE TSE **16**(9) (sep 1990) 1044 –1057
7. Gvero, T., Kuncak, V., Kuraj, I., Piskac, R.: Complete completion using types and weights. In: PLDI. (2013) 27–38
8. Jacobs, S., Kuncak, V., Suter, P.: Reductions for synthesis procedures. In: VMCAI. (2013)
9. Zhang, Z., Sun, Y.: Automatically synthesizing sql queries from input-output examples. In: Automated Software Engineering, (ASE). (2013)
10. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Computer Aided Verification. (2013) 934–950
11. Le, V., Gulwani, S., Su, Z.: Smartsynth: synthesizing smartphone automation scripts from natural language. In: MobiSys. (2013) 193–206
12. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT **15**(5-6) (2013) 455–474
13. Gvero, T., Kuncak, V., Piskac, R.: Interactive synthesis of code snippets. In: Computer aided verification, CAV. (2011) 418–423
14. Jha, S., Gulwani, S., Seshia, S., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE. (May 2010) 215–224
15. Ernst, M.D., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming **69** (December 2007) 35–45
16. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: ICSE. (2010)
17. Khurshid, S., Marinov, D.: TestEra: Specification-based testing of java programs using sat. Automated Software Engineering **11** (October 2004) 403–434
18. Wenzel, M., Paulson, L., Nipkow, T.: The isabelle framework. In: Theorem Proving in Higher Order Logics, TPHOLs. (2008) 33–38
19. Kaufmann, M., Moore, J.: How can i do that with acl2? recent enhancements to acl2. In: ACL2 Workshop. (2011) 46–60
20. Bjorner, N., de Moura, L.: Z3 10: Applications, enablers, challenges and directions. In: CFV. (2009)
21. Mishchenko, A., Eén, N., Brayton, R., Case, M., Chauhan, P., Sharma, N.: A semi-canonical form for sequential AIGs. In: DATE. (2013) 797–802
22. Brayton, R., Sangiovanni-Vincentelli, A., McMullen, C., Hachtel, G.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic (1984)
23. Ackerman, W.: Solvable cases of the decision problem. North-Holland (1954)
24. Bradley, A., Manna, Z., Sipma, H.: What 's decidable about arrays? In: VMCAI. Volume 3855. (January 2006)
25. Furia, C.: What's decidable about sequences? In: ATVA. (2010)
26. Lamport, L.: Specifying Systems. Addison Wesley Longman Inc. (2003)
27. Zeller, A.: Mining models. In: Model Checking Software. Volume 7385. (2012)

Appendices

A Proofs

Proposition 1. *Let $\mathcal{F} = (\bigvee V_\tau : V_\tau \in \tau \mapsto \{tt, f\!f\} \wedge [V_\tau] \subseteq vtt : fm(V_\tau))$. Then $[\mathcal{F}] = vtt$, and so $\mathcal{F}$ is a solution to the formula construction problem.*

Proof. By Axiom-T, each $[V_\tau]$ is wholly contained in either vtt or in $vf\!f$. Hence the union of all V_τ that are contained in vtt is exactly vtt: $vtt = (\bigcup V_\tau : [V_\tau] \subseteq vtt : [V_\tau])$. Hence, the disjunction of all the formulae $fm(V_\tau)$ corresponding to V_τ that are contained in vtt yields a formula which is true at all elements of vtt and false outside of vtt. That is, for $\mathcal{F} = (\bigvee V_\tau : [V_\tau] \subseteq vtt : fm(V_\tau))$, we obtain $[\mathcal{F}] = vtt$.

Proposition 2. *Let $\mathcal{F} = (\bigvee V_\nu : V_\nu \in \nu \mapsto \{tt, f\!f\} \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$. Then $[\mathcal{F}] = vtt$, and so $\mathcal{F}$ is a solution to the formula construction problem.*

Proof. By definition of $[\mathcal{F}]$, we have $[\mathcal{F}] = (\bigcup V_\nu : V_\nu \in \nu \mapsto \{tt, f\!f\} \wedge [V_\nu] \subseteq vtt : [fm(V_\nu)])$. Since $[fm(V_\nu)] = [V_\nu]$, we have $[\mathcal{F}] = (\bigcup V_\nu : V_\nu \in \nu \mapsto \{tt, f\!f\} \wedge [V_\nu] \subseteq vtt : [V_\nu])$. By the above discussion, $[\mathcal{F}] = vtt$.

Theorem 1 (Correctness of ConstructFormula). *Assume that (1) voc is* adequate *for $\{vtt, vf\!f\}$ and (2) that no invocation of the SMT solver by ConstructFormula fails, and (3) the developer responds accurately to all queries. Then ConstructFormula $(\nu, vtt, vf\!f)$ returns formula $\mathcal{F}$ such that $[\mathcal{F}] = vtt$.*

Proof. We first establish the validity of the invariant $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ and postcondition $\mathcal{F} \equiv (\bigvee V_\nu : [V_\nu] \subseteq vtt : fm(V_\nu))$ for ConstructFormula; see Figure 3. The set φ consists of those valuations over ν that remain to be queried to the developer. Initially, $\varphi = \nu \mapsto \{tt, f\!f\}$, i.e., it consists of all the valuations over ν, and so the range $V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt$ is empty. Hence both sides of the Invariant are false, and so the Invariant holds initially.

Consider an arbitrary iteration of the while loop in which valuation V'_ν is selected (we use V'_ν rather than V_ν as in the code to avoid a name clash with the bound V_ν). By assumption (2), the SMT solver succeeds and returns a result.

Case 1: *$fm(V'_\nu)$ is unsatisfiable.* Then V'_ν, and possibly other unsatisfiable valuations, are removed from φ, in lines 18–19. Since $fm(V'_\nu) \equiv f\!f$, the Invariant is unaffected by the removal of unsatisfiable valuations from φ.

Case 2: *$fm(V'_\nu)$ is satisfiable.* Let σ_v be the satisfying assignment for $fm(V'_\nu)$ returned by the SMT solver in line 10. In line 11, the developer is queried as to whether $\sigma_v \in vtt$ or not. By assumption (3), the developer responds accurately to this query. We have two subcases.

Subcase 2.1: $\sigma_v \in vtt$. Hence the developer responds $\sigma_v \in vtt$, which causes line 13 to be executed; $fm(V'_\nu)$ is added as a disjunct in $\mathcal{F}$. Since $\sigma_v.fm(V_\nu) = tt$ and $\sigma_v \in vtt$, we have $[V'_\nu] \cap vtt \neq \emptyset$. By assumption (1) and (Ad), we have $\Sigma/\nu \leqslant$

$\{vtt, vff\}$. Hence $[V'_\nu] \subseteq vtt$. So, V'_ν is added to the range $V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt$ in $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ while $fm(V'_\nu)$ is added to $\mathcal{F}$ as a disjunct. This preserves the equivalence, and so the invariant continues to hold.

Subcase 2.2: $\sigma_v \in vff$. Hence the developer responds $\sigma_v \in vff$, which causes line 15 to be executed; $\mathcal{F}$ remains unchanged. Since $\sigma_v.fm(V_\nu) = tt$ and $\sigma_v \in vff$, we have $[V'_\nu] \cap vff \neq \emptyset$. By assumption (1) and (Ad), we have $\Sigma/\nu \leqslant \{vtt, vff\}$. Hence $[V'_\nu] \subseteq vff$. So, V'_ν is not added to the range $V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt$ in $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$. Since $\mathcal{F}$ remains unchanged, both sides are unchanged, which preserves the equivalence. Hence the invariant continues to hold.

Thus we have shown, in all cases, that $\mathcal{F} \equiv (\bigvee V_\nu : V_\nu \notin \varphi \wedge [V_\nu] \subseteq vtt : fm(V_\nu))$ is preserved by a single execution of the while loop. We also showed that it holds initially. Hence it is actually an invariant.

Upon termination, we have $\varphi = \emptyset$. Substituting into the invariant, we obtain that $\mathcal{F} \equiv (\bigvee V_\nu : [V_\nu] \subseteq vtt : fm(V_\nu))$ holds upon termination. This is the required postcondition.

Let σ be an arbitrary element of vtt. Hence $\sigma \in [V_\nu]$ for some V_ν such that $[V_\nu] \subseteq vtt$, since $\Sigma/\nu \leqslant \{vtt, vff\}$. Since $\sigma.fm(V_\nu) = tt$ by our choice of σ and V_ν, we have $\sigma.\mathcal{F} = tt$.

Now let σ be an arbitrary element of vff. Hence $\sigma \in [V_\nu]$ for some V_ν such that $[V_\nu] \subseteq vff$, since $\Sigma/\nu \leqslant \{vtt, vff\}$. Also since $\Sigma/\nu \leqslant \{vtt, vff\}$, there is no V'_ν such that $\sigma \in V'_\nu \wedge V'_\nu \subseteq vtt$. Hence $\sigma.\mathcal{F} = ff$.

We conclude from the above that $[\mathcal{F}] = vtt$, i.e., $\mathcal{F}$ is accurate with respect to $\{vtt, vff\}$.

B Not-in-order example

The following is a successful run of the tool that implements our method. The run specifies that an array is out of order. The result will be negated later to specify that the array is in order. The user passed the `notinorder` theory below to the tool.

```
theory inorder {
 int [] a;
 constants {0,1}
 grammar { (a,a,<=);}
 //The following are default values.
 //We list them here for illustration
 num_operations_per_clause_bound = 3;
 num_quantifiers_bound = 1;
}
```

The tool injects a quantified variable i as an index into a, and uses the production rules of the grammar to construct the following vocabulary.

```
theory inorder{
  node 'a' of type array v--- value 0 sizes (0,0,0)  id 0 and 0 operands.
  node 'a.size_minus_1' of type int v--- value 0 sizes (0,0,0)  id 1 and 0 operands.
  node 'lit(0)' of type literal -c-- value 0 sizes (0,0,0)  id 2 and 0 operands.
  node 'lit(1)' of type literal -c-- value 1 sizes (0,0,0)  id 3 and 0 operands.
  node 'lit(-1)' of type literal -c-- value -1 sizes (0,0,0)  id 4 and 0 operands.
  node 'i' of type int v-u- value 0 sizes (0,0,0)  id 5 and 0 operands.
  node '[]' of type access --ud value 0 sizes (0,0,0)  id 6 and 2 operands.
  node '+' of type binary-arithmetic --u- value 0 sizes (0,0,0)  id 7 and 2 operands.
  node '[]' of type access --ud value 0 sizes (0,0,0)  id 8 and 2 operands.
  node '<=' of type binary-relational --u- value 0 sizes (0,0,0)  id 9 and 2 operands.
  node '<=' of type binary-relational --u- value 0 sizes (0,0,0)  id 10 and 2 operands.
  node '=' of type binary-relational --u- value 0 sizes (0,0,0)  id 11 and 2 operands.
  node '<=' of type binary-relational --u- value 0 sizes (0,0,0)  id 12 and 2 operands.
  node '=' of type binary-relational --u- value 0 sizes (0,0,0)  id 13 and 2 operands.
  node '<=' of type binary-relational --u- value 0 sizes (0,0,0)  id 14 and 2 operands.
  node '<=' of type binary-relational --u- value 0 sizes (0,0,0)  id 15 and 2 operands.
  rule (a,a,<=)
  rule (a,i,[])
  rule (lit(0),i,<=)
  rule (i,a.size_minus_1,<=,=)
  rule (i,lit(0),=)
  rule (i,lit(1),+)
  vocab {
  }
  genvocab {
    9: (0 <= i)
    10: (0 <= (1 + i))
    11: (0 = i)
    12: (i <= a.size_minus_1)
    13: (a.size_minus_1 = i)
    14: (a[i] <= a[(1 + i)])
    15: (a[(1 + i)] <= a[i])
  }
  qfvocab {
  }
}
```

The tool declares the variables and the formulae and their negations with the SMT Solver.

```
SPCHK: building SMT formulae from vocab...
(benchmark SMT: extrafuns( (a (Array Int Int)) (a.size_minus_1 Int) (i Int))
  :formula (<= 0 i) :formula ( not ( (<= 0 i)) )
  :formula (<= 0 (+ 1 i)) :formula ( not ( (<= 0 (+ 1 i))) )
  :formula (= 0 i) :formula ( not ( (= 0 i)) )
  :formula (<= i a.size_minus_1) :formula ( not ( (<= i a.size_minus_1)) )
  :formula (= a.size_minus_1 i) :formula ( not ( (= a.size_minus_1 i)) )
  :formula (<= (select a i) (select a (+ 1 i))) :formula ( not ( (<= (select a i) (select a (+ 1 i)))) )
  :formula (<= (select a (+ 1 i)) (select a i)) :formula ( not ( (<= (select a (+ 1 i)) (select a i))) ) )
SPCHK: parsing SMT formulae...
SPCHK: made 4 declarations as follows.
SPCHK:   0; NAME:i; DCL:(define i Int); AST:e15808
SPCHK:   1; NAME:a; DCL:(define a (Array Int Int)); AST:e15868
SPCHK:   3; NAME:a.size_minus_1; DCL:(define a.size_minus_1 Int); AST:e15848
```

The the tool enumerates all evaluations, queries the SMT solver for a satisfying assignment for each evalaution, and presents that to the user. Notice that for the asignment presented below the size of the array is 1 (`a.size_minus_1`=0) and thus the array is ordered. So the user answers “incorrect” and blames the size of the array for it by selecting valuation formulae 2 and 4 as the reason for the judgement.

```
SPCHK: enumerating vocab evaluations using recursive traversal...
SPCHK: checking choice...[1 1 1 1 1 1 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
       Notice that a specification accepts a don't care assignment. (this notice will be issued only twice).
  a[1:int] = 7719:int, a[0:int] = 7719:int, a[otherwise]= 7719:int
  a.size_minus_1 = 0:int
  i = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment. 'M' to view more details from the solver about the model.
USER > N
SPCHK: If the rejection is due to part of the assignment,
       press 'V' to select the bad variables,
       press 'B' to select the bad vocabulary values, or
       press 'C' to continue.
USER > b
  + 0+ (<= 0 i) is true
  + 1+ (<= 0 (+ 1 i)) is true
  + 2+ (= 0 i) is true
  + 3+ (<= i a.size_minus_1) is true
  + 4+ (= a.size_minus_1 i) is true
  + 5+ (<= (select a i) (select a (+ 1 i))) is true
  + 6+ (<= (select a (+ 1 i)) (select a i)) is true
SPCHK: Type the ids of the vocab formulae that are the reason for rejecting the satisfying example.Type '-1'  to finish, '-2' to
USER > 2 4
SPCHK: The reason for rejecting the model is the partial assignment
+ 2+ (= 0 i) is true
+ 4+ (= a.size_minus_1 i) is true
SPCHK: please confirm by typing 'Y'. Restart by typing 'R'. Ignore and continue by typing any other key.
USER > y
```

The formula valuations that are extensions of formula 2 and 4 being true will be automatically ignored as follows.

```
SPCHK: ignore bad partial assignment: [- - 1 - 1 - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 1 1 1 1 1 0 ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 1 1 1 1 0 - ]
```

When the user is presented by an array that is not in ascending order, the user judges that as correct. Notice that in the array below $a[0] = 1797$ is bigger than $a[1] = 1796]$. The valuation formula is also joined to the specification.

```
SPCHK: checking choice...[1 1 1 1 0 0 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
  a[1:int] = 1796:int, a[0:int] = 1797:int, a[otherwise]= 1796:int
  a.size_minus_1 = 1:int
  i = 0:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment. 'M' to view more details from the solver about the model.
USER > y
SPCHK: Adding vocab choice to the specification.
```

When an unsatisfiable formula is met, the tool computes the unsat cores and adds them to the eliminated patterns.

```
SPCHK: checking choice...[1 1 1 1 0 0 0 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is not satisfiable.
SPCHK: Adding unsat core to eliminated patterns...[- - - - - 0 0 ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 1 1 0 1 - - ]
```

The user judges that an empty array is also an ordered array (`a.size_minus_1 = -1`).

```
SPCHK: checking choice...[1 1 1 0 0 1 0 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is satisfiable.
       Is the satisfying assignment below a good model for your property?
  a[1:int] = 449:int, a[0:int] = 448:int, a[otherwise]= 449:int
  a.size_minus_1 = -1:int
SPCHK: Answer by 'Y' to accept and 'N' to reject the assignment. 'M' to view more details from the solver about the model.
USER > n
```

The tool now automatically ignores formula valuations because of both unsatisfiable cores and user selected partial assignments.

```
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 1 0 0 0 0 - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 1 1 1 - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 1 1 0 1 - ]
SPCHK: checking choice...[1 0 1 1 0 0 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is not satisfiable.
SPCHK: Adding unsat core to eliminated patterns...[1 0 - - - - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 1 1 0 0 0 ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 1 0 - - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[1 0 0 - - - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[0 1 1 1 1 - - ]
SPCHK: checking choice...[0 1 1 1 0 1 1 ]
SPCHK: Calling SMT Solver. This may take time..
SPCHK: choice is not satisfiable.
SPCHK: Adding unsat core to eliminated patterns...[0 - 1 - - - - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[0 1 1 1 0 1 0 ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[0 1 1 1 0 0 - ]
SPCHK: Ignore: subtree satisfies an eliminated pattern.[0 1 1 0 - - - ]
```

When the enumeration is done, the tool shows the SMT formula before Boolean simplification (22 lines in this case).

Then the tool simplifies the formula with ESPRESSO and ABC.

```
SPCHK: Calling logic minimization (Espresso) to simplify formula.
...
SPCHK: using abc for simplification.
SPCHK: abc command is read_blif yes.16943.abc.blif
...
SPCHK: simplified abc formula is:
Spec = exists i.  (0 <= i) and (0 <= (1 + i)) and (i <= a.size_minus_1) and !(a[i] <= a[(1 + i)]) and (!(a.size_minus_1 = i));
```

Then the tool prints statistics.

```
SPCHK: Timing and statistics report:
       number of vocab clauses: 7
       number of variables: 3

       number of queries to user: 16
       number of partial assignments: 5
       number of unsat cores: 8
       number of calls to sat solver (Z3): 20

       total traverse time (seconds):  221.476
       total sat time (micorseconds): 6605
       total match unsat core time (micorseconds): 54
```

C Reduction to finite theories

We present a reduction from the case of infinite theories to the finite case. Our reduction applies to both type theory and vocabulary, so we assume a generic theory ε.

An element of Σ defines values for some scalar variables $\bar{z}$ (e.g., booleans and integers) and some arrays $\bar{a}$. For ease of exposition, we assume that there is exactly one array a. It is straightforward to remove this restriction. Let $\bar{i}$ be a set of "dummy" variables, which we use to index a. Our reduction requires some syntactic restrictions, which we give as a definition of *reducible* theory:

Definition 7 (Reducible Theory Syntax). *A* reducible theory ε *consists of a finite number of* scalar formulae $g_1(\bar{z}, |a|), \ldots, g_m(\bar{z}, |a|)$, *and a finite number of* indexed formula set expressions $\{f_1(\bar{z}, a, \bar{i}) \mid r_1(|a|, \bar{i})\}, \ldots, \{f_n(\bar{z}, a, \bar{i}) \mid r_n(|a|, \bar{i})\}$.

The range predicate $r(|a|, \bar{i})$ must be monotonic in $|a|$: for $b' \geqslant b$, $\{\bar{v} \mid r(\bar{v}, b)\} \subseteq \{\bar{v} \mid r(\bar{v}, b')\}$. The $\bar{i}$ are dummy variables.

As indicated, a scalar formula can refer to the scalar variables $\bar{z}$ and to the size $|a|$ of array a. An indexed formula $f(\bar{z}, a, \bar{i})$ can refer to the $\bar{z}$, and to elements of a by using any of $\bar{i}$ as an index. The range predicate $r(|a|, \bar{i})$ can refer to $\bar{i}$ and $|a|$.

Definition 8 (Bounded Theory ε_b). *For $b > 0$ and reducible theory ε, the corresponding theory with bound b, ε_b, is the set of wffs $\{g_1(\bar{z}, b), \ldots, g_m(\bar{z}, b)\} \cup \{f_1(\bar{z}, a, \bar{i}) \mid r_1(b, \bar{i})\} \cup \cdots \cup \{f_n(\bar{z}, a, \bar{i}) \mid r_n(b, \bar{i})\}$. Each formula set expression $\{f(\bar{z}, a, \bar{i}) \mid r(b, \bar{i})\}$ denotes the set of formulae consisting of $f(\bar{z}, a, \bar{v})$ for each $\bar{v}$ such that $r(b, \bar{v})$ holds.*

As an example, a reducible theory ε for the array search speciifcation (search of an array a between indices ℓ and r inclusive) is:

- $\ell = r$
- $\ell < r$
- $\ell \geqslant 0$
- $\ell \leqslant |a| - 1$
- $r \geqslant 0$
- $r \leqslant |a| - 1$
- $\{a[i] = e$ for all i such that $0 \leqslant i < |a|\}$

The corresponding theory with bound $b = 5$ (i.e., set $|a| = 5$), is:

- $\ell = r$
- $\ell < r$
- $\ell \geqslant 0$
- $\ell \leqslant 4$
- $r \geqslant 0$
- $r \leqslant 4$
- $a[0] = e$, $a[1] = e$, $a[2] = e$, $a[3] = e$, $a[4] = e$

We wish to find a "threshold" value β for b such that we can execute our algorithms ConstructFormula, ConstructQuantifiedFormula, and MakeAdequate using ε_β instead of ε, i.e., we execute ConstructFormula(ε_β, *vtt*, *vff*), ConstructQuantifiedFormula(ε_β, *vtt*, *vff*), and MakeAdequate(ν, ε_β, *vtt*, *vff*). Since ε is the union of ε_b for all $b > 0$, we must show how every ε_b can be "represented" in ε_β. We require that every satisfiable valuation V_{ε_b} in ε_b have a representative valuation in ε_β. We will then process this representative, rather than V_{ε_b}. If we can do this for all valuations V_{ε_b} for all $b > \beta$, then we can replace reasoning about the infinite theory ε with reasoning about the finite theory ε_β.

Given β, b such that $\beta < b$, we define the mapping $M_{\beta b} : \varepsilon_\beta \mapsto \varepsilon_b$ as follows, with reference to Definition 7 for the syntax. For $j = 1, \ldots, m$, $g_j(\bar{z}, \beta)$ maps to $g_j(\bar{z}, b)$. For $k = 1, \ldots, n$, $f_k(\bar{z}, a, \bar{v})$ maps to $f_k(\bar{z}, a, \bar{v})$ for each $\bar{v}$ such that

$r_k(\beta, \bar{v})$ holds. Note that $r_k(b, \bar{v})$ also holds, by monotonicity of range predicates. That is, scalar formulae are mapped with β replaced by b. Indexed formulae are mapped as is, since they are also present in ε_b, due to $\beta < b$ and monotonicity of range predicates. $M_{\beta b}$ is injective from ε_β to ε_b, but not onto, since some indexed formulae in ε_b (roughly, those corresponding to the part of the range due to $b - \beta$) do not have an inverse image under $M_{\beta b}$.

Continuing the above example, for ε_b, the scalar formulae are $\ell = r$, $\ell < r$, $\ell \geqslant 0$, $\ell \leqslant b - 1$, $r \geqslant 0$, $r \leqslant b - 1$, and the indexed formulae are $a[0] = e, a[1] = e, \ldots, a[b-1] = e$. Now for ε_β with $\beta < b$, the scalar formulae are $\ell = r$, $\ell < r$, $\ell \geqslant 0$, $\ell \leqslant \beta - 1$, $r \geqslant 0$, $r \leqslant \beta - 1$, and the indexed formulae are $a[0] = e, a[1] = e, \ldots, a[\beta - 1] = e$. Note that the indexed formulae of ε_β are a subset of those of ε_b, while the scalar formulae are not: they result by substituting β for b.

We stated above that every satisfiable valuation V_{ε_b} in ε_b will have a representative valuation in ε_β, which we will process in place of V_{ε_b}. We denote this representative of V_{ε_b} by $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta$, and we define it as a projection onto ε_β, as follows.

Definition 9 (Representative valuation). *Let $V_{\varepsilon_b} : \varepsilon_b \mapsto \{tt, ff\}$ be a valuation of ε_b. Define $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta : \varepsilon_\beta \mapsto \{tt, ff\}$, a valuation of ε_β, as follows:*
for every $f \in \varepsilon_\beta$: $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta(f) = V_{\varepsilon_b}(M_{\beta b}(f))$.

That is, we evaluate a formula f of ε_β in $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta$ by mapping it to V_{ε_b} using $M_{\beta b}$, and then applying V_{ε_b}.

We use $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta$ as the representative of V_{ε_b}. For our algorithms to work correctly under this mapping, we require, for some β and all $b > \beta$:

1. If V_{ε_b} is satisfiable, then so is $V_{\varepsilon_b} \upharpoonright \varepsilon_\beta$. That is, if $[V_{\varepsilon_b}] \neq \emptyset$, then $[V_{\varepsilon_b} \upharpoonright \varepsilon_\beta] \neq \emptyset$.
2. For all $\sigma_b \in [V_{\varepsilon_b}]$, $\sigma_\beta \in [V_{\varepsilon_b} \upharpoonright \varepsilon_\beta]$, the user classifies σ_b and σ_β in the same way, i.e., both in *vtt* or both in *vff*.

Clause 1 can be checked mechanically by submitting it to a SMT solver. Our first attempt to write Clause 1 as a first order wff is:

$$\exists\, \beta > 0, \forall\, b > \beta : \,=\!\!\mid fm(V_{\varepsilon_b}) \;\Rightarrow\; =\!\!\mid fm(V_{\varepsilon_b} \upharpoonright \varepsilon_\beta),$$

where $=\!\!\mid$ means "is satisfiable", and we render each occurrence of $=\!\!\mid$ using existential quantification.

However, the formulae in ε_b depend on b, which presents a problem: $(=\!\!\mid fm(V_{\varepsilon_b}))$ is a wff which depends on b, so that different b give different formulae. Thus, we have to check an infinite set of wff's, one for each b. We deal with this by verifying a single formula which implies each of these wff's.

Define $fm(V_{\varepsilon_b}) \triangleq fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_b})$, where $fm(V^s_{\varepsilon_b})$ is the assignment to the scalar formulae in ε_b, and $fm(V^i_{\varepsilon_b})$ is the assignment to the indexed formulae in ε_b. Likewise define $fm(V_{\varepsilon_\beta}) \triangleq fm(V^s_{\varepsilon_\beta}) \wedge fm(V^i_{\varepsilon_\beta})$, where $V_{\varepsilon_\beta} \triangleq V_{\varepsilon_b} \upharpoonright \varepsilon_\beta$. We wish to check

$$=\!\!\mid (fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_b})) \;\Rightarrow\; =\!\!\mid (fm(V^s_{\varepsilon_\beta}) \wedge fm(V^i_{\varepsilon_\beta})).$$

By monotonicity of range predicates, we have $\varepsilon^i_\beta \subseteq \varepsilon^i_b$, where $\varepsilon^i_\beta, \varepsilon^i_b$ are the subsets of $\varepsilon_\beta, \varepsilon_b$ respectively, consisting of the indexed formulae. Hence $fm(V^i_{\varepsilon_b}) \Rightarrow fm(V^i_{\varepsilon_\beta})$ is logically valid. So

$$=\!\mid(fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_b})) \;\Rightarrow\; =\!\mid(fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_\beta}))$$

is also logically valid. Hence it suffices to check

$$=\!\mid(fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_\beta})) \;\Rightarrow\; =\!\mid(fm(V^s_{\varepsilon_\beta}) \wedge fm(V^i_{\varepsilon_\beta})).$$

We now replace $=\!\mid$ by existential quantifiers:

$$(\exists\, \bar{z}, a : fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_\beta})) \;\Rightarrow\; (\exists\, \bar{z}, a : fm(V^s_{\varepsilon_\beta}) \wedge fm(V^i_{\varepsilon_\beta})).$$

Now the set of scalar formulae $\{g_1(\bar{z}, b), \ldots, g_m(\bar{z}, b)\}$ in ε_b varies with b only inasmuch as the value of b changes. Otherwise the formulae are fixed, and so we have a closed form for all b. Hence we can restore the $\forall\, b > 0$ quantifier:

$$\forall\, b > 0 : ((\exists\, \bar{z}, a : fm(V^s_{\varepsilon_b}) \wedge fm(V^i_{\varepsilon_\beta})) \Rightarrow (\exists\, \bar{z}, a : fm(V^s_{\varepsilon_\beta}) \wedge fm(V^i_{\varepsilon_\beta}))). \quad \mathrm{Th}(\beta)$$

The above is a first order logic formula, for each β. However, different values of β give different formulae, since the set of indexed formulae $\{f_1(\bar{z}, a, \bar{i}) \mid r_1(\beta, \bar{i})\} \cup \cdots \cup \{f_n(\bar{z}, a, \bar{i}) \mid r_n(\beta, \bar{i})\}$, varies with β, since β is referenced in the range predicates $r_1(\beta, \bar{i}), \ldots, r_n(\beta, \bar{i})$. Hence we do not have a closed form for all β. So, we cannot add a $\exists\, \beta$ quantifier at the beginning, since the form of the formula changes with β. So, we check validity of $\mathrm{Th}(\beta)$ for values of β starting from 1 and incrementing. Hence, we find the smallest value of β which works, as desired.

Clause 2 must be assumed as an axiom, since it is a restriction on user behavior:

> **User-Consistency**: Let $b > \beta$, and let $V_{\varepsilon_\beta} = V_{\varepsilon_b} \restriction \varepsilon_\beta$. Then, for every $\sigma_b \in [V_{\varepsilon_b}]$ and every $\sigma_\beta \in [V_{\varepsilon_\beta}]$, the user assigns the same classification (*vtt* or *vff*) to σ_b and σ_β.

From Clauses 1 and 2, we obtain:

> for all $b \geqslant \beta$, if some $\sigma_b \in [V_{\varepsilon_b}]$ exists, then some $\sigma_\beta \in [V_{\varepsilon_\beta}]$ exists, and the user gives the same answers to the queries "is σ_b in *vtt* or in *vff*?" and "is σ_β in *vtt* or in *vff*?"

Hence we can present "is σ_β in *vtt* or in *vff*?" to the developer rather than "is σ_b in *vtt* or in *vff*?"

Example: array search. Let V_{ε_b} be an assignment to $\ell = r$, $\ell < r$, $\ell \geqslant 0$, $\ell \leqslant b-1$, $r \geqslant 0$, $r \leqslant b-1$, $a[0] = e, a[1] = e, \ldots, a[b-1] = e$.

Let V_{ε_β} be an assignment to $\ell = r$, $\ell < r$, $\ell \geqslant 0$, $\ell \leqslant \beta - 1$, $r \geqslant 0$, $r \leqslant \beta - 1$, $a[0] = e, a[1] = e, \ldots, a[\beta - 1] = e$.

$\mathrm{Th}(\beta)$ states that if V_{ε_b} is satisfiable, then so is V_{ε_β}. Suppose that V_{ε_b} assigns true to $\ell < r$, $\ell \geqslant 0$, $r \leqslant b-1$, and truth values to other formulae so that V_{ε_b} is satisfiable. Then V_{ε_β} assigns true to $\ell < r$, $\ell \geqslant 0$, $r \leqslant \beta - 1$, and must also be satisfiable. This requires $\beta \geqslant 2$. Suppose we include $\ell < r - 1$ in ε, e.g., to require at least one array element between the left and right boundaries, then we would have $\beta \geqslant 3$. We validated this (with $\ell < r - 1$ included) by composing Th(β) manually and submitting to Z3, with values 1,2,3 for β. Th(β) was not valid for 1,2, and was valid for 3, as expected.