Specification-Based Sketching with Sketch#

Hesam Samimi
University of California, Los Angeles
hesam@cs.ucla.edu

Kaushik Rajan
Microsoft Research India
krajan@microsoft.com

ABSTRACT

We introduce a new tool employing the *sketching* synthesis technique in programs annotated with declarative contracts. While *Sketch*, the original sketching tool, reasons entirely on imperative code, *Sketch#* works on top of the full-fledged specification language Spec#. In such a language, high-level specifications in the form of pre- and postconditions annotate code, which can be formally verified using decision procedures. But once a given method's implementation is verified, there is no need to look inside its body again. An invocation of the method elsewhere simply implies its specified postcondition. The approach widens the scalability of the sketching technique, as reasoning can be done in a modular manner when specifications accompany implementations.

This paper describes our implementation of Sketch# on top of Spec# and its program verifier *Boogie*. We also recount our experience applying the tool to aid optimistic parallel execution frameworks, where we used it to discover and verify operation inverses, commutativity conditions, and operational transformations for several data structures.

Categories and Subject Descriptors

D.1.2 [**Program Techniques**]: Automatic Programming; D.2.4 [**Software/Program Verification**]: Programming by Contract

General Terms

LANGUAGES, DESIGN, VERIFICATION

Keywords

Program Synthesis, Sketching, Specification Languages, Contracts, Optimistic Parallelization

1. INTRODUCTION

Sketching [10, 11] is a synthesis technique where the user specifies the desired functionality for the final implementation of a particular operation, along with a *sketch*, an incomplete implementation that omits certain low-level details that are often easy to get wrong. An automatic synthesis tool is then used to try to search for pieces of code that complete the implementation, while satisfying the specifications.

The main advantage of this idea is that it scales better than full synthesis, as it is up to the programmer how much of the final implementation to fill in. A source of problem, however, is that the *Sketch* tool currently cannot reason in the modular manner that, for instance, recent program verifiers supporting high-level specifications can. There are two ways that functionality can be specified in Sketch: using low-level assertions, or using function equivalence, i.e. asserting the operation being synthesized must behave as another already implemented method. However, low-level program assertions are often not expressive enough to describe interesting specifications, forcing the user to write imperative code to express the desired functionality. This in turn further complicates the synthesis conditions as large pieces of code, usually with loops, get into the reasoning. Consequently, the applicability of the sketching technique in real-world applications is reduced.

In order to improve its scalability, we designed a new tool that enables sketching in the presence of declarative contracts. While the original sketching tool reasons entirely on imperative code, *Sketch#* works on top of C#'s full-fledged specification and verification framework *Spec#* [1]. The result is that (1) high-level specifications in the form of pre- and postconditions can annotate code, and thus more interesting properties and sketches can be expressed declaratively, (2) the technique scales better as we take advantage of the modular reasoning power that is present in this program specification and verification environment.

To evaluate Sketch#, we utilize it to aid the development of programs for speculative parallelization frameworks. Such frameworks rely on precise semantic properties such as operation inverses, commutativity conditions [9], and operational transformations [3] to parallelize or distribute applications. We used the tool to discover and verify these properties for various data structures, an error-prone task if done by hand.

After providing an overview of the various components of Spec# development framework (Section 2), introducing Sketch# by example (Section 3), and discussing its implementation strategy (Section 4), we describe our experience applying Sketch# as a resource for speculative parallel runtimes (Section 5). Sketch# is built as a fork of Spec# compiler, and available at

`http://www.cs.ucla.edu/~hesam/sketchsharp`

2. BACKGROUND

Spec# Specification Language

JML [6] for Java and Spec# for C# are among the most comprehensive specification languages today. These languages allow Java and C# programs to be annotated with expressive specifications, and support first-order logic, ownership specifications, frame conditions, and much more. As a result, interesting properties can be expressed declaratively and without having to write imperative code.

The need for specification languages is not just a matter of convenience. A major source of strength for these languages is that they plug into a program verifier at the back-end. This enables the user to automatically and formally prove that a program does in fact meet its specifications, or else obtain a counterexample where it fails. But once the implementation of a particular operation is proven correct, there is no need to look inside the implementation for the purposes of reasoning beyond this operation. An invocation of the operation elsewhere, for instance, simply implies its specified postcondition. This modular form of reasoning is key in enabling these tools to deal with the level of complexity and the widespread usage of libraries in today's software. Sketch# is designed with this goal in mind.

Boogie Verifier and Z3 SMT Solver

The Spec# language is powered in the back-end by the program verifier *Boogie* [7]. Boogie translates the C# source along with its Spec# annotations into verification condition formulas, which are then passed down to the theorem prover *Z3* [2] to extract a counterexample. If none is found the implementation is validated. Z3 is a state of the art Satisfiability Modulo Theory (SMT) solver, making C#/Spec# a powerful and efficient development tool with high-level specifications and automated verification capabilities.

3. USING Sketch#

Sketch# is an extension over the full-fledged specification language Spec#. It extends the syntax to enable program sketching for C#, i.e. it allows the user to leave out parts of program as *holes*, letting the tool find values for them that are proven correct with respect to the given specifications. We introduce Sketch# by an integer `Stack` example.

3.1 An Example

Our starting point is the Spec# code for `Push` and `Pop` operations for the `Stack` class in Figure 1, whose implementations are not shown, yet already validated using the verifier Boogie. Let us start with a trivial task of discovering the inverse operation for `Push(int)`. The `Equals` method is specified to be used in defining the semantics of inverse.

The inverse operation op^{-1} asserts that for any state S of the object, it undoes the effects of *op*. In other words,

$$\forall S.\{op^{-1}(op(S)) = S\}$$

The stub in Figure 2 demonstrates how the inverse semantics can be specified in Spec#. It assumes two equal stacks `s1` and `s2` are given (`requires` clause is a precondition). It then applies to `s1` only a `Push(val)` operation (`val` may have arbitrary value), followed by `Push_Inverse_Sketch(val)`, which is the inverse operation we are sketching. Assuming the inverse operation is implemented correctly, the resulting

```
class Stack {
    public int size, int[]! elements;
    invariant size >= 0;
    invariant size < elements.Length;

    [Operation]
    void Push(int val)
        modifies size, elements[*];
        ensures elements[0] == val;
        ensures forall {int i in (0 : old(size));
            elements[i+1] == old(elements[i])};
        ensures size == old(size) + 1;

    [Operation]
    int Pop()
        modifies size, elements[*];
        requires size > 0;
        ensures result == old(elements[0]);
        ensures forall {int i in (1 : old(size));
            elements[i-1] == old(elements[i])};
        ensures size == old(size) - 1;

    bool Equals(Stack s)
        ensures result <==> size == s.size &&
            (forall {int i in (0 : size);
                elements[i] == s.elements[i]});
}
```

Figure 1: `Stack` in Spec#. Method bodies are not shown.

```
void Inverse_Push(Stack s1, Stack s2, int val)
    requires s1.Equals(s2);
    ensures s1.Equals(s2); {
    s1.Push(val); s1.Push_Inverse_Sketch(val);
}

[Inline]
void Push_Inverse_Sketch(int val) {
    if (?!) Push(?!(val)); else if(?!) Pop();
}
```

Figure 2: Sketch for `Push` Inverse (*bottom*), and the stub that describes its semantics (*top*).

```
void Push_Inverse_Sketch(int val) { ?!{}; }
```

Figure 3: Short-form syntax for the sketch of `Push` Inverse.

state should be the same as the starting state, i.e. `s1` and `s2` must have stayed equals and unchanged (`ensures` clause is a postcondition).

Below the stub, we show the sketch used for finding the inverse. The sketched method is marked as `[Inline]` so that, for the purpose of reasoning, the body of the method is inlined at the location it is called.

The `?!` symbol denotes a free boolean or integer sketch variable depending on the inferred type, whose value should be decided by synthesis. We use a boolean sketch variable as a non-deterministic choice of the operation to call. The integer argument for the `Push` operation is also synthesized as a sketch representing a linear combination of the integer variables that are in scope, namely the only local variable `val`. This is denoted by the `?!(val)` syntax, translating to `(?!  * val + ?!)`.

Now that we have provided the sketch and described its semantics, we can compile the source as usual. Once the Sketch# compiler detects sketches are present, it utilizes the SMT solver to find a set of assignments to the sketch variables that makes the complete program verify. One correct synthesis for the example above is:

`if (`*false*`) Push(0 * val + 0) else if (`*true*`) Pop()`

This example included multiple calls to operations. If sketching was to be done purely on code, the synthesis would have had to consider the bodies of these operations, which include loops, multiple times. In specification-based sketching, on the other hand, only the postconditions matter here.

3.2 Sketch Syntax

Sketch# supports a special sketching syntax for synthesizing a sequence of conditional invocations of finite number of operations. That is, users can (a) mark methods as operations (b) provide high-level hints for sketches involving calls to those operations. Table 1 summarizes the types of sketch variables supported in the tool.

We have already seen the *constant* sketch variable `?!`, as well as the *linear combination* sketched expression `?!(...)`, whose translations depend on the inferred types. Each occurrence of a sketch variable implies a fresh one.

The *call* sketch `?!{...}` is used to sketch a non-deterministic choice for a sequence of operation invocations. The call can be guarded, in which case the sketch is translated as an `if` statement or a conditional expression, depending on its type. Translation of a call sketch depends on the optional parameters specified by the user. These are listed in Table 2.

Let's assume for the previous example, we expected the inverse for `Push` to be a single call to either of the two existing operations. We also did not expect the choice of the inverse operation to depend on the argument to `Push`. This intuition is specified with parameter settings `?!{calls: 1, branches: 1}`. These happen to be default values, and thus unnecessary to specify. Figure 3 represents the short-hand notation for the sketch of `Push` Inverse in Figure 2, as the set of available operations and their formal declarations, as well as variables in scope are all known at compiling time.

Symbol	Type	Translation
`?!`	*int / bool*	`?!`
`?!(`v_1`, `v_2`, ...)`	*int*	`?!*`v_1`+?!*`v_2`+...+?!`
`?!(`v_1`, `v_2`, ...)`	*bool*	`?!*`v_1`+?!*`v_2`+...+?!  >= 0`

`?!{`*Params**`}`	*stmt*	`if (?!(...))`
		`  if (?!)  `Op_1`(?!(...),...)`
		`  else if (?!)  `Op_2`(...)`
		`  ...`
		`else if (?!(...)) ...`
		`...`

`?!{`*Params**`}`	*expr*	*cond. expr equiv. to above*

Table 1: Sketch syntax in Sketch#. *Optional *Params* list described in Table 2.

Parameter	Semantics	Default Value
`branches:` n	*# if branches*	1 *(unguarded)*
`calls:` n	*# calls upper-bound*	1 *(single call)*
`ops:` {...}	*possible operations*	*all operations*
`args:` {...}	*args to synthesize*	*all args*
`vars:` {...}	*vars that can appear in synthesized exprs*	*vars in scope*
`conjuncts:` n `disjuncts:` n	*# of* $\wedge/\vee$ *in synthesized conditions*	0

Table 2: Optional parameters for *call* sketch: `?!{}`

4. IMPLEMENTATION

Enabling sketching on top of Spec#/Boogie did not require much effort. Boogie already generates the verification conditions, a set of logical formulas describing the correctness criteria for a program, that can be handed to SMT solver Z3 for either proof of validity, or else a counterexample. This section describes a few modifications that had to be made, in order to extend the Boogie verifier for synthesis.

Turning a Verifier into a Synthesizer

Given the set of input variables $I_1, ..., I_n$ and the verification condition $f_{vc}(I_1, ..., I_n)$, the verification formula is: $\forall I_1, ..., I_n | f_{vc}$, whose negation is passed to the SMT solver. If this negated formula is satisfiable, a counterexample is found and the program fails to verify. To do synthesis, given that $X_1, ..., X_m$ is the set of present sketch variables in the program, the above formula is modified as: $\exists X_1, ..., X_m \forall I_1, ..., I_n | f_{vc}$.

While making this modification to the formula in the code is trivial, solving it is not. As this formula contains both universal and existential quantifiers, it cannot be handled by a typical SMT solver. Consequently, we followed Sketch to implement the *Counter-Example Guided Inductive Synthesis* (*CEGIS*) [10] loop. In such a case just SMT solving suffices.

CEGIS makes synthesis via SAT solving possible by solving for only a finite (k) number of input sets. With a finite set of inputs, the verification formula can be instantiated (and simplified) for each particular input set, clearing the left side of the formula of the universal quantifier: $\exists X_1, X_2, ..., X_m | f_{vc_1} \wedge f_{vc_2} \wedge ... \wedge f_{vc_k}$.

The starting point for the iterative process is one valid input set. This starting input example itself can be obtained by asking the SMT solver for an assignment of input variables satisfying any user-specified preconditions. The CEGIS loop then starts off with a call to the SMT solver to find an assignment of sketch variables that satisfy the above formula with k=*1*. The candidate synthesized values are then inserted in the sketch to make a complete program, which is then verified with Boogie as usual. If failed, Z3 provides an input counterexample, which is added as a new input set and the loop repeats with k+*1* (see [10], appendix).

5. EVALUATION: OPTIMISTIC PARALLEL RUNTIMES

Speculative execution plays a crucial role in both parallel and distributed systems. Optimistic runtimes rely on the programmer to provide auxiliary information associated with operations such as conditions under which two operations commute, operation inverses, and operational transforms [3]. In the absence of these definitions such runtimes cannot be used to parallelize or distribute applications.

Many optimistic parallelization frameworks (e.g. Galois [5] and CommSets [8]) exploit conditional commutativity and inverse operations to parallelize programs. Commutativity conditions [9] are used to check if operations that were executed speculatively in parallel indeed commute. Inverse operations are then used to roll back one of the operations in case the check fails. For valid parallelization the programmer has to provide for each operation (a) an inverse operation (b) a set of commutativity conditions, one for each operation defined on the object.

Finding the right operations and conditions even on simplest data structures can be error-prone [3, 4], so the sketching technique can become an invaluable resource for such systems to obtain these properties precisely and with sound correctness guarantees. We built the tool described here as an interactive program synthesis framework that can be used by developers to discover and validate these precisely.

While Sketch# can be used as a general specification-based synthesis tool, we have focused on how it may benefit speculative parallelization frameworks. The sketches for such case studies all follow a similar pattern. Given a finite set of fully specified methods designated as operations, we would like to synthesize a sequence of calls to those operations, such that a given specification is satisfied.

5.1 Sketching Inverses, Commuting Conditions

In the `Stack` example of Figure 2, we saw how to use Spec# specifications and a sketch to discover the inverse of an operation.

A commutativity condition for op_1 with respect to a second operation op_2 is a necessary and sufficient predicate over the arguments of op_1, op_2, i.e. $pred(op_{1.args}, op_{2.args})$, that asserts the following.

$$\forall(op_1, op_2, S).\{pred(op_{1.args}, op_{2.args}) \iff op_1(op_2(S)) = op_2(op_1(S))\}$$

Figure 4 presents how a sketch for the commutativity conditions can be verified. Given a sketched condition `Commute_Cond_Sketch(op1, op2)` the stub ensures the predicate represents both a necessary and sufficient commuting condition for the two operations op_1 and op_2.

It should be noted that we only seek commutativity conditions that only depend on operations themselves, and not the state of objects. In other words, it is possible that for some initial state, two non-commuting operations actually commute. More preconditions are added to reject these state-dependent commutativity conditions (not shown, see [4]).

5.2 Sketching Operational Transformations

Distributed systems explore an even more aggressive form of optimistic execution referred to as optimistic replication. Such systems maintain local object replicas at each machine and allow these replicas to be locally updated without

```
void Commutativity_Condition
    (Object s1, Object s2, Op op1, Op op2)
    requires s1.Equals(s2);
    ensures Commute_Cond_Sketch(op1, op2) <==>
        s1.Equals(s2); {
    s1.Apply(op1); s1.Apply(op2);
    s2.Apply(op2); s2.Apply(op1);
}
```

Figure 4: Stub for Commutativity Condition Soundness and Completeness.

```
void T(Object s1, Object s2, Op op1, Op op2)
    requires s1.Equals(s2);
    ensures s1.Equals(s2); {
    s1.Apply(op1); s1.Apply(T_Sketch(op2, op1));
    s2.Apply(op2); s2.Apply(T_Sketch(op1, op2));
}
```

Figure 5: Stub for Operational Transformation.

synchronization. Concurrent operations are then exchanged over the network. At each machine the remote operations are first transformed with respect to local operations using programmer provided operational transforms (OT) [3] before they update the local state. To guarantee eventual consistency the programmer needs to provide a transformation function T such that

$$\forall(op_1, op_2, op'_1, op'_2, S).\{op'_1 = T(op_1, op_2) \wedge op'_2 = T(op_2, op_1) \implies op'_2(op_1(S)) = op'_1(op_2(S))\}$$

For the `Stack` example $T($`Push(`v`)`,`Pop()`$)$ = `Push(`v`)` and $T($`Pop()`,`Push(`v`)`$)$ = `Pop();Pop();Push(`v`)` satisfy the formula above.

Given such transformation functions, optimistic replication frameworks can guarantee eventual consistency. But finding the correct operational transformation has proven extremely difficult for most common data structures [3], preventing the approach to gain much traction in practice. Figure 5 presents the corresponding stub we have used to sketch the OT for several examples.

5.3 Results

A summary of our experiments on several data structures appears in Table 3, showing the particular properties being synthesized, the sketch, as well as the results with running times. Integer sketch variables are initially restricted to 2-bits, essentially used as non-deterministic choice operators in the expression `?! *` $v1$ `+ ?! *` v_2 `+` ... . Whenever this resulted in an unsatisfiable sketch, we expanded the variable bounds to 8-bits. We currently do not simplify the CEGIS synthesis formula of Section 4 for each input set and simply instantiate the same formula k times, so synthesis times can be improved.

Not surprisingly, the tool reports an unsatisfiable sketch in the case of `Delete(i)` inverse. This, unfortunately, does not prove the non-existence of an inverse, but rather that no solution exists within the space of all programs covered by the given sketch.

Type	Class	Operation	Sketch	Synthesized	sec.
Inverse	`Stack`	`Push(`v`)`	*?!{}*	`Pop()`	0.6
		`int` r `= Pop()`	*?!{}*	`Push(`r`)`	1.0
	`ArrayList`	`Insert(`i`,`v`)`	*?!{}*	`Delete(`i`)`	1.3
		`Delete(`i`)`	*?!{}*	*UNSAT*	1.5
Operational Transform (OT)	`Stack`	`Push(`v_1`)/Push(`v_2`)`	*?!{branches: 2, calls: 3}*	`if(`$v_1 \geq v_2$`) Push(`v_1`) else {Pop();Push(`v_1`);Push(`v_2`)}`[1]	9.5
		`Pop()/Pop()`	*?!{}*	`Pop()`	1.1
		`Push(`v_1`)/Pop()`	*?!{}*	`Push(`v_1`)`	2.3
		`Pop()/Push(`v_1`)`	*?!{calls: 3}*	`Pop();Pop();Push(`v_1`)`	
	`ArrayList`	`Insert(`i_1`,`v_1`)/ Insert(`i_2`,`v_2`)`	*?!{branches: 3, ops:{Insert}, args:{0}, conjuncts:2}*	`if (`$i_1 > i_2 \vee i_1 = i_2 \wedge v_1 \geq v_2$`) Insert(`$i_1+1, v_1$`) else Insert(`$i_1, v_1$`)`[1]	240
		`Insert(`i_1`,`v_1`)/ Delete(`i_2`)`	*?!{branches: 2, ops:{Insert}, args:{0}, vars:{i_1, i_2}}*	`if(`$i_1 > i_2$`) Insert(`$i_1 - 1$`,`v_1`) else Insert(`i_1`,`v_1`)`	26
		`Delete(`i_1`)/ Insert(`i_2`,`v_2`)`	*?!{branches: 2, ops:{Delete}, vars:{i_1, i_2}}*	`if(`$i_1 \geq i_2$`) Delete(`i_1+1`) else Delete(`i_1`)`	
Commute Cond.	`Stack`	`Push(`v_1`)/Push(`v_2`)`	*?!(conjuncts:1)*	$v_1 = v_2$	70
		`Pop()/Pop()`	*?!*	*true*	1.2
		`Push(`v_1`)/Pop()`	*?!(conjuncts:1)*	*false (state dependent)*	2.0
	`XmlNode`[2]	`InsertAfter(`id_1`,`n_1`)/ InsertAfter(`id_2`,`n_2`)`	*?!(vars:{$id_1, n_1.id, id_2, n_2.id$}, disjuncts:1)*	$id_1 \neq id_2$	67

[1] v_1, v_2 values are compared not because the data enforces an ordering, but to pick an ordering between two ops consistently.

[2] `XmlNode` based on XML Tree implementation at `http://msdn.microsoft.com/en-us/library/system.xml.xmlnode.aspx`

Table 3: Summary of a few synthesized properties.

6. RELATED WORK

This project is based on the sketching technique introduced by Solar-Lezama *et al.* [10]. Some of the features in Sketch tool are not present in the syntax of Sketch#. Most notably, Sketch lets the user specify a sketch as a grammar, whereas Sketch# currently only supports integer or boolean holes, on top of which the syntactic sugars for call sketches are built. There is no support for synthesizing loops, but sketching methods that include loops is possible. To do that, just as in the case of verification, loop invariants must be specified.

We also tested the examples in our case study in Sketch, and found that Sketch# is significantly more scalable (orders of magnitude in some cases) when compared on equal sizes of space (number of bits for inputs and counterexamples). This is because Sketch# reasons on top of declarative Spec# [1] contracts, and is powered by the efficient Z3 SMT solver [2]. To the best of our knowledge, Sketch# is first to enable synthesis within a mainstream specification language.

Kim *et al.* recently used high-level specifications over abstract states of data structures to formally prove the correctness of operation inverses and commutativity conditions for a large number of data structures and operations [4]. Their study for commutativity conditions was more comprehensive, as they also considered conditions on state as well as operations. Researchers in distributed applications have done the same for operational transformations [3]. Both studies involved the verification of properties, not synthesis.

Synthesizing program inverses, on the other hand, has seen a great deal of attention. Most recently, Srivastava *et al.*'s *PINS* system [12] uses symbolic execution to refine the space of instantiations for the inverse template, and uses the original program as a heuristic for mining the template. While PINS works on imperative code, our approach takes a specification-based angle and works at a higher level. For instance, given the stack operations along with their specs, Sketch# can find an invocation that inverts `Push(`v`)`. PINS, on the other hand, may be used to derive the implementation of `Pop()`, given the code for `Push(`v`)`.

At the expense of an oversimplification, we say while Sketch and PINS are used to synthesize *inside* of an operation (its implementation), Sketch# is more likely to scale when considering *beyond* it (when used in a larger context).

7. OUTLOOK

We have presented Sketch#, a specification-based tool for performing program sketching. Our first contribution in this paper is showing that enabling the sketching synthesis technique on top of a high-level specification language broadens its applicability due to the added modular form of reasoning. This is essential as real-world applications have many complex but decomposable components, and the usage of library code is ubiquitous.

Secondly, we have applied the tool to synthesize and verify inverse operations, commutativity conditions, and operational transformations for several common data structures. While researchers have extensively studied synthesis of inverses, to the best of our knowledge, this synthesis study is a first for commutativity conditions and operational transformations. Providing such precise and provably-correct properties is essential for the success of the emerging speculative parallel runtimes.

We came across a problem during our experiments with the synthesis of operational transformations. There exists a more restrictive definition of OT than the formula given in section 5.2, that is necessary for a distributed application that is truly free of any global ordering of operations (see [3]). For a pair of `Insert`(i, v) operations in a `List` data structure, a transformation satisfying such definition is not known yet. We attempted to find this illusive transformation, but Sketch# reported unsatisfiability for several sketches we tried. However, it is impossible to try all possible sketches manually. This demonstrated that when synthesis is being used to discover an unknown, template-based synthesis cannot work unless refining the sketch is also aided by a computer. The final aspect of this work is highlighting the need for automated techniques to refine (and expand) synthesis templates. We designed the sketch parameters described before in order to open the possibility of a more mechanical search over a space of possible synthesis templates. The project described here lays out some groundwork towards such a study.

8. REFERENCES

[1] M. Barnett, R. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005, 2005.

[2] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] A. Imine, M. Rusinowitch, G. Oster, and P. Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science*, 2006:167–183, 2005.

[4] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[5] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference, on Programming language design and implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[6] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.

[7] K. R. M. Leino. Specifying and verifying software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 2–2, New York, NY, USA, 2007. ACM.

[8] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[9] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on programming languages and systems*, 19(6):1–47, 1997.

[10] A. Solar-Lezama, G. Arnold, L. Tancau, and R. Bodik. Sketching stencils. In *Proceedings of the 2007 ACM SIGPLAN conference, on Programming language design and implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM.

[11] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference, on Programming language design and implementation*, PLDI '05, pages 281–294, New York, NY, USA, 2005. ACM.

[12] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 2011 ACM SIGPLAN conference, on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

```
int abs(int x)
    requires true;
    ensures x >= 0 ==> result == x &&
            x < 0 ==> result == -x;
{
    if (x >= ?!)
        return x;
    else
        return -x;
}
```

Figure 6: `abs` function specification and implementation sketch.

APPENDIX

A Sample of CEGIS [10] Loop at Work

Consider the simple sketch of the absolute value function in Figure 6. The verification condition for the method using weakest-precondition generation is the following.

$$\begin{aligned}\forall x.\{x\ isInt \implies ((x \geq ?! \implies ((x \geq 0 \implies x = x) \wedge \\ (x < 0 \implies x = -x))) \wedge \\ (x < ?! \implies ((x \geq 0 \implies -x = x) \wedge \\ (x < 0 \implies -x = -x))))\}\end{aligned} \tag{1}$$

which simplifies to:

$$\begin{aligned}\forall x.\{x\ isInt \implies ((x \geq ?! \implies x \geq 0) \wedge \\ (x < ?! \implies x \leq 0))\}\end{aligned} \tag{2}$$

The synthesis formula is:

$$\begin{aligned}\exists ?!.\{\forall x.\{x\ isInt \implies ((x \geq ?! \implies x \geq 0) \wedge \\ (x < ?! \implies x \leq 0))\}\}\end{aligned} \tag{3}$$

and the precondition is:

$$x\ isInt \tag{4}$$

The first step is to use the SMT solver to get a starting input set that satisfies the precondition.

$$(4) \xrightarrow{SMT} x = -55$$

Now the CEGIS loop has a finite number of input sets with *k=1.* The CEGIS synthesis formula becomes the following.

$$\begin{aligned}x_1 = -55 \wedge \\ \exists ?!.\{x_1\ isInt \implies \\ ((x_1 \geq ?! \implies x_1 \geq 0) \wedge (x_1 < ?! \implies x_1 \leq 0))\}\}\end{aligned} \tag{5}$$

The loop starts by asking for a candidate synthesis model:

$$(5) \xrightarrow{SMT} ?! = -12$$

The synthesis model works for our current finite number of inputs, but is it correct in general? Next, we need to verify the complete program by checking the validity of the verification condition formula in (2).

$$\begin{aligned}\forall x.\{x\ isInt \implies ((x \geq -12 \implies x \geq 0) \wedge \\ (x < -12 \implies x \leq 0))\}\end{aligned} \tag{6}$$

$$\neg(6) \xrightarrow{SMT} x = -4$$

Adding the new counterexample to the input set, our new synthesis formula becomes as follows, where the existentially quantified sketch variables `?!` appearing in (5) and (7) must be the same.

$$\begin{aligned}(5) \wedge \\ x_2 = -4 \wedge \\ \exists ?!.\{x_2\ isInt \implies \\ ((x_2 \geq ?! \implies x_2 \geq 0) \wedge (x_2 < ?! \implies x_2 \leq 0))\}\}\end{aligned} \tag{7}$$

Asking for a new candidate synthesis model we get:

$$(7) \xrightarrow{SMT} ?! = 108$$

Again, we need to verify the complete program by checking the validity of the verification condition formula in (2):

$$\begin{aligned}\forall x.\{x\ isInt \implies ((x \geq 108 \implies x \geq 0) \wedge \\ (x < 108 \implies x \leq 0))\}\end{aligned} \tag{8}$$

$$\neg(8) \xrightarrow{SMT} x = 1$$

After adding the new counter example and updating the synthesis formula:

$$\begin{aligned}(7) \wedge \\ x_3 = 1 \wedge \\ \exists ?!.\{x_3\ isInt \implies \\ ((x_3 \geq ?! \implies x_3 \geq 0) \wedge (x_3 < ?! \implies x_3 \leq 0))\}\}\end{aligned} \tag{9}$$

And maybe at this point we finally get the right synthesis.

$$(9) \xrightarrow{SMT} ?! = 0$$

which is verified correct as the following verification condition is valid.

$$\begin{aligned}\forall x.\{x\ isInt \implies ((x \geq 0 \implies x \geq 0) \wedge \\ (x < 0 \implies x \leq 0))\}\end{aligned} \tag{10}$$

$$\neg(10) \xrightarrow{SMT} UNSAT$$

□