Synthesis for Regular Specifications over Unbounded Domains

Jad Hamza[*], Barbara Jobstmann[†], Viktor Kuncak[‡]
[*]ENS Cachan, France [†]CNRS/Verimag, France, [‡]EPFL, Switzerland

***Abstract*—Synthesis from declarative specifications is an ambitious automated method for obtaining systems that are correct by construction. Previous work includes synthesis of reactive finite-state systems from linear temporal logic and its fragments. Further recent work focuses on a different application area by doing functional synthesis over unbounded domains, using a modified Presburger arithmetic quantifier elimination algorithm. We present new algorithms for functional synthesis over unbounded domains based on automata-theoretic methods, with advantages in the expressive power and in the efficiency of synthesized code.**

Our approach synthesizes functions that meet given regular specifications defined over unbounded sequences of input and output bits. Thanks to the translation from weak monadic second-order logic to automata, this approach supports full Presburger arithmetic as well as bitwise operations on arbitrary length integers. The presence of quantifiers enables finding solutions that optimize a given criterion. Unlike synthesis of reactive systems, our notion of realizability allows functions that require examining the entire input to compute the output. Regardless of the complexity of the specification, our algorithm synthesizes linear-time functions that read the input and directly produce the output. We also describe a technique to synthesize functions with bounded lookahead when possible, which is appropriate for streaming implementations. We implemented our synthesis algorithm and show that it synthesizes efficient functions on a number of examples.

I. Introduction

Automated synthesis of systems from specifications is a promising method to increase development productivity. Automata-based methods have been the core technique for reactive synthesis of finite-state systems [1], [2], [3]. In this paper, we show that automata-based techniques can also be used to perform functional synthesis over unbounded data domains. In functional synthesis, we are interested in synthesizing functions that accept a tuple of input values (ranging over possibly unbounded domains), and generate a tuple of output values that satisfy a given specification. Our efforts are inspired in part by advances in software synthesis for bit-manipulating programs [4]. Our goal is to develop and analyze complete algorithms that require only a declarative specification as input. Recently, researchers have proposed [5] a technique for functional synthesis based on quantifier elimination of Presburger arithmetic.

In the previous approach, the functions generated by quantifier elimination can be inefficient if the input contains inequalities, possibly performing search over a very large space of integer tuples. Furthermore, this approach handles disjunctions by a transformation into disjunctive normal form. Finally, the specification language accepts integer arithmetic but not bitwise constructs on integers.

In this paper we present a synthesis procedure that is guaranteed to produce an efficient function that computes a solution of a given constraint on unbounded integers in time linear in the combined length of input and the shortest output, represented in binary. Moreover, our specification language supports not only Presburger arithmetic operations, but also bitwise operations and quantifiers. We achieve this expressive power by representing integers as sets in weak monadic second-order logic of one successor (WS1S) which is known to be more expressive than pure Presburger arithmetic [6], [7]. We use an off-the-shelf procedure, MONA [8], to obtain a deterministic automaton that represents a given WS1S specification.

As our central result, we show how to convert an arbitrary automaton recognizing the input/output relation into a function that reads the input sequence and produces an output sequence that satisfies the input/output relation. Consequently, we obtain functions that are guaranteed to run in linear-time on arbitrarily large integers represented as bit sequences. Assuming constant-time lookup of automaton transition, the running time of the synthesized functions is independent of the automaton size. These properties are a consequence of our algorithm, and we have also experimentally verified them on a number of examples. Our result solves the problem of synthesis of general WS1S specifications that are not necessarily causal. Our basic algorithm generates implementations that have $O(N)$ time and space complexity, where N is the number of bits of input and output. We show how to reduce space consumption to $O(\log N)$ if the time is increased to $O(N \log N)$.

We also examine synthesis for sub-classes of WS1S specifications that can be implemented using bounded memory. We introduce a class of implementations based on a finite union of asynchronous transducers, and show that they can be used to implement k-causal specifications as well as specifications in Presburger arithmetic without bitwise operations.

II. Examples

A. Parity Bit Computation

The goal of our first example is to illustrate the form of the functions produced by our synthesizer. For a non-negative integer x, let $x[k]$ denote the k-th least significant bit in the

This research was facilitated by the COST Action IC0901 *Rich Model Toolkit—An Infrastructure for Reliable Computer Systems* and the Dagstuhl Seminar on Software Synthesis, December 2009. The author list has been sorted according to the alphabetical order.

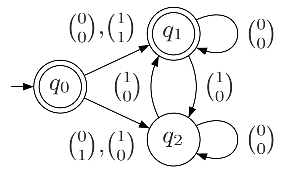
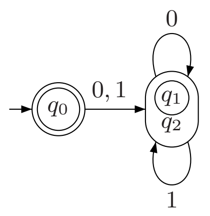


Fig. 1. Automaton A for parity specification between x and y

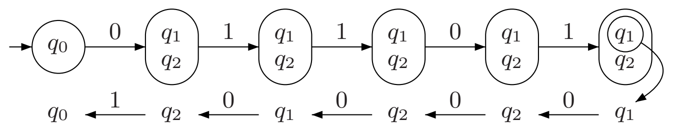
```
x: 0 1 1 0 1
y: 1 0 0 0 0
```

Fig. 2. Input x and output y satisfying parity specification



Transition	State	τ
$\{q_0\}\xrightarrow{0}\{q_1,q_2\}$	q_1	$(q_0,0)$
$\{q_0\}\xrightarrow{0}\{q_1,q_2\}$	q_2	$(q_0,1)$
$\{q_0\}\xrightarrow{1}\{q_1,q_2\}$	q_1	$(q_0,1)$
$\{q_0\}\xrightarrow{1}\{q_1,q_2\}$	q_2	$(q_0,0)$
$\{q_1,q_2\}\xrightarrow{0}\{q_1,q_2\}$	q_1	$(q_1,0)$
$\{q_1,q_2\}\xrightarrow{0}\{q_1,q_2\}$	q_2	$(q_2,0)$
$\{q_1,q_2\}\xrightarrow{1}\{q_1,q_2\}$	q_1	$(q_2,0)$
$\{q_1,q_2\}\xrightarrow{1}\{q_1,q_2\}$	q_2	$(q_1,0)$

Fig. 3. Automaton A' for computing parity y of input x



q_0 0 q_1 q_2 1 q_1 q_2 1 q_1 q_2 0 q_1 q_2 1 q_1 q_2
q_0 1 q_2 0 q_1 0 q_2 0 q_2 0 q_1

Fig. 4. Running synthesized function on input shown in Fig. 2

binary representation of x. (We write the binary digits starting with the least significant one on the left, so $\overline{11001}_2$ is a binary representation of 19.) Our first specification states that the first output bit, $y[0]$ indicates the parity of the number of one-bits in the input (Figure 2): $y[0] = |\{k \mid x[k] = 1\}|\%2$.

Consequently, the synthesized function must examine the entire input before emitting the first bit of the output.

One way to specify this computation is as follows. Let n_{max} have the property $\forall k > n_{max}. x[k] = 0$. To specify y, introduce first an auxiliary sequence of bits z such that

$$z[n] = |\{k \leq n \mid x[k] = 1\}|\%2$$

for all $n \leq n_{max}$, by defining $z[k+1]$ as xor of $z[k]$ and $x[k+1]$. Then define $y[0]$ to be $z[n_{max}]$.

Figure 1 shows the generated automaton A for this specification, accepting the words $\binom{x[0]}{y[0]}\binom{x[1]}{y[1]}\ldots\binom{x[n]}{y[n]}$ which satisfy the given relation between x and y. After applying our construction to compute a function from x to y, we obtain the input-determinstic automaton A' shown on the left of Figure 3, augmented with two labeling functions τ and ϕ. The automaton is the result of first projecting out the part of A' labels corresponding to the output, then applying the subset construction. Therefore, the labels in A' correspond to input bits, and the states are sets of states of the automaton A. Function τ tells us how to move backwards within a run of A' to construct an accepting run of the underlying automaton A; it thus recovers information lost in applying the projection to A. Finally, function ϕ tells us for every accepting state in A' at which state of A to start the backward reconstruction. The table on the right of Figure 3 shows τ for A': it maps every transition $S \xrightarrow{\sigma_i} S'$ of A' and every state $q' \in S'$ into a predecessor state $q \in S$, and a matching output value σ_o, such that $(q, (\sigma_i \cup \sigma_o), q')$ is a transition in the automaton A. We indicate function ϕ in A' by additional circles around individual states, e.g., $\phi(\{q_1, q_2\}) = q_1$. Figure 4 shows the run of A' on the input 01101. The synthesized function first runs the deterministic automaton A' (the upper part of Figure 4, ending in state $\{q_1, q_2\}$). The synthesized function then picks a state q according to ϕ (the state q_1 in case of our example), and runs backwards according to τ while computing the output bits. The lower part of Figure 4 shows the backward computation following τ defined in Figure 3; the backward run generates the bits 10000 of the output.

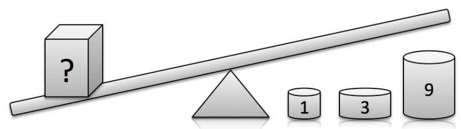


Fig. 5. Beam balance with three weights

B. Synthesizing Specialized Constraint Solvers

Our next examples illustrate a range of problems to which our synthesis technique applies. Consider first the beam balance (scale) depicted in Figure 5. We are interested in a function that tells us, for any object on the left-side of the beam, how to arrange the weights to balance the beam. We are given three weights, with 1, 3, and 9kg, respectively. We use the variable w for the weight of the unknown object. For each available weight i, we use two variables l_i to indicate whether the weight is placed on the left side and r_i to indicate it is placed on the right side of the beam. We obtain the constraint:

$$w + l_1 + 3l_3 + 9l_9 = r_1 + 3r_3 + 9r_9. \tag{1}$$

Because each weight can only be use at most once, we require that the solution also respects the following three constraints

$$l_1 + r_1 \leq 1,\ l_3 + r_3 \leq 1,\ l_9 + r_9 \leq 1. \tag{2}$$

When we give these four constraints to our tool, it compiles them into a function. The function accepts arbitrary input values and returns corresponding output values, performing computation in time linear in the number of bits in the input. E.g., if the object weights 11kg, then the program tells us that we should use Weight 1 on the left and Weight 3 and 9 on the right side to balance the beam. It is easy to verify that this response is correct by insertion into Equation 1 leading to $11 + 1 \cdot 1 = 3 \cdot 1 + 9 \cdot 1$. When asked for $w = 15$, the program correctly responds with “There is no output for your input.”

C. Modifying Example to Minimize Output

Next, we consider a modified version of the balance example to show that neither inputs nor outputs need to be bounded. It also shows how to specify a function that minimizes the output. In the previous example, we could only balance objects up to 13kg because only one copy of each weight was available. Assume we want to balance arbitrary heavy objects with the minimal number of balance weights of 1, 3, and 9kg. We keep the constraint from Eqn. (1) and replace the constraints in Eqn. (2) by a constraint that asks for a minimal solution:

$$\forall l'_1, l'_3, l'_9, r'_1, r'_3, r'_9. \quad \mathsf{balance}(w, l'_1, l'_3, l'_9, r'_1, r'_3, r'_9) \rightarrow \\ \mathsf{sum}(l_1, l_3, l_9, r_1, r_3, r_9) \leq \mathsf{sum}(l'_1, l'_3, l'_9, r'_1, r'_3, r'_9)$$

where $\mathsf{balance}(w, l'_1, l'_3, l'_9, r'_1, r'_3, r'_9)$ is the constraint obtained from Eqn. 1 by replacing l_i and r_i by l'_i and r'_i, respectively, and sum refers to the sum of the listed variables. This constraint requires that every other solution that would also balance the scale for the given object has to use more weights than the solution returned.

The newly synthesized program gives correct answers for arbitrary large natural numbers. E.g., let us assume the object weighs 1234512345123451234512345123456789kg, then the program tells us to take 1371680383470501371680384088 times Weight 9 on the right side and once Weight 3 on the left side.

D. Finding Approximate Solutions

Consider the constraint $6x+9y = z$, where z is the input and x, y are inputs. The solution exists only when z is a multiple of 3, so we may wish to find x, y that minimizes $|6x + 9y - z|$, using a similar encoding with quantifiers as in the previous example. The support for disjunctions allows us to encode the absolute value operator that is useful for finding approximate solutions. The tool synthesizes a function that given a value of z, computes x, y to be as close to z as possible. For example, given the input 104, the tool outputs $x = 13$ and $y = 3$.

E. Folding and Inverting Computations

Consider the Syracuse algorithm function, whose one step is given by $f(x) =$ if $(2 \mid x)$ then $x/2$ else $3x + 1$. Consider a relation on integers corresponding to iterating f six times: $r(x, y) \leftrightarrow f^6(x) = y$. (We could use such function to speed-up experimental verification of the famous $3n + 1$ conjecture that states $\forall x > 0. \exists n. f^n(x) = 1$.) When we use $r(x, y)$ as the specification and indicate x as input and y as output, our synthesizer generates a function that accepts a sequence of bits of x and outputs in linear time a sequence of bits of y that is given by 6-fold iteration of f. Note that, if the synthesis from a specification (e.g. $y = f^n(x)$) succeeds, the runtime of the computation is independent of n and is linear in the number of bits of x. Therefore, our approach can effectively fold n iterations of f into one linear-time function on the binary representations of inputs and outputs.

F. Processing Sequences of Bits

We next illustrate the use of specification of unbounded numbers in simple signal processing task. Suppose we have an input signal X with discrete values in the range $\{0, 1, 2, \ldots, 15\}$ and we wish to compute a smoothed output signal Y by averaging signal values with its neighbors, using the formula $Y_i = (X_{i-1} + 2X_i + X_{i+1}) \operatorname{div} 4$. We specify this function in WS1S as a relation between unbounded integers x and y, where we reserve 4 bits for value of the signal at each time point (see Figure 6). For constants a, b, let $x[k+a, k+b]$

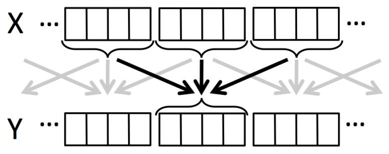


Fig. 6. Averaging signal values

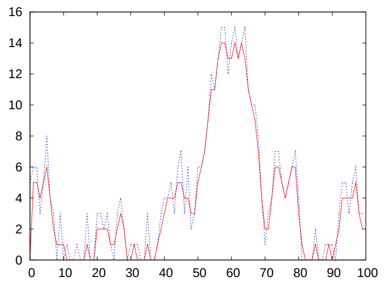


Fig. 7. The result of applying the synthesized function that computes a smoothed version of a signal. The function on an arbitrarily long signal was specified in WS1S.

denote the number represented by the subrange of digits of x between $k + a$ and $k + b$:

$$x[k+a, k+b] = x[k+a] + 2x[k+a+1] + \ldots + 2^{b-a}x[k+b]$$

We define the smoothing relation between numbers x and y by:

$$\forall i. \quad (4|i) \rightarrow y[i+4..i+7] = \\ (x[i..i+3] + 2x[i+4..i+7] + x[i+8..i+11]) \operatorname{div} 4$$

Our synthesizer generates a function that, given the sequence of bits x, produces a sequence of bits y. Figure 7 shows an input signal (dotted line) and the resulting smoothed signal (full line) that results after we applied the linear-time function synthesized by our tool to the input.

III. Preliminaries

A. Words and Automata

Given a finite set of variables V, we use Σ_V to denote the alphabet $\Sigma_V = 2^V$. We omit V in Σ_V if it is clear from the context. When used as a letter, we denote $\emptyset \in \Sigma_V$ by $\mathbf{0}$. Given a finite word $w \in \Sigma_V^*$, we use $|w|$ to denote the length of w, and w_i to denote the letter on the i-th position of w. By ε we denote the empty word, of length zero. Given a partitioning of V into the sets I and O and a letter $\sigma \in \Sigma_V$, we use $\sigma|_I$ to denote the projection of σ to I, i.e., $\sigma|_I = \sigma \cap I$. We extend projection in usual sense to words and languages.

A *finite automaton* A over a finite set of variables V is a tuple $(\Sigma, Q, \mathsf{init}, F, T)$, where $\Sigma = 2^V$ is the *alphabet*, Q is a finite set of *states*, $\mathsf{init} \in Q$ is the *initial state*, $T \subseteq Q \times \Sigma \times Q$ is the *transition relation*, and $F \subseteq Q$ is a set of *final states*. Automaton A is *deterministic*, if for all transitions $(q_1, \sigma_1, q'_1), (q_2, \sigma_2, q'_2) \in T$, $q_1 = q_2$ and $\sigma_1 = \sigma_2$ implies $q'_1 = q'_2$ holds. A is *complete*, if for all states $q \in Q$ and letters $\sigma \in \Sigma$, there exists a state $q' \in Q$ such that $(q, \sigma, q') \in T$. Note that if A is deterministic and complete T describes a total function from Q and Σ to Q.

$$\begin{array}{rcl}
F & ::= & F \wedge F \mid F \vee F \mid \neg F \mid t_N < t_N \mid t_N = t_N \\
 & \mid & t_N[t_P] \mid t_P < t_P \mid t_P = t_P \mid (C \,|\, t_N) \mid t_N \underset{t_P}{\sim} t_N \\
 & \mid & \forall_{\mathsf{pos}} k.F \mid \exists_{\mathsf{pos}} k.F \mid \forall x.F \mid \exists x.F \\
t_N & ::= & x \mid C \mid t_N + t_N \mid C \cdot t_N \mid t_N \,\mathsf{div}\, C \mid t_N \,\%\, C \\
 & \mid & (t_N \veebar t_N) \mid (t_N \barwedge t_N) \mid t_N \ll C \mid t_N \gg C \\
 & \mid & 2^{t_P} \mid t_N[t_P..^{+}C] \mid t_N[0..t_P] \\
t_P & ::= & k \mid C \mid k + C \mid k \dot{-} C \mid \mathsf{maxBit}(t_N) \\
C & ::= & \text{non-negative integer constant}
\end{array}$$

Fig. 8. Syntax of WS1S where sets denote natural numbers (T_N) and elements denote positions (T_P) in binary representations of numbers

Given an automaton $A = (\Sigma, Q, \text{init}, F, T)$ and a state $q \in Q$, we use A_q to refer to the automaton (Σ, Q, q, F, T) that has the same structure as A but starts at q.

A *run* ρ of A on a word $w \in \Sigma^*$ is a sequence of states $q_1 \dots q_{|w|+1}$ such that (i) $q_1 = \text{init}$ and (ii) for all $1 \le i \le |w| : (q_i, w_i, q_{i+1}) \in T$. A run is *accepting* if $q_{|w|+1} \in F$. We say w *is accepted by* A if there exists a run of A on w that is accepting. We denote by $\mathcal{L}(A) \subseteq \Sigma^*$ the set of words accepted by A.

The *exhaustive run* ρ of A on a word $w \in \Sigma^*$ is a sequence of sets of states $S_1 \dots S_{|w|+1}$ such that (i) $S_1 = \{\text{init}\}$ and (ii) for all $1 \le i \le |w|$, $S_{i+1} = \{q' \in Q \mid \exists\, q \in S_i, (q, w_i, q') \in T\}$. An exhaustive run is *accepting* if $S_{|w|+1} \cap F \ne \emptyset$. Note that if A is deterministic, then the run of A on a word w is unique and the elements in the exhaustive run of A on w are singletons.

Lemma 1: For an automaton A with a set of states Q, computing an exhaustive run of A for a word $w \in \Sigma^*$ can be done in time $O(|Q| \cdot |w|)$ for a non-deterministic A, and can be done in time $O(|w|)$ for a deterministic A.

Given an automaton $A = (\Sigma_V, Q, \text{init}, F, T)$ over variables V and a set $I \subset V$, the *projection of* A *to* I, denoted by $A|_I$, is the automaton $(\Sigma_I, Q, \text{init}, F, T_I)$ with $T_I = \{(q, \sigma_I, q') \in Q \times \Sigma_I \times Q \mid \exists \sigma \in \Sigma_V, (q, \sigma, q') \in T \wedge \sigma|_I = \sigma_I\}$. In the remainder, we fix I to be the set of input and O to be the set of output variables.

B. WS1S as extension of Presburger Arithmetic

Figure 8 shows the syntax of weak monadic second-order logic of one successor, which we use as our specification language for unbounded non-negative integers. The logic contains all integer linear arithmetic operations and quantifiers, thus subsuming Presburger arithmetic. Furthermore, it contains the expression $x[k]$ to extract the k-th least significant bit of the number x. It is also possible to find a c-successor of position k, with notation $k + c$, as well as the c-predecessor, with notation $k \dot{-} c$, denoting the position $\max(k - c, 0)$. Together with quantification over positions, this allows the specification of arbitrary uniform bitwise relations on integer variables. To illustrate the expressive power of WS1S, we introduce shorthands for some of the constraints that can be defined in this way: bitwise operations ($\barwedge$, $\veebar$), left and right shifting ($\ll$, $\gg$), a sub-word of length c at position k of a given integer x (denoted $x[k..^{+}c]$), congruence modulo 2^p (denoted $x \sim_p y$), the initial prefix of an integer $x[0..k]$, the integer 2^p for a position p, and the smallest p such that $x < 2^p$, denoted $\mathsf{maxBit}(x)$.

C. Amortized Cost of Synthesis

We describe the cost of synthesis and synthesized program in a unified framework, by considering the entire amortized cost of applying a given specification a on a series of inputs $b_1, \dots, b_n$. Let f be a function with two arguments, so that $f(a, b) = c$ if the input-output pair (b, c) satisfies the specification a. We implement function f using a function of the form $g(a, b, s) = (f(a, b), s')$ that computes f and updates its local state from s to s'. We assume a fixed initial state s_0. The presence of local state can make the computation more efficient on a series of inputs. This framework accounts for simple cases such as memoization and caching, as well as the more general case of on-the-fly specialization.

Given the specification a and the inputs $b_1, \dots, b_n$ we define $s_i = g(a, b_i, s_{i-1})$ for $i \in \{1, \dots, n\}$. Let $g'(a, b, s)$ denote the time to compute $g(a, b, s)$. Let $|x|$ denote the length of value x. We define the amortized cost of g on inputs $a; b_1, \dots, b_n$ by $\frac{1}{n} \sum_{i=1}^{n} g'(a, b_i, s_{i-1})$. Our main complexity measure is then $c(s_a, s_b, n)$, which we define as the maximum amortized cost over all $a; b_1, \dots, b_n$ for which $|a| \le s_a$ and $|b_i| \le s_b$ for all i.

Observe that $c(s_a, s_b, 1)$ is simply the complexity of running function f once on inputs of size s_a and s_b, respectively. Another useful measure, of particular interest in synthesis, is $c_\infty(s_a, s_b) = \lim_{n \to \infty} c(s_a, s_b, n)$, which amortizes any pre-computation that happens in finitely many steps. We next present several examples to illustrate the cost function $c_\infty(s_a, s_b)$ for implementations of several problems.

Example 1 (Finding an enclosing interval): Consider the problem of computing the smallest interval enclosing a given number. More precisely, the goal is to compute $f([x_1, \dots, x_m], y) = (L, U)$ where $L = \max\{x_i \mid x_i \le y\}$ and $U = \min\{x_j \mid y \le x_j\}$ given an unordered list of numbers $x_1, \dots, x_m$ (with the result arbitrary if the $\max$ or $\min$ expressions above are not defined). In this example, we assume that each number takes constant space to represent, so $|[x_1, \dots, x_m]| = m$ and $|y| = 1$. An algorithm for one invocation can simply make a single pass through the list, computing the current $\max$ of lower bounds of y and the current $\min$ of the upper bounds up to a given position in the list. This gives the worst-case complexity m of the algorithm. If we use this algorithm as the implementation g (without making use of state), we obtain $c_\infty(m, 1)$ of $O(m)$.

Consider next an alternative implementation, given by $g'([x_1, \dots, x_m], y, s)$, which behaves as follows: on the first invocation, $g([x_1, \dots, x_m], y, s_0)$, builds a balanced binary search tree storing the set of numbers $x_1, \dots, x_m$ in time $O(m \log m)$, and returns this tree in the resulting state s'. On subsequent invocations, g uses this tree to find the enclosing interval (L, U), which can be done in time $O(\log m)$ by doing a lookup in the tree. Therefore, we obtain that n invocations require $O(m \log m + n \log m)$, which gives $c(m, 1, n) \in O(\frac{1}{n}(m \log m) + \log m)$ and $c_\infty(m, 1) = O(\log m)$. Thus, we have seen that precomputation improves the amortized time $c_\infty(m, 1)$ from $O(m)$ to $O(\log m)$. $\square$

IV. Synthesis Algorithm

A. *Constructing Specification Automaton*

The input to our algorithm is a WS1S formula G whose free variables $z_1, \ldots, z_r$ denote unbounded integers. We assume a partitioning of the index set $\{1, \ldots, r\}$ into inputs I and the outputs O. In the first step, our algorithm constructs a deterministic specification automaton A accepting words in the alphabet $\Sigma_{I \cup O}$. We use a standard automaton construction [9] and obtain an automaton A characterizing the satisfying assignments of G, i.e. whose language $\mathcal{L}(A)$ contains precisely the words $\sigma_0 \sigma_1 \ldots \sigma_n \in \Sigma^*_{I \cup O}$ for which G holds in the variable assignment $(z_1, \ldots, z_r)$ in which the k-th least significant bit of z_i is one iff $0 \leq k \leq n$ and $i \in \sigma_k$. We use $\mathcal{L}(G)$ to denote the language over $\Sigma_{I \cup O}$ characterizing the satisfying assignments of G. From this correctness property it follows that $w \in \mathcal{L}(A)$ implies $w\mathbf{0}^p \in \mathcal{L}(A)$ for every $p \geq 0$.

B. *Overview*

All subsequent steps of our algorithm work with the specification automaton A and do not depend on how this automaton was obtained. Given A, our goal is to construct a function that computes, for a given sequence of inputs bits a corresponding sequence of output bits such that the combined word is accepted by the deterministic automaton.

Note that we seek an implementation that works uniformly for *arbitrarily long sequences of bits*, which means that it is not possible to pre-compute all possible input/output pairs.

We show our construction in several steps. First, we assume that we are only interested in outputs whose length does not exceed the length of inputs. For this case we start by describing a less time-efficient implementation (Subsection IV-C) that depends on the size of A, then describe an efficient version, showing that we can avoid the dependence on the size of A (Subsection IV-D). Finally, we show how to lift the assumption that the outputs are no longer than the inputs (Subsection IV-E).

C. *Input-Bounded Synthesis of Unspecialized Implementations*

In the first version of our solution we assume that, given an input bit sequence, we seek an output sequence of the *same length* such that the input and output pair are accepted by the specification automaton A.

Our unspecialized implementation P_{unspec} simulates the given automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$ on the input word $w \in \Sigma^*_I$ and tries to find an accepting run. P_{unspec} first constructs the exhaustive run $\rho = S_1 \ldots S_{|w|+1}$ of the projected automaton $A|_I$ on w (see preliminaries for the definition of automaton projection and exhaustive run). If ρ is not accepting, then there is no matching output word and P_{unspec} terminates. Otherwise, P_{unspec} picks a state $q_{|w|+1}$ in $S_{|w|+1} \cap F$ and constructs an accepting run $q_1 \ldots q_{|w|+1}$ of A and the output word v by proceeding backwards over i, from $i = |w|$ to $i = 1$, as follows: it picks $v_i \in \Sigma_O$ and $q_i \in S_i$ such that $(q_i, w_i \cup v_i, q_{i+1}) \in T$. When it reaches one of the initial states in S_1, the result is an accepting run of the automaton A; the desired output is the sequence $v_1 \ldots v_{|w|}$ of the output components of the labels in the reconstructed run.

The P_{unspec} implementation repeats the above construction for each input word w. From Lemma 1 we obtain the amortized cost of P_{unspec}.

Lemma 2: If s_A denotes the size of the input automaton A and s_w denotes the size of the input word, then the unspecialized implementation P_{unspec} solves the synthesis for input-bounded specifications in amortized time $c(s_A, s_w, n)$ of $O(s_A \cdot s_w)$ (consequently, $c_\infty(s_A, s_w)$ is also $O(s_A \cdot s_w)$).

D. *Input-Bounded Synthesis of Specialized Implementations*

We next present our main construction (illustrated in the Example II-A), which avoids the dependence of the running time of computation of on the (potentially large) number of states of the automaton A. To obtain an implementation with optimal runtime, we transform the given automaton A into an input-deterministic automaton A' using the subset construction on the projection $A|_I$. The challenge is to extend the subset construction with the additional labeling functions that allow us to efficiently reconstruct an accepting run of A from an accepting run of A'. Given such additional information, our specialized implementation P_{spec} runs A' on the input w and uses the labeling to construct the output v.

Our construction introduces two labeling functions, ϕ and τ. The function ϕ maps each accepting state S of A' into one state $q \in S$ that is accepting in A. The τ function indicates how to move backwards through the accepting run; it maps each transition (S, σ_i, S') of A' and a state $q' \in S'$ into a pair $(q, \sigma_o) \in S \times \Sigma_o$ of new a state and an output letter, such that $(q, \sigma_i \cup \sigma_o, q')$ is a transition of the original automaton A.

Definition of synthesized data structure A', ϕ, τ. Given an automaton $A = (\Sigma_{I \cup O}, Q, \text{init}, F, T)$, we construct an automaton $A' = (\Sigma_I, Q', \text{init}', F', T')$ and two labeling functions $\phi : F' \to Q$ and $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ such that (i) A' is deterministic, (ii) $\mathcal{L}(A)|_I = \mathcal{L}(A')$, and (iii) for every word $u \in \mathcal{L}(A')$ with an accepting run $S_1 \ldots S_{n+1}$ of A', there exists a word $w \in \mathcal{L}(A)$ with $w|_I = u$ and an accepting run $q_1 \ldots q_{n+1}$ of A such that $\phi(S_{n+1}) = q_{n+1}$ and for all $1 \leq i \leq n$, $(q_i, w_i|_O) \in \tau((S_i, u_i, S_{i+1}), q_{i+1})$. We define A' as follows:

$$\begin{array}{lcl}
Q' & = & 2^Q \\
\text{init}' & = & \{\text{init}\} \\
F' & = & \{S \in Q' \mid S \cap F \neq \emptyset\} \\
T' & = & \{(S, i, S') \in Q' \times \Sigma_I \times Q' \mid \\
 & & \quad S' = \{q' \mid \exists q, \sigma.(q, \sigma, q') \in T \wedge q \in S \wedge \sigma|_I = i\}\}
\end{array}$$

We define $\phi : F' \to Q$ such that if $S \in F'$ then $\phi(S) \in S \cap F$; such value exists by definition of F'.

We define $\tau : (T' \times Q) \to (Q \times \Sigma_O)$ for $(S, i, S') \in T'$ and $q' \in S'$ as follows. By definition of T', there exists a transition

$(q, \sigma, q') \in T$ of the original automaton such that $\sigma|_I = i$. We pick an arbitrary such transition and define $\tau((S, i, S'), q') = (q, \sigma|_O)$.

Computing A' and τ through automata transformations. In our implementation, we represent both A' and τ in one automaton, which we compute using the following sequence of automata transformations. Because τ refers to sets of transitions, we first turn each transition of A into a state, i.e, given $A = (\Sigma_{I\cup O}, Q, \text{init}, F, T)$, we construct an automaton $B = (\Sigma_{I\cup O}, Q_B, \text{init}_B, F_B, T_B)$ such that

$$\begin{array}{lcl} \text{init}_B & = & (q, \sigma, \text{init}_A) \quad \text{for arbitrarily chosen } q, \sigma \\ Q_B & = & \{\text{init}_B\} \cup T \\ F_B & = & \{(q, \sigma, q') \in Q_B \mid q' \in F\} \\ T_B & = & \{(t, \sigma, t') \in Q_B \times \Sigma_{I\cup O} \times Q_B \mid \\ & & \qquad \exists q, q', q'' \in Q.\ \exists \sigma' \in \Sigma_{I\cup O}. \\ & & \qquad t = (q, \sigma', q') \text{ and } t' = (q, \sigma, q'')\}. \end{array}$$

Next, we project B to I, i.e., we replace every transition (q, σ, q') in B by (q, σ_I, q'). Finally, we obtain automaton C by determinizing $B|_I$ using the classical subset construction. Now, every reachable state in C (other than init_B) corresponds to a transition in A'. Assume we are given a state q_c in C, then q_c has the form $\{(q_1, \sigma_1, q'_1), \ldots, (q_k, \sigma_k, q'_k)\}$ with $\forall i, j, \sigma_i|_I = \sigma_j|_I = \sigma_I$ and corresponds to the transition (t, σ_I, t') in A', where $t = \{q_i \mid 1 \le i \le k\}$ and $t' = \{q'_i \mid 1 \le i \le k\}$. So, every state q_c in C defines a labeling function for the corresponding transition (t, σ_I, t') that maps every state $q'_i \in t'$ to a set of available pairs $(q_i, \sigma_i|_O)$. Our final labeling function τ picks for each state q_i one of the available pairs.

Specialized implementation and its complexity. The specialized implementation P_{spec} runs A' on the input word w and constructs a run $\rho = S_1 \ldots S_{|w|+1}$. If ρ is not accepting, then there is no matching output word and the function terminates. Otherwise, it computes an accepting run $q_1 \ldots q_{|w|+1}$ of A and the output word v as follows: $\phi(S_{|w|+1}) = q_{|w|+1}$ and, for all $1 \le i \le |w|$, $(q_i, v_i) = \tau((S_i, w_i, S_{i+1}), q_{i+1})$.

The following theorem states the correctness of P_{spec} and follows by construction.

Theorem 1: Consider an automaton A and an input $w_1 \ldots w_n$. Then if there exists an output $v_1 \ldots v_n$ such that $(w_1 \cup v_1) \ldots (w_n \cup v_n)$ is accepted by A, then P_{spec} computes one such output $v_1 \ldots v_n$. If there is no corresponding output then P_{spec} indicates that there is no output.

The following theorem states that our construction achieves the desired linear-time behavior and independence from the size of the initial automaton. The construction of A', ϕ, τ takes time singly exponential in the size of the automaton, but is done only once, so it is amortized for each invocation of the automaton. Extracting the output for a given input takes time independent of the number of states in A' because A' and τ have deterministic transitions.

Theorem 2: If s_A denotes the size of the specification automaton A and s_w denotes the size of the input word, then P_{spec} solves the synthesis for input-bounded specifications in amortized time $c(s_A, s_w, n)$ of $O(\frac{1}{n} 2^{s_A} + s_w)$. Consequently, the amortized time $c_\infty(s_A, s_w)$ as the number of queries approaches infinity is $O(s_w)$.

E. Extending Synthesis to Arbitrary Regular Specifications

In this section we extend the result of the previous section to allow computing an output that satisfies the specification even if the output has a larger number of bits than the input. Consider the simple specification $x < y$, where x is the input and y is the output. Given the input $\overline{111}_2$ of length three (representing the number 7), every value of output satisfying the specification has the length at least four.

To adapt the solution in the previous section to the full synthesis problem we generalize the notion of acceptance to take into account any number of zeros that could be appended to the input without changing the meaning of the input. Therefore, if the automaton A' finishes reading the input word and none of the states reached in the last step are accepting, it checks whether one of the states can reach an accepting state while reading only the input letter $\mathbf{0}$. The closure with the input $\mathbf{0}$ can be computed in polynomial time by computing the states that are backward-reachable from an accepting state using only edges with input label $\mathbf{0}$.

To be able to emit the appropriate segment of the output, the backward-reachability computation keeps, for every state, an output word that leads to an accepting state. We use the function $\psi : Q \to \Sigma_O^* \cup \{\bot\}$ to store these words, where Q are the states of the specification automaton A. We write $\psi(q) = \bot$ to denote that there is no input word $w \in \mathbf{0}^*$ that is accepted starting from q. Formally, given the automaton $A = (\Sigma_{I\cup O}, Q, \text{init}, F, T)$, we set $\psi = \psi^{|Q|}$ and define ψ^i inductively: for all $q \in Q$:

(i) $\psi^0(q) = \begin{cases} \varepsilon & \text{if } q \in F \\ \bot & \text{otherwise} \end{cases}$

(ii) let R^i be the set of states q for which $\psi^i(q) \neq \bot$,

$$\psi^{i+1}(q) = \begin{cases} \psi^i(q) & \text{if } q \in R^i \\ \sigma|_O \psi^i(q') & \text{elsif } \exists (q, \sigma, q') \in T : \sigma|_I = \mathbf{0} \wedge q' \in R^i, \\ \bot & \text{otherwise.} \end{cases}$$

Observe that if $\psi(q) \neq \bot$ then $\psi(q)$ is a word of length bounded by the number of states of the specification automaton A. Therefore, the maximal amount by which the output is longer than the input is bounded by the size of the specification automaton.

To recognize leading zeros, we adapt the final states F' of A' (computed as for P_{spec} in the previous section) and extend the labeling function ϕ as follows. Let $\mathsf{fin}(S) = \{q \in S \mid \psi(q) \neq \bot\}$ be the states in S that can reach input on zeros.

$$\begin{array}{l} F' \quad = \{S \in Q' \mid \mathsf{fin}(S) \neq \emptyset\} \\ \phi(S) = q \in \mathsf{fin}(S) \text{ s.t. } |\psi(q)| = \min\{|\psi(q') \mid q' \in \mathsf{fin}(S)|\} \end{array}$$

Note that the function $\phi(S)$ chooses one of the states that lead to an accepting state with an output word of minimal length.

The implementation and its time complexity. Given an input word $w_1 \ldots w_n$, the implementation P_{gspec} generates, as P_{spec} in the previous Subsection (IV-D), a run $S_1, \ldots, S_{n+1}$.

TABLE I
SYNTHESIS TIMES, SIZE OF THE GENERATED AUTOMATA, AND AVERAGE RUNNING TIMES OF SYNTHESIZE FUNCTIONS

No	*Example*	*MONA (ms)*	*Synthesis (ms)*	$\lvert A\rvert$	$\lvert A'\rvert$	512b	1024b	2048b	4096b
1	addition	318	132	4	9	509	995	1967	3978
2	approx	719	670	27	35	470	932	1821	3641
3	company	8’291	1’306	58	177	608	1312	2391	4930
4	parity	346	108	4	5	336	670	1310	2572
5	mod-6-test	341	242	23	27	460	917	1765	3567
6	3-weights-min	26’963	640	22	13	438	875	1688	3391
7	4-weights	2’707	1’537	55	19	458	903	1781	3605
8	smooth-4bits	51’578	1’950	1781	955	637	1271	2505	4942
9	smooth-f-2bits	569	331	73	67	531	989	1990	3905
10	smooth-b-2bits	569	1’241	73	342	169	347	628	1304
11	forward-6-3n+1	834	1’007	233	79	556	953	1882	4022

If $S_{n+1} \notin F'$, then there is no corresponding output; the implementation indicates this and stops. Otherwise, suppose $S_{n+1} \in F'$ and $q_{n+1} = \phi(S_{n+1})$. The implementation generates the backward run from q_{n+1} as in Subsection (IV-D), producing the output bits $v_1 \dots v_n$. The final output is then the word $v_1 \dots v_n \psi(q_{n+1})$.

Theorem 3: Let s_A denote the size of the specification automaton A, s_w the size of the input word, and s_v the size of the shortest output word that satisfies the input word. Then $s_v \leq s_w + s_A$. The P_{gspec} implementation solves the synthesis for arbitrary regular specifications in amortized time $c(s_A, s_v, s_w, n)$ of $O(\frac{1}{n}2^{s_A} + s_w + s_v)$. Consequently, the amortized time $c_\infty(s_A, s_w)$ is $O(s_w + s_v)$.

Together with the correctness of the construction of specification automata from WS1S, we obtain the following soundness and completeness theorem.

Theorem 4: Consider a WS1S formula F with input variables $(z_k)_{k\in I}$ and output variables $(z_k)_{k\in O}$. Consider a binary representation of input variables $(z_k)_{k\in I}$. If there exist values of output variables $(z_k)_{k\in O}$ such that F holds for $(z_k)_{k\in I\cup O}$, then P_{gspec} outputs a sequence of bits for one such $(z_k)_{k\in O}$. If there is no corresponding output then P_{gspec} indicates that there is no output.

V. EXPERIMENTAL RESULTS

We implemented our algorithm in Scala [10] using MONA [8] to construct the specification automaton. Our tool accepts (1) a specification written in WS1S and (ii) the list of input variables. Our tool first invokes MONA to construct the specification automaton, then applies the construction described in Section IV. The construction covers the general case of WS1S specifications. The synthesized function can be invoked at a later point in the program any number of times. The synthesized function computes the output by running a deterministic automaton (Section IV-D) on the input, first forward, then backward. It also computes the additional information necessary to produce outputs that are longer than the inputs (Section IV-E).

We have tested our tool on several specifications including simple addition and the examples described in Section II and the appendix. The specifications were written in WS1S. In Table I, we summarize the results. In the second column, we give a short description that relates the results to the description of the example in Section II. In the column labeled *MONA* we show the time (in ms) that MONA needs to create the specification automaton A. The number of states of A is shown in Column labeled $|A|$. The column *Synthesis* shows the time (in milliseconds) necessary to create the synthesized function. The column $|A'|$ shows the number of states of the projected and augmented automaton A'. The last four columns show the time (in milliseconds) to run the synthesized functions on 1000 random inputs of different bit length (512, 1024, 2048, and 4096 bits). All tests were performed on an Intel Core 2 Duo P7450 (2,13 GHz) CPU with 4096 MB DDR2 (667 MHz) RAM.

In most cases (9 out of 11) creating the augmented automaton (column *Synthesis*) is faster than creating the initial automaton (column *MONA*). In some cases (e.g., Line 6 and 8) the synthesis time is only a fraction of the overall time. The running time of the synthesized function is (as expected) linear in the number of bits of the input.

We are not aware of any other tool that can handle the examples that we present in Table I. The closest tool that we are aware of is Comfusy [5], which does not support quantifiers and large integers for implementation reasons, and does not support bitwise operations due to a fundamental restriction of the underlying quantifier elimination algorithm. Conversely, there are examples supported by Comfusy (multiplication with symbolic constants and set constraints) that our approach does not support, so a future combination of approaches based on automata and quantifier elimination would be fruitful.

VI. SPACE COMPLEXITY FOR SYNTHESIZED SYSTEMS

So far we have mostly focused on the time that the synthesized systems. This section examines the space complexity. Let $N = |w|$ denote length of the input.

Logspace upper bound for general WS1S. The algorithm we described so far (and implemented, as discussed in Section V) produces systems that run in $O(N)$ time and use $O(N)$ space to store the results of the forward automaton run. We next sketch how to obtain an implementation that has $O(N\log(N))$ time and only $O(\log^2(N))$ space. We assume the ability to randomly access any arbitrary letter of the input string. The idea then is to avoid storing all states $q_0, \ldots, q_N$ of the forward run, and instead compute them on demand, storing only a sparse subsequence $q_{i_0}, q_{i_1}, \ldots, q_{i_m}$ where $m = \lceil \log N \rceil$. Let p denote the current position in the backward run of the synthesized function. The synthesized function maintains the invariant $0 = i_0 \leq i_1 \leq \ldots \leq i_m \leq p$. Initially it sets $i_k \approx N(1 - 2^{-k})$. To move back from p to $p-1$, it re-runs the forward automaton from the largest i_k for which $i_k < p$, and redistributes $i_{k+1}, \ldots, i_m$, similarly as for the initial run, maintaining the ordering and the decreasing geometric progression of distances $i_{k+j+1} - i_{k+j}$. Because each position pointer i_j requires $O(\log N)$ space and there are $\log N$ of them, this implementation needs $O(\log^2 N)$ space. A run that updates pointers i_{k+i} for $i \geq 0$ re-reads 2^{-k} fraction of the input and is called 2^k times, so the total time is $O(N \log N)$.

Unions of asynchronous transducers. An *(asynchronous) transducer* $M = (A, \lambda, \varphi)$ over input variables I and output variables O consists of (1) a deterministic automaton $A = (\Sigma_I, Q, \mathrm{init}, F, T)$ and (2) two labeling functions $\lambda : T \to \Sigma_O^*$ and $\varphi : F \to \Sigma_O^*$. A (more conventional) *synchronous transducer* is a special case of an asynchronous transducer where $|\lambda(t)| = 1$ for all $t \in T$ and $|\varphi(q)| = \varepsilon$ for all $q \in F$.

The *outcome* of $M = (A, \lambda, \varphi)$ on a valid input word $w \in \mathcal{L}(A)$, denoted by $\mathrm{out}_M(w)$, is the concatenation of output words $u_1, \ldots, u_n$ produced by M while reading w concatenated with the final word produced by φ, i.e., if $\rho = q_1 q_2 \ldots q_{n+1}$ is the accepting run of A on $w \in \mathcal{L}(A)$, then $\mathrm{out}_M(w) = u_1 \ldots u_n u_{n+1}$, where $u_i = \lambda(q_i, w_i, q_{i+1})$ for all $1 \leq i < n$ and $u_{n+1} = \varphi(q_{n+1})$. Note that the outcome of M is only defined for valid input words. The language of M, denoted $\mathcal{L}(M)$ is the is the union of valid input/output pairs padded with trailing zeros to have equal length: $\mathcal{L}(M) = \{w \in \Sigma_{I \cup O}^* \mid \exists j, k.\ w|_I \in \mathcal{L}(A)\mathbf{0}^j \wedge w|_O = \mathrm{out}_M(w|_I)\mathbf{0}^k\}$.

An asynchronous transducer can express even certain specifications that are not WS1S expressible. For example, consider a transducer that emits ε when reading 0 and emits 1 when reading 1. Such transducer outputs a contiguous sequence of output bits whose length is the number of bits in the input.

Given a finite set of transducers $M_1, \ldots, M_k$ with $M_i = (A_i, \lambda_i)$ and a language L over the variables $I \cup O$, we say that $M_1, \ldots, M_k$ *jointly implement* L, written $M_1, \ldots, M_k \models L$ iff (1) each transducers M_i produces outputs satisfying the specification, i.e., $\mathcal{L}(M_i) \subseteq \mathcal{L}(G)$ and (2) the union of M_i's covers the valid inputs, i.e., $\mathcal{L}(G)|_I \subseteq \bigcup_i \mathcal{L}(A_i)$. We say $M_1, \ldots, M_k$ implements a WS1S formula G, denoted $M_1, \ldots, M_k \models G$, iff $M_1, \ldots, M_k \models \mathcal{L}(G)$.

Note that if $M_1, \ldots, M_k \models G$, then there exists a finite-memory implementation for G that performs only two passes over the input (regardless of k). In the first pass, the implementation generates no output, but simply determines which of the transducers accept. In the second pass, the implementation generates the output for one of the transducers that accept.

Transducers for Presburger specifications. The following lemma can be shown by analyzing the output of the quantifier-elimination based synthesis algorithm for Presburger arithmetic specifications [5]. They key observation is that functions implementing Presburger specifications have the form $\bigvee_i (P_i(x) \wedge y = t_i(x))$ for input x and output y.

Lemma 3: For every WS1S specification G that encodes a formula in Presburger arithmetic, there exists a finite set of transducers $M_1, \ldots, M_k$ such that $M_1, \ldots, M_k \models G$.
The key observation is that witness terms $t(x)$ are computable using asynchronous transducers.

Note that Presburger specifications are not computable using only one asynchronous transducer due to presence of disjunctions in specifications. They are also not computable using a finite union of *synchronous* transducers because of the division by constants.

Limitations of asynchronous transducers.

Lemma 4: There exists WS1S specifications cannot be implemented using a finite union of asynchronous transducers.

The proof is based on considering the following WS1S specification G over input I and output O. We give G as regular expression over the binary presentation over I and O:

$$G = \begin{matrix} I: \\ O: \end{matrix} \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}^+ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)^* \begin{pmatrix} 1 \\ 1 \end{pmatrix}^+ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}^* .$$

Observation 1: Every transducer $M = (A, \lambda, \varphi)$ with $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ and less than n states that accepts an input word $(1^n 0)^k$ must output a non-empty word within every n steps while reading this input.

Observation 2: Every transducer $M = (A, \lambda, \varphi)$ with $\mathcal{L}(M) \subseteq \mathcal{L}(G)$ and less than n states that accepts the input word $(1^n 0)^k$ for some $k > 0$, rejects all input words $(1^n 0)^l$ for $l > k$.

Using the above observations and given k asynchronous transducers $M_1, \ldots, M_k$ with $M_i = (A_i, \lambda_i, \varphi_i)$ such that $\mathcal{L}(M_i) \subseteq \mathcal{L}(G)$ it suffices to consider words $(1^n 0)^i$ for $i = 1, \ldots, k+1$ to conclude that it cannot be the case that $\mathcal{L}(G)|_I = \bigcup_{i=1,\ldots,k} \mathcal{L}(A_i)$.

Note that G can be implemented by a finite set of transducers if the input is read from right to left. However, we can concatenate specifications such as G with their reversed versions to obtain specifications that cannot be realized by transducers making both forward and backward passes.

VII. Lookahead-Causal Specifications

An interesting class of specifications that can be implemented using a single asynchronous transducer are lookahead-causal specifications discussed in this section.

The algorithms presented so far first read the entire input and then generate a corresponding output. In some cases (e.g., in streaming applications), one might prefer an implementation that starts outputting before reading the entire input. Specifications such as the signal processing example require reading a bounded number of bits ahead (three, in this case) to compute an output bit.

For notational simplicity we consider specifications $\mathsf{spec}(x, y)$ containing a single input-output pair x and y. Furthermore, we assume that the specifications are total, that is, $\forall x. \exists y. \mathsf{spec}(x, y)$. If a specification is not total, we can transform it into a total specification $\mathsf{spec}'(x, y, e)$ given by $(\mathsf{spec}(x, y) \wedge e = 0) \vee ((\neg\exists y.\mathsf{spec}(x, y)) \wedge y = 0 \wedge e = 1)$.

Definition of k-causality. We next define lookahead-k-causality, or k-causality for short. We say that an input output pair x, y is k-causal for spec, written $\mathsf{causal}_k(x, y)$ iff $\forall p.\ \forall x' \sim_{p+k} x.\ \exists y' \sim_p y.\ \mathsf{spec}(x', y')$. where $z' \sim_p z$ means that z' and z have identical the initial p bits. spec is *k-causal* iff it implies $\mathsf{causal}_k(x, y)$ for all x, y.

Observe that a k-causal specification can be implemented by an asynchronous transducer, but there are specifications (such as the sign function) implementable by asynchronous transducers that are not k-causal. If spec is not k-causal but some inputs have multiple possible outputs, a general strategy to turn spec it into a causal specification is to simply conjoin it with $\mathsf{causal}_k(x, y)$ and check whether the resulting specification is still total, that is, whether $\forall x.\exists y.\mathsf{spec}(x, y) \wedge \mathsf{causal}_k(x, y)$.

Synthesized system for a k-causal specifications. Let $\mathsf{spec}(x, y)$ be a k-causal and total specification. We show how to construct an implementations that emits the input after reading k steps of the output. Construct first the specification automaton A and apply the construction described in Section IV-E. We obtain the automaton A' and the labeling functions τ, ϕ, and ψ. We extend A', τ, ϕ, and ψ so that they include, for all states q of A, the determinized version A'_q of A_q, where A_q is the automaton that differs from A only in that its initial state is changed to q. The synthesized program P_{caus} for k-causal specification has a fill parameter $\mu > 0$. It uses a buffer of length at least $(1 + \mu)k$ and alternates operations Read and Flush. The Read operation reads one more input bit into the buffer and advances the state S of A' accordingly, as for P_{gspec}. The Flush operation is invoked when the input buffer contains at least j input bits for $j \geq \lceil (1 + \mu)k \rceil$. It runs backwards k steps from the current state $q = \phi(S)$ following τ and reaches state q'. It then treats q' as a final state of the entire input, emits $j - k$ outputs going backwards and reaching state q''. It then empties the corresponding buffer elements and moves forward from q' using $A_{q'}$ until the current position of the input. This gives a streaming implementation that traverses the input bits only $1/\mu$ more times compared to P_{gspec}, regardless of k.

VIII. Related Work

Like synthesis of combinational circuits from relations (e.g.,[11]) our work synthesizes a function implementing the given relation. However, our implementation works for arbitrarily long input sequences. Techniques [1], [2], [12] to synthesize reactive systems that implement a given S1S specification can handle arbitrarily long input sequences. They assume that the specification can be implemented by a (usually finite-state) system that produces the output immediately while reading the input, i.e., the system cannot look ahead. These techniques usually take specifications in a fragment of temporal logic [13] and have resulted in tools that can synthesize useful hardware components [14], [3]. Recent work [15] establishes theoretical results (without implementation) regarding the problem of deciding when an S1S specification can be implemented using a system with lookahead. The (k-bounded) causality checks in our problem could be performed using this decision procedure based on infinite game theory. Our specification language uses finite instead of infinite words, which allows us to eliminate the non-causal behaviors and thus simplify the synthesis process. Moreover, our technique is not restricted to k-bounded specifications.

The work on graph types [16] proposes to synthesize fields given by definitions in monadic second-order logic and also uses the MONA tool [8]. However, it focuses on computing assignments to update fields of linked data structures as opposed to numerical and bit constraints.

IX. Conclusion

We presented an algorithm to synthesize linear-time functions from general WS1S specifications. Our software implementation works on a number of interesting examples. We have also identified interesting classes of specifications that can be implemented using finite unions of asynchronous transducers, and provided examples of specifications for which such finite-memory implementations do not suffice. Our results therefore contribute to the understanding and to the algorithm toolbox of automated synthesis approaches for software and hardware.

Acknowledgements. We thank Nir Piterman for discussion of the synthesis algorithm for general specifications, and Roderick Bloem for inspiring questions about the necessity of unbounded memory.

References

[1] J. R. Büchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies," *Trans. of the American Math. Society*, 1969.

[2] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*, ser. Regional Conference Series in Mathematics, 1972.

[3] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *FMCAD*, 2006.

[4] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *ACM PLDI*, 2005.

[5] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, "Complete functional synthesis," in *ACM PLDI*, 2010.

[6] W. Thomas, "Languages, automata, and logic," in *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.

[7] T. Schüle and K. Schneider, "Verification of data paths using unbounded integers: Automata strike back," in *Haifa Verification Conference*, 2006.

[8] N. Klarlund, A. Møller, and M. I. Schwartzbach, "MONA implementation secrets," in *CIAA*. LNCS, 2000.

[9] J. R. Büchi, "Weak second-order arithmetic and finite automata," *Z. Math. Logik Grundl. Math.*, vol. 6, pp. 66–92, 1960.

[10] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: a comprehensive step-by-step guide*. Artima Press, 2008.

[11] J. H. Kukula and T. R. Shiple, "Building circuits from relations," in *CAV*, 2000, pp. 113–123.

[12] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL '89*. New York, NY, USA: ACM, 1989, pp. 179–190.

[13] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, 2006.

[14] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *CAV*, 2007.

[15] M. Holtmann, L. Kaiser, and W. Thomas, "Degrees of lookahead in regular infinite games," in *FOSSACS*, 2010, pp. 252–266.

[16] N. Klarlund and M. I. Schwartzbach, "Graph types," in *ACM POPL*, 1993.