Automatic Synthesis of a Voting Machine Design

Lili Dworkin
Haverford College

Sanjit Seshia
University of California, Berkeley

Wenchao Li
University of California, Berkeley

Haverford College Computer Science Tech Report 2010-02
Archived November 2010

Abstract

Synthesis from specification has long been considered the "holy grail" of the design process. Instead of implementing a design by hand, we prefer to provide a high-level description of the design, and have the implementation automatically generated. Synthesis also eliminates the need for verification, since the synthesized design is correct-by-construction. We apply two methods of synthesis to the design of an accurate and reliable voting machine. In the first approach, we use an automata-theoretic approach to perform synthesis from a linear temporal logic specification. In the second approach, we use a modified version of the L* algorithm to perform synthesis from scenarios. We compare the two approaches in terms of effort required and results produced, and suggest a third alternative combining aspects of each.

1 Introduction

Automatic synthesis of a digital design is a desirable alternative to the manual design process. The traditional approach to design is first to implement the design in a *hardware description language* (HDL), and then perform *verification*. An HDL is a programming language used to describe electronic

circuits. Verification refers to the process of checking whether an implementation satisfies its *specification*, a set of requirements that the program must fulfill in order to be correct. The motivation for automatic synthesis stems from the difficulties and costs associated with both the implementation and verification steps. A hand-written implementation of the design is susceptible to human error, and must be rewritten every time the specification is updated. Additionally, verification is often the most difficult and time-consuming part of the design process.

An alternative approach is to automatically synthesize the design from its specification, thereby eliminating the need for manual implementation. We explore two methods, both of which synthesize a design, but which differ in the input required. We apply both variations to the problem of designing an accurate and reliable voting machine. Voting machine synthesis is an especially compelling problem, because there are few ways to guarantee that the machine will work both accurately and in the expected manner. Thus, we are motivated by the goal to synthesize a design that is guaranteed to work properly. Specifically, we seek to synthesize the voting machine described by Sturton et al [7]. This simple machine can be used for an election with multiple contests and multiple candidate selections in each. The display screen includes buttons for navigating between contests, choosing candidates, and casting a vote. We implement the design as a finite state machine, in which each button press corresponds to a transition between states.

In the first synthesis approach we attempt, the prerequisite for synthesis is a complete formal specification. The specification is usually given in *temporal logic*, a language for representing properties of a system in terms of time [6]. The benefit of synthesizing directly from a formal specification is that we eliminate the need for verification, because the synthesis algorithm guarantees that the resulting design satisfies its specification. In this case, we say the synthesized design is "correct-by-construction." Our task then is simply to write a specification that accurately describes the desired behavior. If such a specification proves too difficult to write, we turn to our second approach, which does not require a specification. The prerequisite is instead a set of *scenarios*, which are traces of the correct execution of the program. If we provide a set of scenarios that describe all correct program behaviors, the synthesis will output a design meeting the specification.

The contributions of this paper are the following:

- Assuming the structural properties of the voting machine are satisfied,

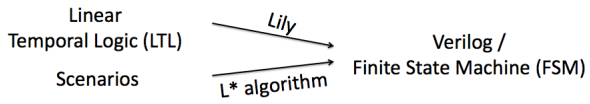


Figure 1: Synthesis Approaches

we can synthesize an implementation both from a linear temporal logic specification and from scenarios.

- Given a box diagram of the voting machine, a set of scenarios satisfying the coverage criteria described by Sturton et al [7] is sufficient for synthesis of a hierarchical state machine.
- We present a comparison of the time and effort required to use each synthesis approach, as well as evaluating the quality of the results produced.
- We discuss current work being done on a promising third approach that combines aspects of both LTL and scenario-based synthesis.

We proceed as follows: Section 2 reviews background material We discuss LTL synthesis in section 3, synthesis from scenarios in section 4, and compare the two approaches in section 5. In section 6 we explore the possibility of a combined approach. Section 6 concludes and presents directions for further work.

2 Preliminaries

2.1 Finite State Transducer

A *finite state transducer* (FST) is a finite state machine with both an input and output tape. We use an FST to represent the voting machine design. We formally define an FSM as the 6-tuple $(I, O, S, \delta, \rho, s_{init})$ where:

- I is the set of inputs
- O is the set of outputs
- S is the set of states

Operator	Meaning
X ϕ	Next ϕ
G ϕ	Always ϕ
F ϕ	Eventually ϕ
ϕ U φ	ϕ Until φ

Table 1: LTL Modal Operators

- $\delta : S \times I \to S$ is the transition function
- $\rho : S \to O$ is the output function
- s_{init} is the initial state

2.2 Linear Temporal Logic

Linear temporal logic (LTL) is a type of temporal logic used to express properties of a system over linear time. LTL formulas are evaluated over paths or traces of a system's exeuction. The syntax consists of propositions, Boolean algebra, and *temporal modal operators*, as explained in Table 1. Formally, the language of LTL is as follows. Let P be a set of propositions defined over a system's input and output signals. LTL formulas are defined recursively:

- $True$ and $False$ are formulas
- $p \in P$ are formulas
- If ϕ and φ are formulas, then $\neg\phi$, $\phi \wedge \varphi$, $\phi \vee \varphi$, $\phi \to \varphi$, and $\phi \leftrightarrow \varphi$ are formulas
- Additionally, if ϕ and φ are formulas, then Xϕ, Gϕ, Fϕ, and ϕUφ are formulas

There are two main types of LTL properties: *safety* and *liveness* properties. Safety properties ensure that something "bad" never happens, and have the general form G$\neg\phi$. Liveness properties ensure that something "good" always happens, and have the general form G(F(ϕ)) or G($\varphi \to$ F(ϕ)).

2.3 Angluin's Algorithm

Angluin [2] presented an algorithm, termed as L*, for learning a deterministic finite automata (DFA) that accepts a regular language $\mathcal{L}$ over the alphabet Σ. The algorithm is *online* meaning it can receive and process input while running. An *offline* algorithm, on the other hand, can only use a set of inputs provided initially.

Like many online algorithms, L* works by interacting with a *teacher* and an *oracle*. The teacher answers *membership queries*, in which L* asks whether the target DFA accepts a certain string s. The oracle answers *equivalence queries,* in which L* asks whether a conjecture DFA is correct. If the conjecture is incorrect, the oracle returns a *counterexample*, which is a string s that is accepted by either the conjecture or target DFA, but not the other.

The algorithm works by building a set S of string prefixes and a set E of suffixes. For every prefix-suffix combination, the algorithm asks a membership query and records the result in an *observation table.* We access an entry in the observation table by calling the function $T : S \times E = \{\text{accepted}, \text{rejected}\}$. When an equivalence query is answered affirmatively, the algorithm terminates, and the observation table is used to generate a corresponding DFA.

2.4 Voting Machine

Sturton et al [7] present a voting machien design as an FST implemented in the hardware description language Verilog. This state machine representation is semantically equivalently to the Verilog implementation.

The voting machine consists of the following components, or modules:

- Map - converts user's touch to a corresponding button press
- Controller - determines which contest is currently active
- Selection State - controls the candidates selected in each contest
- Cast - commits the final values of the selection state of each contest to memory
- Display - generates the output screen

The formal specification of the voting machine consists of:

Algorithm 1 Angluin's Algorithm

$S = \{\lambda\}$, $E = \{\lambda\}$
Fill in table by asking membership queries for λ and $a \in \Sigma$
while correct DFA not found **do**
  while not closed or not consistent **do**
    if $\exists s \in S, a \in \Sigma$ such that $\forall s' \in S, T(s \cdot a) \neq T(s')$ **then**
      Add $s \cdot a$ to S
      Fill in table by asking membership queries over $(S \cup (S \cdot A)) \cdot E$
    end if
    if $\exists s_1, s_2 \in S$ such that $T(s_1 \cdot e) = T(s_2 \cdot e), \forall e \in E$ and $\exists a \in \Sigma, e \in E$ such that $T(s_i \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$ **then**
      Add $a \cdot e$ to E
      Fill in table by asking membership queries over $(S \cup (S \cdot A)) \cdot E$
    end if
  end while
  Ask equivalence query
  if given counterexample t **then**
    Ad t and prefixes of t to S
    Fill in table by asking membership queries over $(S \cup (S \cdot A)) \cdot E$
  else
    Correct DFA found
  end if
end while

1. Formal specification of each component
2. Behavioral properties of entire machine
3. Structural properties of entire machine
4. Formal model of human expectations of a voting machine's operation, called the *specification voting machine*

The specification voting machine is defined as an FST where the set of input events correspond to a user's button press, and the set of outputs include changes in the current contest and candidate selection states. Formally, the input set I includes input events $b \in I$ corresponding to the navigation buttons *next*, *previous*, and *cast*, as well as the candidate selection buttons $\{candidate_0 ... candidate_{k-1}\}$ where k is the maximum number of candidates in each contest. The output set O includes output events of the form (i, s_i) where i represents the active contest and s_i is the *selection state*, or set of candidates currently selected in contest i. We let $i \in \{1...n\}$ where n is the number of contests in a given election. Additionally, $s_i \subseteq \{1...k\}$ and $|s_i| \leq l$, where l is the maximum number of candidates that can be selected in each contest.

A possible trace of the specification voting machine is:

$$next, (2, \{\}), (candidate_1), (2, \{1\})$$

In this project, we work with a simplified version of the voting machine. The map and display modules have been abstracted away, leaving the controller, selection state, and cast modules. The machine consists of only two contests, each of which have at most two candidates. Only one candidate can be selected per contest. Thus, we have $n = 2$, $k = 2$, and $l = 1$.

3 LTL Synthesis

3.1 Lily

Jobstmann and Bloem [3] have designed a command-line tool Lily, written in Perl, that implements an automata-theoretic approach to LTL synthesis. We use Lily to synthesize the cast, controller, and selection state modules of the voting machine. Generally, LTL synthesis is impractical, as the algorithm is

2EXPTIME-complete. If we restrict the specification to a particular subset of LTL, we can reduce the complexity to polynomial time. This subset of LTL consists of formulas of rank 1 generalized reactivity (GR(1)). Lily does not restrict the specification to GR(1) formulas. Instead, in order to speed the execution of the algorithm, Lily performs a highly optimized version of Kupferman and Vardi's LTL synthesis algorithm [4]. Given a *realizable* LTL specification, Lily generates both a Verilog and state machine implementation of a design that fulfills the specification. A specification is realizable if a corresponding implementation exists.

Lily works by treating the synthesis problem as a game between the environment and system. The purpose of the tool is to synthesize a program with a set of inputs I and outputs O. The environment controls the input signals in I, and the system controls the output signals in O. The strategy the system follows is to determine a correct output value for each input sequence. The environment is hostile, in that it tries to force the system to violate the specification. If this occurs, the system loses the game, which indicates that the specification is unrealizable. The system wins when it fulfills the specification over $I \cup O$. In this case, a correct implementation of the specification has been identified, and the synthesis terminates successfully.

3.2 LTL Specifications

The challenge of using Lily is to provide an accurate and complete LTL specification for each module, one that results in a correct design and cannot be satisfied trivially. As mentioned in section 2.1, the formal voting machine specification given by Sturton et al [7] provides a LTL specification for each module, as well as behavioral properties that apply to the entire machine. Before these specifications could be given to Lily as input, two modifications were made. First, we added *fairness* properties, a type of liveness property that put necessary constraints on the environment. Second, all inputs and outputs were converted to Boolean values, as Lily can only process Boolean signals. For the number of properties in each specification provided as input to Lily, see Table 2.

3.3 Results

The LTL specifications written by Sturton et al [7] proved to be sufficient for synthesis of a simplified voting machine. For each of the controller, selection

Module	Inputs		Outputs		LTL Properties	
Controller	7	3	4	3	14	8
Selection State	7	3	2	1	25	7
Cast	5	3	4	2	25	12

Table 2: LTL Specifications (second column gives values if Lily accepted non-Boolean signals)

Module	Synthesis Time (sec)	Lines of Verilog	States in FSM
Controller	130.85	90	7
Selection State	23.12	60	4
Cast	540.68	340	16

Table 3: Synthesized Designs

state, and cast modules, Lily synthesized a correct Verilog implementation and corresponding FSM. We have run these experiments, as well as those described in the rest of the paper, on a Mac OS 10.5 machine with a 2.4 GHz Intel Core 2 Duo processor and 2 GB of RAM. Our results are summarized in Table 3.

Once Lily has synthesized each module, we can link the Verilog implementations together to produce the final design. The multiplexor between the controller and selection state modules was written by hand.

We test each module, as well as the complete voting machine, using the Verilog simulation tool Icarus Verilog. Each individual component of the machine is correct-by-design. However, because the synthesis was component-based, the global behavioral properties given by Sturton et al for the original machine remain to be verified [7]. We used Cadence SMV [5], a symbolic model checking tool, to verify the three behavioral properties.

We now review an alternate method for synthesis from specification. This synthesis approach, discussed in the next section, uses scenarios instead of LTL properties.

4 Synthesis from Scenarios

4.1 L* Implementation

For our synthesis problem, it was necessary to implement a modified version of the L* algorithm. We wrote this implementation of the algorithm as an executable Python script. Our implemenation differs from L* in two ways. First, our implementation generates a FST instead of a DFA. Therefore, instead of asking membership queries, the algorithm asks *output queries*. Whereas membership queries determine whether a given input string is accepted, output queries determine the corresponding output string for a given input string.

Second, our implementation works offline instead of online. A set of example input/output string pairs are provided to the algorithm initially, and output queries are answered by searching this set. If the input string is not found in the set of examples, the query returns null. Equivalence queries are answered by iterating through the set of examples and checking whether the candidate FST produces the correct output for each. If one fails, that input string is used as the counterexample.

4.2 Scenarios

Before the above implementation of L* could be used, a set of scenarios have to be provided. Here we used scenarios satisfying a set of coverage criteria defined by Sturton et al [7]. If we have shown that our design satisfies certain structural properties, we can generate a test suite T that satisfies these coverage criteria. As proven by Sturton et al[7], T is sufficient for verification of the design. The significance of this result is that T includes only a polynomial number of tests. Specifically, the size of T is $O(n \cdot k)$ where n is the number of contests and k is the number of candidates in each. The coverage criteria are described below:

1. Initial state coverage: $\exists t \in T$ such that t does not transition out of the initial state of the design.

2. Transition coverage: For every transition $\delta((i, s_i), b)$ in the design, $\exists t \in T$ such that t includes this transition.

3. Output screen coverage: For every output state (i, s_i), $\exists t \in T$ such that t ends at this state.

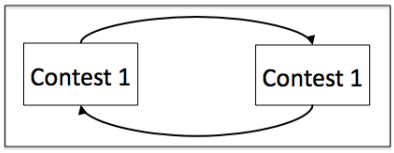


Figure 2: Voting Machine

We used these criteria to write a set of scenarios for synthesis. The hypothesis we tested was that such a test suite, already proven to be sufficient for verification, would be sufficient for synthesis, as well. Because the complexity of the L* algorithm is polynomial in the number of states of the target state machine, and our machine has $O(n^k)$ states, we aim for synthesis of a *hierarchical state machine*, or HSM. An HSM is a state machine in which the states can be either ordinary states or state machines themselves [1]. Thus, instead of synthesizing the entire "flat" state machine, we synthesize each independent level separately. In order to do this, we must have a box diagram of our target state machine. The voting machine's structural properties guarantee independence between navigatation between contests and candidate selection within contests, so we come up with the structure in Figure 2. Synthesizing the navigation and candidate selection separately reduces the number of states in each target FSM to $O(n)$ and $O(k)$, respectively.

We used the following sets of scenarios (also summarized in Table 4), each of which satisfies the coverage criteria given above:

Navigation:

1. $\epsilon \rightarrow (1, s_i)$
2. $(previous) \rightarrow (1, s_i)$
3. $(next) \rightarrow (2, s_i)$
4. $(next), (next) \rightarrow (2, s_i)$
5. $(next), (previous) \rightarrow (1, s_i)$

Individual Contest:

1. $\epsilon \rightarrow (i, \{\})$

	Inputs	Outputs	Scenarios
Navigation	2	2	5
Candidate Selection	2	3	7

Table 4: Scenarios

2. $(candidate_1) \rightarrow (i, \{1\})$
3. $(candidate_2) \rightarrow (i, \{2\})$
4. $(candidate_1), (candidate_1) \rightarrow (i, \{\})$
5. $(candidate_1), (candidate_2) \rightarrow (i, \{1\})$
6. $(candidate_2), (candidate_2) \rightarrow (i, \{\})$
7. $(candidate_2), (candidate_1) \rightarrow (i, \{2\})$

Note that because the navigation inputs only affect the contest number i in the output tuple (i, s_i), we leave the selection state s_i unspecified in the navigation test suite. Similarly, because the candidate selection inputs only affect the selection state s_i , we leave the contest number i unspecified.

4.3 Results

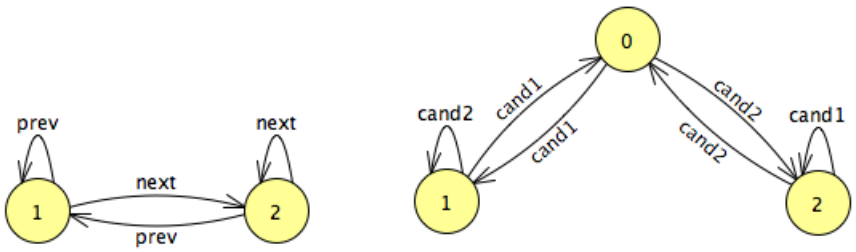
The synthesized navigation and candidate selection state machines are shown in Figure 3. We combine these state machines into an HSM (Figure 4), which can then be flattened into a regular FSM by taking the cross product of the three machines.

5 Comparison

5.1 Effort

5.1.1 Writing Tool

There was no need to write a tool for LTL synthesis, since Lily was built for that purpose. In order to perform synthesis from scenarios, however, it was necessary to write a modified implementation of the L* algorithm. Although not a difficult algorithm to program, this step required a moderate amount



(a) (b)

Figure 3: (a) Navigation and (b) Candidate Selection (using the following shorthand: prev = *previous*, cand1 = $candidate_1$, and cand2 = $candidate_2$)

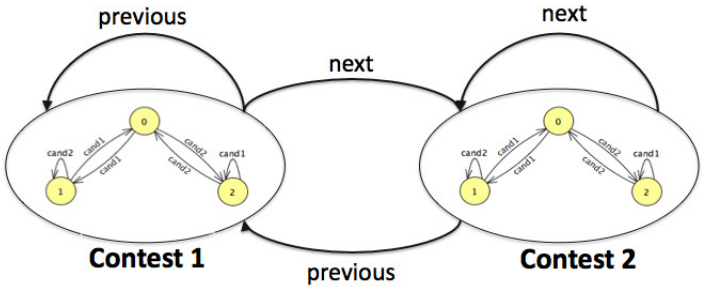


Figure 4: HSM

	Writing Tool	Writing Inputs	Manipulating Results
LTL	None	1 week ; difficult	3-4 days ; difficult
Scenarios	3-4 days ; difficult	$<$ 1 day ; easy	$<$ 1 day ; easy

Table 5: Effort Comparison

Module	Transition Properties	Total Properties
Controller	11	14
Selection State	24	25
Cast	25	25

Table 6: Transition Properties

of time and consideration. A notably challenging step was deciding how best to write an offline implementation of the original online algorithm.

5.1.2 Writing Inputs

Although Lily worked as expected when provided with complete and realizable LTL specifications, writing such specifications was difficult and time-consuming. The main challenge was not being able to tell when the LTL specification included sufficient properties for correct synthesis. This problem arose often because a large number of LTL properties had to be written to define state transitions. If the LTL property is of the form $\mathrm{G}(\varphi \rightarrow \mathrm{X}(\phi))$, we categorize it as a transition property. While writing the specification, it was easy to miss necessary transition properties and assume the specification was finished when it was still incomplete. Table 6 illustrates the high ratio of transition properties to total LTL properties.

Another source of difficulty was coming up with appropriate fairness constraints. For the controller and selection state module specifications, it was often necessary to preface an LTL property π with the fairness constraint $\mathrm{G}(\mathrm{F}(reset = 1)) \rightarrow \pi$. For the cast module specification, $\mathrm{G}(\mathrm{F}(\mathrm{cast} = 1)) \rightarrow \pi$ was used instead. In each case, choosing the appropriate environmental restriction was not straightforward.

These issues demonstrate the limitations of LTL synthesis. If our experience with the voting machine example is indicative of LTL synthesis in general, the method does not scale well. While it is possible to write a correct and realizable specification for a large design, the process is time-consuming and difficult.

On the other hand, writing a set of scenarios to be used as input to L* is relatively easy. The coverage criteria provide a helpful template, and allow us to determine when enough scenarios have been written. Furthermore,

although we have no algorithm for generating this suite, doing so manually is not difficult. Each scenario is an intuitive way to represent a property of the voting machine. For instance, if we want to specify the property that pressing a candidate button twice in a row results in deselection of that candidate, we write the following scenarios:

$$(candidate_1), (i, \{1\}), (candidate_1), (i, \{0\})$$

$$(candidate_2), (i, \{2\}), (candidate_2), (i, \{0\})$$

In comparison, we would write the following LTL formulas to specify the same property:

$$\mathrm{G}(reset \rightarrow \mathrm{X}(\mathrm{G}(ss\_selector * \neg reset * candidate_1 * selection\_state_1 \rightarrow \mathrm{X}(selection\_state_1))))$$

$$\mathrm{G}(reset \rightarrow \mathrm{X}(\mathrm{G}(ss\_selector * \neg reset * candidate_2 * selection\_state_2 \rightarrow \mathrm{X}(selection\_state_2))))$$

In our experience, the scenarios given above are easier to write and understand than the corresponding LTL properties. Of course, whereas an implementation synthesized from the LTL properties is guaranteed to be correct, an implementation synthesized from the scenarios is not. In the voting machine example, a suite satisfying the coverage criteria proved sufficient for correct synthesis, but it remains to be proven whether this result holds generally. We revisit this issue in Section 5.2.2.

One difficulty that applies to writing both LTL and scenario specifications is determining when the specification is complete. Besides attempting synthesis and evaluating the result, there is no way to know whether a specification contains sufficient LTL properties or scenarios. We often fell into a cycle of rewriting the specification, resynthesizing, and repeating until the correct implementation was produced. The desire to reduce iterations of this cycle motivates the incremental synthesis approach, as discussed in Section 6.

5.1.3 Manipulating Results

In both synthesis approaches, providing correct inputs to the corresponding tool or algorithm did not produce an immediately correct result. Lily

	Synthesis Time	Correctness	States in FSM
LTL	10 min	Correct-by-design	448 (upper bound)
Scenarios	< 1 sec	Needs to be verified	18

Table 7: Results Comparison

was used to synthesize three Verilog modules, which then had to be wired together to produce a final design. This process was done by manually, by writing a Verilog wrapper module. It was also necessary to implement an additional multiplexor module to connect the controller to the appopriate selection state. Although not a conceptually difficult task, a certain amount of tweaking inputs and outputs was necessary to connect the modules together properly. For instance, the *ss_enable* signal, originally defined as a register, had to be implemented as a wire in order for the design to work. Thus, understanding and working with the Verilog generated by Lily was moderate hurdle in the LTL synthesis process.

Although it was also necessary to combine the FSMs generated by the scenario-based approach, this task was very straightforward. Since the box diagram was a prerequisite for synthesis, the structure of the final HSM was known ahead of time. The two layers synthesized by the L* algorithm fit together exactly as described by this structure. Additionally, flattening the HSM into a regular FSM simply requires taking the cross product of the component machines.

5.2 Results

5.2.1 Synthesis Time

The total time required to synthesize the three voting machine modules using Lily was about ten minutes. In contrast, L* synthesizes the navigation and candidate selection machines in less than a second. This comparison is of limited use, however, because the three modules synthesized by Lily do not correspond to the two state machines synthesized by L*. The former use internal signals of the voting machine, whereas the latter only connect initial

inputs, or button presses, to the corresponding final outputs, or ballots. Thus, we would expect the LTL synthesis to take longer. Not only does Lily need to synthesize an additional module, but it also processes more signals in each (see also Tables 2 and 4).

5.2.2 Correctness

One major benefit of LTL synthesis is that the output is correct-by-design. Since the specification was used to generate the result, the result automatically satisfies the specification. Thus, each module synthesized by Lily is guaranteed to meet its specification. However, the three behavioral properties of the entire machine still remained to be verified. This is a consequence of using component-based synthesis. Thus, we used Cadence SMV to verify these properties, as mentioned in Section 3.3. The command-line implementation of this tool verified the three global properties in less than a second.

When synthesizing from scenarios, there is no guarantee that the resulting design will meet its specification. Thus, formal verification is still necessary. In the future, perhaps a set of criteria will be proven to be accurately describe a test suite sufficent for correct synthesis. If we can prove that, in the general case, a suite satisfying the coverage criteria always produces the correct result, then the formal verification step could be eliminated. At this point, however, no such proof exists.

5.2.3 Size

Neither Lily nor L* is guaranteed to produce the minimum state machine. Since Lily was used to synthesize individual components, rather than an entire machine, there is no way to accurately determine the size of the synthesized voting machine. We can obtain an upper bound by taking the product of the sizes of each component module. The resulting bound is 448 states, which is significantly larger than 18, the number of states in the flattened HSM synthesized using L*. However, the usefulness of this comparison is debatable. A number of the 448 states may be equivalent, and therefore collapsable into a single state. Additonally, some of these states may be unreachable. Thus, had Lily been used to synthesize the entire machine, rather than each component, the number of states in the resulting machine might have been much less than 448.

6 Combined Approach

We are currently exploring a synthesis approach that combines synthesis from LTL with synthesis from scenarios. The idea is to synthesize a design from a partial LTL specification, and then refine the design using examples. There are many reasons to pursue such an approach. The main motivation is that certain properties of a design are easier to express in LTL formula, whereas others are easier to describe using examples. A hybrid approach allows us to use both LTL and examples, without writing a full set of either, to express the specification most naturally. This allows us a great deal more flexibility when describing our design. As long as the LTL specification is realizable, it does not matter if the specification is complete or correct. For instance, an additional safety constraint or missing transition property can be incorporated into the design after the initial synthesis by using examples. As such, this hybrid approach facilitates incremental synthesis. Instead of resynthesizing the entire design when the specification changes, we simply refine the implementation with examples that reflect the change.

An overview of the method is as follows. We provide as input a partial LTL specification S and a set of scenarios T, which reflect properties of the design not covered by S. We use S to generate a candidate FSM, M, which is then provided as input to the oracle of L*. The equivalence query is answered by testing M against each $t \epsilon T$. If M fails to produce the correct output for one of the scenarios, that scenario will be used as the counterexample. The L* algorithm refines M until the machine, now called M', produces the correct output for each scenario.

Two directions for this work are:

- Prove that M' still satisfies S
- Write an algorithm for generating T

7 Conclusion

We have demonstrated that, assuming knowledge of the structural properties of the voting machine, we can synthesize an implementation both from formal specification and from scenarios. In the former case, the LTL specifications given by Sturton et al [7] are sufficient for synthesis. In the latter case, a test

suite that satisfies the coverage criteria given by Sturton et al [7] is sufficient for synthesis.

This work highlights many directions for further research. Although we have discussed one example of using Lily for LTL synthesis, it remains to be seen how the tool performs on other designs. Our experience suggests that the tool may not be practical for large or complex designs, given the difficulty of writing accurate and complete LTL specifications. More work could be done to determine whether such experience is typical of LTL synthesis using Lily, or particular to our example.

Similarly, more work could be done to determine whether the results of our scenario-based synthesis apply to the general case. We have demonstrated that, given a box diagram of the voting machine's structure, a test suite satisfying the coverage criteria is sufficient for synthesis. We do not know if this result holds for all designs. Furthermore, we have not yet identified a method for generating such a test suite automatically. Thus, more work could be done to identify a heuristic for generating sufficient, yet minimum, scenarios, as well as to determine the complexity of HSM synthesis in the general case.

Perhaps the most interesting direction of research is combining LTL specifications with scenarios. This idea is briefly summarized in the previous section. This approach would address the limitations of both approaches, and provide the freedom to write a specification using LTL formulas as well as concrete examples. Thus, the work presented here identifies the strengths and weaknesses of two synthesis approaches, and lays the groundwork for a third, more promising, strategy.

References

[1] Rajeev Alur. *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, chapter Formal analysis of hierarchical state machines, pages 434–435. Springer Berlin, 2004.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75:87–106, 1987.

[3] Barbara Jobstmann and Roderick Bloem. Optimizations for ltl synthesis. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided*

Design, pages 117–124, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Orna Kupferman and Moshe Y. Vardi. Safraless decision procedures. In *FOCS '05: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, pages 531–542, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Kenneth L. McMillan. Cadence smv. http://www.kenmcmil.com/smv.html.

[6] Amir Pnueli. The temporal logic of programs. In *FOCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[7] Cynthia Sturton, Susmit Jha, Sanjit A. Seshia, and David Wagner. On voting machine design for verification and testability. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 463–476, New York, NY, USA, 2009. ACM.