SPT: Storyboard Programming Tool

Rishabh Singh* and Armando Solar-Lezama

MIT CSAIL, Cambridge, MA, USA

Abstract. We present SPT, a tool that helps programmers write low-level data-structure manipulations by combining various forms of insights such as abstract and concrete input-output examples as well as implementation skeletons. When programmers write such manipulations, they typically have a clear high-level intuition about how the manipulation should work, but implementing efficient low-level pointer manipulating code is error-prone. Our tool aims to bridge the gap between the intuition and the corresponding implementation by automatically synthesizing the implementation. The tool frames the synthesis problem as a generalization of an abstract-interpretation based shape analysis, and represents the problem as a set of constraints which are solved efficiently by the SKETCH solver. We report the successful evaluation of our tool on synthesizing several linked list and binary search tree manipulations.

1 Introduction

When programmers write data-structure manipulations, they typically have clear high-level visual insights about how the manipulation should work, but the translation of these insights to efficient low-level pointer manipulating code is difficult and error prone. Program synthesis [1, 5, 6] offers an opportunity to improve productivity by automating this translation. This paper describes our tool SPT[1] (Storyboard Programming Tool) that helps programmers write low-level implementations of data-structure manipulations by combining various forms of insights, including abstract and concrete input-output examples as well as implementation skeletons.

Our tool is based on a new synthesis algorithm [4] that combines abstract-interpretation based shape-analysis [2, 3] with constraint-based synthesis [5, 7, 8]. The algorithm uses an abstraction refinement based approach to concisely encode synthesis constraints obtained from shape analysis.

In this paper, we present a high-level storyboard language that allows programmers to succinctly express the different elements that make up a storyboard. The language is more concise than the one described in [4] thanks to the use of inference to derive many low-level details of the storyboard. The paper also describes the architecture of the Storyboard Programming Tool and presents some new results comparing SPT with the Sketch synthesis system.

* Supported by NSF under grant CCF-1116362.

[1] The Storyboard tool and benchmarks are available for download at http:/people.csail.mit.edu/rishabh/storyboard-website/

2 Overview: Linked List deletion

We present an overview of our tool using linked list deletion as a running example. The goal of this manipulation is to delete a node pointed to by a variable `y` from an acyclic singly linked list. The manipulation iterates over the list until it finds the required node and then performs a sequence of pointer assignments to delete the node. SPT synthesizes an imperative implementation of deletion from a high-level storyboard in about two minutes.

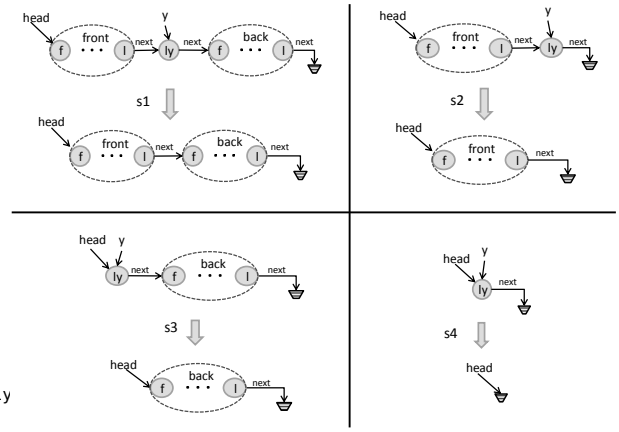
The storyboard that SPT takes as input is composed of *scenarios*, *inductive definitions* and a *loop skeleton*. A scenario describes the (potentially abstract) state of a data-structure at different stages of the manipulation. Each scenario contains at least two configurations: `input` and `output` corresponding to the state of the data-structure before and after the manipulation; a scenario may also contain descriptions of the state at intermediate points in the computation.

The scenarios for linked list deletion are shown in Figure 1(a), and the corresponding visual description of the scenarios is shown in Figure 1(b). The first scenario `s1` describes an abstract input-output example, where the input list consists of two *summary nodes* `front` and `back` and a concrete node `ly` that is to be deleted. In this case, the summary node serves as an abstract representation of a list of arbitrary size. In general, a summary node represents an arbitrary set of nodes; those nodes in the set which are connected to other nodes not in the set are given concrete names and are called *attachment points*. The summary node `front` contains two attachment points `front::f` and `front::l` denoting the first and last elements of the `front` list respectively.

The state configurations are defined using a list of state predicates such as `(head -> front::f)` which denotes that the `head` variable points to the attachment point `f` of `front`. The other scenarios `s2`, `s3` and `s4` correspond to the cases of deleting the last node, the first node and the only node of the list respectively. Notice that there is no scenario corresponding to the case where the node to be deleted is not in the list. That means that the behavior of the synthesized code will be unspecified in such a case.

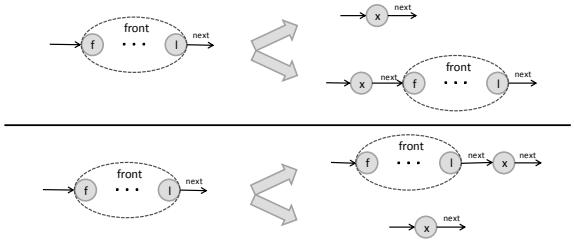
In order for the synthesizer to reason about summary nodes, the user needs to specify their structure. In this case, for example, the user needs to express the fact that `front` and `back` are not just arbitrary sets of nodes; they are lists. In SPT, this structural information is provided inductively through `unfold` rules. The two possible `unfold` rules for the summary node `front` and their corresponding visual description are shown in Figure 2. The rule states that the summary node `front` either represents a single node `x` or a node `x` followed by another similar summary node `front`. The unfold predicate consists of the summary node, the replacement node, the incoming and outgoing edges, and additional constraints that hold after the unfold operation. In SPT, the programmer can also provide `fold` rules to describe to the system how sets of nodes can be summarized by a single node. In most cases, such as the example, the synthesizer can reason about the correctness of a manipulation by using the inverse of the `unfold` rules for summarization, but for some tree algorithms, the synthesis process can be made more efficient by providing explicit `fold` rules for summarization.

```
scenario s1
input: head -> front::f, y -> ly
 front::l.next -> ly,
 back::l -> null, ly.next -> back::f
output: head -> front::f,
 front::l.next -> back::f,
 back::l.next -> null
scenario s2
input: head -> front::f, y -> ly
 ly.next -> null, front::l.next -> ly
output: head -> front::f,
 front::l.next -> null
scenario s3
input: head -> ly,ly.next -> back::f,
 y -> ly, back::l.next -> null
output: head -> back::f,
 back::l.next -> null
scenario s4
input: head -> ly,ly.next -> null,y -> ly
output: head -> null
```



(a) (b)

Fig. 1. Scenarios describing input and output state descriptions for linked list deletion.

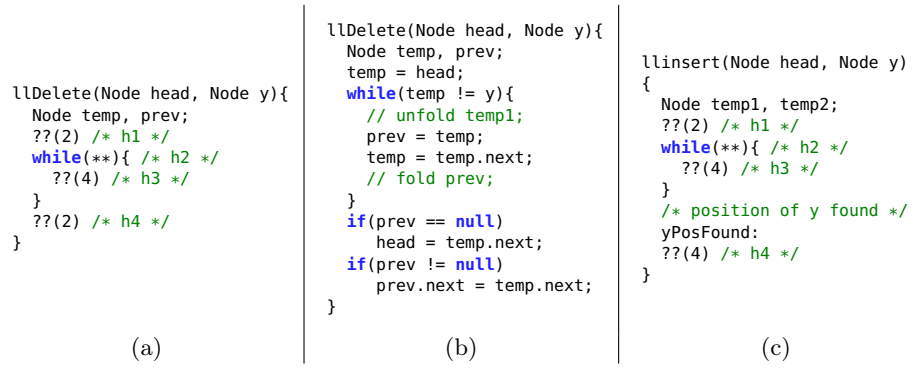



```
unfold front::f x [in (front::f, x)] [out (front::l, x)] ()
unfold front::f x [in (front::f, x)] [out (front::l, front::l)] (x.next -> front::f)

unfold front::f x [in (front::f, x)] [out (front::l, x)] ()
unfold front::f x [in (front::f, front::f)] [out (front::l, x)] (front::l.next -> x)
```

Fig. 2. Two possible unfold definitions for summary node front.

In addition to the scenario descriptions, Spt also requires users to provide a loop skeleton of the desired implementation. This helps the synthesizer focus on implementations that are close to the user's expectations, and also lets them specify intermediate state constraints. The loop skeleton for the running example is shown in Figure 3(a). It consists of a while loop with a set of unknown statements before the loop, in the loop body and after the loop. The unknown statements are denoted by the `??(n)` operator, where n represents the maximum length of the unknown statement block. Spt restricts unknown statements to be of two forms: i) guarded statements of the form `if(**) then ASSIGN`, where $**$ represents a conditional over pointer variables and `ASSIGN` denotes pointer assignments with at most one pointer dereference, and ii) unfold/fold statements of the form `unfold var` (resp. `fold var`) that corresponds to unfolding (folding) the location pointed to by variable `var`.



```
llDelete(Node head, Node y){
  Node temp, prev;
  ??(2) /* h1 */
  while(**){ /* h2 */
    ??(4) /* h3 */
  }
  ??(2) /* h4 */
}
```

(a)

```
llDelete(Node head, Node y){
  Node temp, prev;
  temp = head;
  while(temp != y){
    // unfold temp1;
    prev = temp;
    temp = temp.next;
    // fold prev;
  }
  if(prev == null)
     head = temp.next;
  if(prev != null)
     prev.next = temp.next;
}
```

(b)

```
llinsert(Node head, Node y)
{
  Node temp1, temp2;
  ??(2) /* h1 */
  while(**){ /* h2 */
    ??(4) /* h3 */
  }
  /* position of y found */
  yPosFound:
  ??(4) /* h4 */
}
```

(c)

Fig. 3. (a) The loop skeleton and (b) the synthesized implementation for linked list deletion and (c) the loop skeleton for sorted linked list insertion.

Given the scenarios, recursive definitions and the loop skeleton, SPT synthesizes the imperative implementation shown in Figure 3(b). The `true` conditionals and `skip` statements are removed from the code for better readability.

3 Algorithm

The details of the synthesis algorithm used by SPT can be found in [4]. At a high level, the algorithm first translates the loop skeleton into a set of equations relating the inputs and outputs of all the unknown blocks of code. Let F_i denote the unknown transfer function that maps a set of program states to another set of program states. The relationships between the inputs and outputs of all the transfer functions is captured by a set of equations of the form:

$$(t_0 = In_k) \quad \wedge \quad \forall_{v_i \in (V \setminus v_0)} \quad t_i = F_i(\bigcup_{j \in pred(v_i)} t_j) \tag{1}$$

where $pred(v_i)$ denotes the predecessors of node v_i in the CFG. In_k denotes the input state constraint for the k^{th} scenario, and the goal is to find F_i such that $t_N = Out_k$ for each scenario k. The system works by representing the F_i as parameterized functions and solving for the parameters by using a counterexample guided inductive synthesis [4].

4 Tool Architecture

The architecture of Storyboard Programming Tool consists of four major components as shown in Figure 4:

A. Storyboard Parser: The parser takes as input a storyboard description (`*.sb`) written in the storyboard language and translates it into our intermediate constraint language that consists of a set of prolog predicates (`*.pl`). The

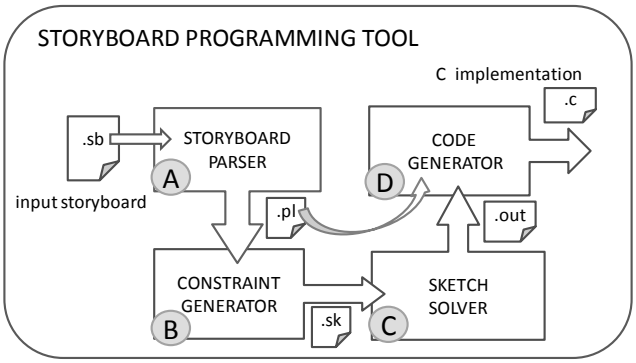


Fig. 4. The architecture of Storyboard Programming Tool.

storyboard language is more concise than [4] because the parser also performs some type inference to infer variables, locations, selectors, and summary nodes.

B. Constraint Generator: The constraint generator translates the prolog predicates into a sketch constraint file (`*.sk`). SKETCH provides an expressive language to define the shape analysis problem, allowing us to use high-level language constructs such as functions, arrays and structures to model the abstract state, as well as holes to model the unknown transfer functions. The problem is encoded in a way that leverages the counterexample guided inductive synthesis algorithm of sketch to avoid having to represent sets of shapes explicitly.

C. Sketch Solver: The SKETCH solver solves the sketch constraint file and produces an output (`.out`) file where all unknown function choices are resolved.

D. Code Generator: The code generator takes as input the output generated by the SKETCH solver and completes the loop skeleton (`.c`) by mapping the function choice values to their corresponding program statements and conditionals using the intermediate constraint file (`*.pl`).

5 Experiments and Tool Experiences

We evaluated the tool on several linked list and binary search tree manipulations as well as AIG (And-Inverter Graph) insertion. The details about the experiments and benchmarks can be found in [4]. We present here a comparison with the SKETCH tool on a small sample of benchmarks.

Comparison with SKETCH: Table 1 shows the running times of the Storyboard tool with the SKETCH system. We can see that SKETCH is faster, but can only perform bounded reasoning (N=5) and quickly times out for larger values of N. On the other hand, the storyboard tool performs unbounded analysis using abstract interpretation. However, the biggest difference we found was in the usability of the tools. We had to spend almost three hours for writing a spec in SKETCH for these manipulations. For writing a sketch, one has to write a converter (and its inverse) for converting (resp. translating back) an array to that

Benchmark	Sketch (N=5)	Storyboard
ll-reverse	18s	1m40s
ll-insert	31s	2m3s
ll-delete	25s	2m8s
bst-search	39s	2m51s
bst-insert	3m35s	3m12s

Table 1. Performance comparison with Sketch.

data structure. Then one has to use quantified input variables for writing tricky specs. In our storyboard tool, we only have to provide input-output examples which in our experience was a lot more natural. We now present some of our other experiences in handling complicated manipulations with the tool.

Intermediate State Configurations: Our tool allows users to write intermediate state configurations to reduce the search space and enable the synthesizer to synthesize more complex manipulations. The user can label a program location in the loop skeleton and provide the state description at that point using the `intermediate` keyword as part of the scenario description. For example in the case of insertion in a sorted linked list, we add an additional insight based on the fact that the loop skeleton for the insertion (Figure 3(c)) is performing two tasks: first to find a suitable location for inserting the node `y` and the second task of inserting `y` into the list. In the abstract scenario, we provide an intermediate state configuration at the label `yPosFound` in which two variables point to the two locations between which the insertion is to be performed.

The Spt tool illustrates a new approach to synthesis namely *Multimodal Synthesis*, where the synthesizer takes input specification in many different forms such as concrete examples, abstract examples, and implementation insights, and synthesizes code that is provably consistent with all of them. We believe this idea of multimodal synthesis has applicability in many other domains as well.

References

1. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.
2. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.
3. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.
4. R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *FSE*, pages 289–299, 2011.
5. A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS, UC Berkeley, 2008.
6. A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.
7. S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.
8. M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.