 Reactive Synthesis Modulo Theories using Abstraction Refinement

Benedikt Maderbacher
Graz University of Technology
Graz, Austria
benedikt.maderbacher@iaik.tugraz.at

Roderick Bloem
Graz University of Technology
Graz, Austria
roderick.bloem@iaik.tugraz.at

Abstract—Reactive synthesis builds a system from a specification given as a temporal logic formula. Traditionally, reactive synthesis is defined for systems with Boolean input and output variables. Recently, new techniques have been proposed to extend reactive synthesis to data domains, which are required for more sophisticated programs. In particular, Temporal stream logic (TSL) extends LTL with state variables, updates, and uninterpreted functions and was created for use in synthesis. We present a new synthesis procedure for TSL(T), an extension of TSL with theories. Our approach is also able to find predicates, not present in the specification, that are required to synthesize some programs. Synthesis is performed using two nested counter-example guided synthesis loops and an LTL synthesis procedure. Our method translates TSL(T) specifications to LTL and extracts a system if synthesis is successful. Otherwise, it analyzes the counterstrategy for inconsistencies with the theory, these are then ruled out by adding temporal assumptions, and the next iteration of the loop is started. If no inconsistencies are found the outer refinement loop tries to identify new predicates and reruns the inner loop. A system can be extracted if the LTL synthesis returns realizable at any point, if no more predicates can be added the problem is unrealizable. The general synthesis problem for TSL is known to be undecidable. We identify a new decidable fragment and demonstrate that our method can successfully synthesize or show unrealizability of several non-Boolean examples.

I. INTRODUCTION

Reactive synthesis [1] is the problem of automatically constructing a system from a specification. The user provides a specification in temporal logic and the synthesis procedure constructs a system that satisfies it if one exists. Traditionally this only works for systems with Boolean input and output variables. However, real-world systems often use more sophisticated data like integers, reals, or structured data. For finite domains, it is possible to use bit-blasting to obtain an equivalent Boolean specification. However, in general, bit-blasting techniques do not work for infinite domains, bit-blasted specifications are hard to read, and a large number of variables make the specifications very hard to solve.

In recent years multiple theories have been proposed to perform reactive synthesis with non-Boolean inputs and outputs. There have been decidability results for synthesis using register automata [2], [3], [4] and variable automata [5].

Our work builds on temporal stream logic (TSL). TSL, proposed by Finkbeiner et al. [6], uses a logic based on linear temporal logic (LTL) with state variables, uninterpreted functions and predicates, and update expressions. TSL allows for an elegant and efficient synthesis method that separates control from data. However, the ability to specify how data is handled is limited because functions and predicates remain uninterpreted. Finkbeiner et al. [7] describe an extension to TSL modulo theories, but consider only satisfiability and not synthesis.

In this paper, we propose a new synthesis algorithm for temporal stream logic modulo theories that can be applied to arbitrary decidable theories in which quantifier elimination is possible. Let us consider a concrete example using the theory of linear integer arithmetic (LIA).

Example 1. We want to build a system with one integer state variable $x$ and one integer input $i$. The objective is to keep the value of the state variable between 0 and 100. At any time step the system can select one of two updates: increase or decrease $x$ by $i$, where $i$ is chosen by the environment in the interval $0 \leq i < 5$. We assume that the initial state is any value inside the boundaries. These requirements can be written as the TSL formula

$$
\phi \triangleq (0 \leq x \land x < 100 \land \square(0 \leq i \land i < 5)) \rightarrow \\
\square(0 \leq x \land x < 100 \land ([x \leftarrow x - i] \lor [x \leftarrow x + i])),
$$

where the propositions $[x \leftarrow x - i]$ and $[x \leftarrow x + i]$ describe updates to $x$. Figure 1 shows a mealy machine that realizes the specification above. It is impossible to write a correct system using only the predicates from the specification: if the environment chooses initially $x = 0$ the system has to first perform addition, however, if the environment chooses $x = 99$ the system has to perform subtraction. The predicates in the specification cannot distinguish these cases.

Inspired by this example we want our synthesis algorithm to function with expressions from theories, as well as identify new predicates where necessary. Figure 2 shows an overview of our approach. We use a different refinement loop than the...
original TSL synthesis approach [6], our approach, which is depicted in Figure 2, relies on checking local properties and also is able to create new predicates.

First, the TSL specification is encoded into an LTL formula that contains a Boolean variable for each theory predicate in the TSL formula. These variables are seen as inputs, which means that the environment determines their truth values. As a result, realizability of the LTL formula implies realizability of the TSL formula, but not vice versa, because the environment can choose values for the variables that are not consistent with the theory. The LTL formula is then given to a propositional LTL synthesis tool [8], [9]. If the Boolean synthesis is successful we obtain a Boolean system that can be concretized into a system that operates on the original value domain. If synthesis of the LTL formula is not successful, we get a Boolean counterstrategy that we analyze for inconsistencies with respect to the theory.

The central part of our algorithm is the theory consistency analysis of the counterstrategy. In contrast to Finkbeiner et al. [6] we treat the counterstrategy as a Moore machine instead of a tree. The output of a state in a counterstrategy is a valuation of the predicates and the transitions perform updates on the register values. Whereas the original approach analyzes potentially long traces in the tree, we perform a local analysis of individual states and transitions. We use an SMT solver to check whether the predicate valuation in each state is consistent and whether the valuations in two consecutive states are consistent with the updates on the transition between them. We show that if all states and transitions are (locally) consistent, then the counterstrategy is globally theory-consistent as well and thus the TSL specification is unrealizable.

If an inconsistency is found, the counterstrategy is spurious. We thus need to refine the LTL formula and start a new iteration. We refine the LTL specification by adding new assumptions and possibly new predicates. The assumptions refer to the relation between predicates (for inconsistent outputs of states) or between predicates and updates (for inconsistent transitions). If a transition is consistent for some values but inconsistent for others, we create a new predicate that distinguishes whether the update is valid or not.

The procedure can be likened to a CEGAR loop [10] or to the DPLL(T) in which LTL synthesis plays the role of the propositional SAT solver and the consistency check is performed by the theory solver. The main difference is that in our case inconsistencies can span multiple time steps. Our approach has multiple advantages over the original TSL synthesis approach: it can easily be extended to new theories, it can find new predicate needed to realize certain specifications, and it can show unrealizability (without bound on the size of the considered systems).

The main contributions of our paper are as follows:

- A new synthesis procedure for TSL that works with theories and can generate additional predicates that are necessary to realize certain specifications.
- Synthesis for TSL, in general, is known to be undecidable [6], we show that the problem is decidable for equality logic.
- Our algorithm can prove that certain specifications are unrealizable.

The remaining paper is structured as follows: Section II summarizes required definitions from TSL. Section III formalizes the synthesis problem for TSL modulo theory. We describe the Boolean abstraction and the theory consistency analysis in detail in Section IV. The main synthesis procedure is described in Section V. An experimental evaluation was performed for multiple examples using the theories of linear integer arithmetic and linear real arithmetic (Section VI). We discuss related work in Section VII and conclude our work in Section VIII.

II. Preliminaries

We use Temporal Stream Logic (TSL) [6], with the addition of decidable theories [7]. This section repeats the definitions from Finkbeiner et al. [6], [7] with some changes in notation and with a more general treatment of theories.

A. Theories and Updates

In contrast to [7], where axiomatic semantics of the used theories is used, we rely on the definitions built into an SMT solver. A theory \( T \) consists of a signature (symbols for constants, functions, and predicates) and semantics as defined by the SMT solver (which can be axiomatic). The domain is called \( \mathbb{T} \). In case of variables of different sorts (i.e. types) we use \( \mathbb{T} \) for the union of all domains and assume that all variables take values from their domain. In the following, we will use \( E_T(V) \) to denote the set of expressions in \( \mathbb{T} \) with the set of variables \( V \) denoting a superset of the free variables in the expression. The set \( E_T(V) \) is partitioned into a set \( E^T_T(V) \) of terms (denoting values in \( \mathbb{T} \)) and a set \( E^\mathbb{B}_T(V) \) of formulas (denoting truth values). We assume that the theories used have decidable procedures for satisfiability checking and quantifier elimination. We assume that we are given a procedure \( \text{sat} \) that returns true iff a formula \( \phi \) is satisfiable and a function \( \text{quantelim} \) that takes a formula and returns a theory-equivalent formula that does not contain quantifiers.
B. Temporal Stream Logic Modulo Theories

TSL(T) is based on linear temporal logic, but instead of Boolean variables it uses updates and Boolean theory expressions. Key concepts of TSL and TSL(T) are state variables \( R \) and input variables \( I \) which both hold values from the theory domain and updates that access state variables and input variables and write new values to the state variables. The grammar for TSL(T) formulas is

\[
\begin{align*}
\langle \text{ap} \rangle & := E^F_\mathcal{P} (R \cup I) \\
\langle \text{term} \rangle & := E^F_\mathcal{P} (R \cup I) \\
\langle \text{bconst} \rangle & := \text{true} \mid \text{false} \\
\langle \text{upd} \rangle & := [\langle \text{var} \rangle \leftarrow \langle \text{term} \rangle] \\
\langle \text{tsl} \rangle & := (\langle \text{ap} \rangle \mid \langle \text{upd} \rangle \mid \langle \text{bconst} \rangle \mid \neg \langle \text{tsl} \rangle \mid \langle \text{tsl} \rangle \land \langle \text{tsl} \rangle) \\
& \mid \langle \text{tsl} \rangle \cup \langle \text{tsl} \rangle \mid X \langle \text{tsl} \rangle.
\end{align*}
\]

To define the semantics of TSL(T) we need some additional notation that will be used throughout the paper. An update \([r \leftarrow e]\) assigns a state variable \( r \) an expression \( e \in E^F_\mathcal{P} (R \cup I) \). We use \( U \) for the finite set of all updates that occur in the formula under consideration as well as the updates that assign each state variable to itself. An update function \( u \) is a function that associates each state variable \( r \in R \) with an expression \( e \in E^F_\mathcal{P} (R \cup I) \). We refer to the set of update functions where all pairs \((r, e)\) are updates in \( U \) as \( \mathcal{U} \). Equivalently, \( \mathcal{U} \) can be seen as the set of all subsets of \( U \) that contain exactly one update for each state variable in \( R \).

We introduce the notations \( R \triangleq 2^{R \rightarrow \mathcal{T}} \) and \( I \triangleq 2^{I \rightarrow \mathcal{T}} \) for the sets of valuations of variables. We write \( R/r \ (I/i) \) to denote the replacement of all variables in \( R \) (\( I \), resp.) by their corresponding values in \( r \in R \) (\( i \in I \), resp.). With slight abuse of notation, we identify \( e[R/r, I/i] \) with the corresponding value in the domain. To apply an update function \( u \in \mathcal{U} \) to valuations \( r \in R \) and \( i \in I \) we write \( u[r, i] \) which is defined as \( u[r, i](r) = u(r)[R/r, I/i] \) for each \( r \).

The semantics of TSL(T) is defined with respect to a trace \( \rho \in (I \times R)^\omega \) of inputs and state variable valuations as follows. We assume that \( \rho = \rho_0, \rho_1, \ldots \) and that \( \rho_j = (r_j, i_j) \) and we define

\[
\begin{align*}
\rho \models p & \iff \rho_0 \models p \text{ for } p \in \langle \text{ap} \rangle, \\
\rho \models [r \leftarrow e] & \iff r_1(r) = e[R/r_0, I/i_0], \\
\rho \models \text{true} & , \\
\rho \models \text{false} & , \\
\rho \models \neg \phi & \iff \rho \not\models \phi, \\
\rho \models \phi \land \psi & \iff \rho \models \phi \land \rho \models \psi, \\
\rho \models \phi \cup \psi & \iff \exists j, \rho_j, \rho_{j+1}, \ldots \models \psi \text{ and } \forall i < j, \rho_i, \rho_{i+1}, \ldots \models \phi \\
\rho \models X\phi & \iff \rho_1, \rho_2, \ldots \models \phi.
\end{align*}
\]

The unary temporal operators eventually (\( \diamond \)) and globally (\( \Box \)) can be added using their usual definitions: \( \diamond \varphi \equiv \text{true} \cup \varphi \) and \( \Box \varphi \equiv \neg \diamond \neg \varphi \).

C. LTL Synthesis

Our algorithm relies on existing solvers for the linear temporal logic (LTL) synthesis problem which we also refer to as propositional synthesis. For completeness, we provide a brief description of the problem. A formal treatment is available in [1].

Given an LTL formula \( \phi \) containing Boolean variables \( X \) separated into the two disjoint sets \( X_I \) and \( X_O \). Consider a game between two players (environment and system) where at each point in time both players pick the values of their Boolean variables (first \( X_I \) by the environment followed by \( X_O \) by the system). The game is won by the system if the resulting infinite trace satisfies \( \phi \) and is won by the environment otherwise. The synthesis problem is: does there exist a Mealy machine strategy for the system that wins against every environment? If no such strategy exists there exists a Moore machine strategy for the environment that wins against every system. An LTL synthesis tool such as Strix [8] can determine who wins and construct a (Mealy or Moore) strategy for the winning player.

III. SYNTHESIS PROBLEM FOR TSL(T)

We want to synthesize systems from a TSL(T) specification, which we defined in the previous section. Before giving a formal definition of synthesis we need to define the systems we want to build.

Our constructed systems differ from those considered by Finkbeiner et al. [6]. They synthesize control flow models, which consist of a circuit of logic gates and vertices of uninterpreted functions that determines the values of outputs and new cells based on inputs and old cells. We instead target an extension of Mealy machines.

A. Theory Mealy and Moore Machines

A system using state variables of an unbounded domain can be hard to represent finitely. To create actual programs our systems need to have a finite structure. This is achieved by restricting all operations on state and input variables to a finite set of symbolic operations. The values of state variables need to be determined by an update chosen from a finite set \( \mathcal{U} \). The set of all update functions using updates from \( \mathcal{U} \) is denoted by \( \mathcal{U} \). To make decisions based on the values of variables we use a finite set of predicates \( P \subseteq E^F_\mathcal{P} (R \cup I) \). For a given valuation \( v = (r, i) \), let \( \mathcal{P}_v \subseteq P \) be the subset of predicates that is true in \( v \): \( \mathcal{P}_v = \{ p \in P \mid v \models p \} \). Using these we can define an extended trace as an infinite sequence \( \rho_E = (r_0, i_0, u_0, p_0), \ldots \) over \( (R \times I \times \mathcal{U} \times 2^P) \) such that for all \( j \), \( r_{j+1} = u_j[r_j, i_j] \) and \( p_j = p_j(r_j, i_j) \). The corresponding theory trace is the trace \((r_0, i_0, \ldots) \) over \( R \times I \).

We introduce the new concept of Theory Mealy Machines that are state machines with inputs and state variables that range over the theory domain. A Theory Mealy Machine \( \mathcal{M}_T = (Q, q_0, U, P, r_d, \delta, \mu) \) consists of a finite set of states \( Q \), an initial state \( q_0 \in Q \), a finite set of updates \( U \), a finite set of predicates \( P \subseteq E^F_\mathcal{P} (R \cup I) \), an initial valuation \( r_0 \in R \), a transition function \( \delta \in (Q \times 2^P) \rightarrow Q \) and an update selection function \( \mu \in (Q \times 2^P) \rightarrow U \).
A run $\sigma$ of a theory Mealy machine induced by a sequence of input valuations $i = i_0, i_1, \ldots \in I^*$ is an infinite sequence of states $Q$ and valuations $R = (q_0, r_0), (q_1, r_1), \ldots$ Any two consecutive configurations $(q_i, r_i)$ and $(q_{i+1}, r_{i+1})$ must be related by $q_{i+1} = \delta(q_i, P(r_i, i))$ and $r_{i+1} = u_i[r_i, i]$ where $u_i = \mu(q_i, P(r_i, i))$.

An extended trace is obtained from a run as the infinite sequence $(r_0, i_0, u_0, P(r_0, i_0)), \ldots$ A Theory Mealy Machine $M_T$ realizes a TSL(T) formula $\phi$ if for all input sequences $i \in I^*$ the resulting theory trace $p \equiv (i, r)$ satisfies $\phi$.

We also define Theory Moore machines, which read the updates produced by a Mealy machine and produce the inputs read by a Mealy machine. Intuitively, Mealy machines are used to show realizability of a TSL(T) specification, while Moore machines are used to show unrealizability. A Theory Moore Machine $M_T = (Q, q_0, U, P, r_0, \delta, i)$ consists of a finite set of states $Q$, an initial state $q_0 \in Q$, a finite set of updates $U$, a finite set of predicates $P \subseteq \mathbb{P}(R)$, an initial valuation $v_0 \in R$, and a transition function $\delta \in (Q \times U) \rightarrow Q$, and $i : Q \times R \rightarrow I$ is the output function.

A run $\sigma$ of a Theory Moore Machine induced by an infinite sequence of update functions $\bar{u} \in U^\omega$ is a sequence of states and valuations $(q_0, r_0), (q_1, r_1), \ldots$ Any two consecutive entries $(q_i, r_i)$ and $(q_{i+1}, r_{i+1})$ must be related by $q_{i+1} = \delta(q_i, u_i)$ and $r_{i+1} = u_i[r_i, i](q_i, r_i)$.

B. Problem Statement

Given a TSL(T) formula $\phi$, the inputs $I$, the state variables $R$, and the updates $U$, the synthesis problem asks whether there exists a Theory Mealy machine $M_T$ over $R, I, U$ such that for all input sequences $i$ the trace generated by $M_T$ satisfies $\phi$. Note that the created machine $M_T$ must use the same variables $I$ and $R$ as well as the updates $U$ as $\phi$, but it may use predicates $P$ that are not present in $\phi$.

IV. BOOLEAN ABSTRACTION

A. Propositional Encoding of TSL(T)

This subsection describes the propositional encoding of TSL(T) into LTL as proposed by Finkbeiner et al. [6]. The fact that the functions and predicates in our terms have an interpretation does not affect this translation and it is equivalent to the one for TSL.

A TSL(T) formula $\phi$ is encoded to an LTL formula $\phi_B$. Formula $\phi_B$ is obtained by replacing each update $u$ in $\phi$ by a Boolean output variable $p_u$ and each atomic proposition $ap$ by a Boolean input variable $p_{ap}$. Additionally, the formula ensures that for each variable exactly one update is active at any point in time. This results in: $\phi_B \equiv \square \left( A_n \bigwedge_{i} \left( \square_{r_i \leq e_i} \land \bigwedge_{j \neq i} \neg \square_{r_j \leq e_j} \right) \right) \land \phi[ap/p_{ap}, u/p_{u}, \ldots].$

Example 2. The TSL(T) formula from Example 1 is encoded as the LTL formula

$\phi_B \equiv \square[\bar{x} \rightarrow x \land \neg p_{x \rightarrow x} + p_{x \rightarrow x}] \land \left( p_{0 \leq x} \land p_{x \rightarrow 100} + \square(p_{0 \leq y} \land p_{y < 5}) \land \square(p_{0 \leq x} \land p_{y < 100} + (p_{x \rightarrow x} \land p_{x \rightarrow x})) \right)$,

where $p_{0 \leq x}$, $p_{x \rightarrow 100}$, $p_{0 \leq y}$, and $p_{y < 5}$ are input variables and $p_{x \rightarrow x}$ and $p_{x \rightarrow x}$ are output variables.

B. Boolean Mealy and Moore Machines

Given a set of Boolean variables $V = \{p_u | u \in U\} \cup \{p_{ap} | ap \in P\}$, we say that a Boolean trace $\rho_B = (r_0, r_1, \ldots)$ over $2^V$ corresponds to an extended trace $p_{e0}$ iff for all $j, p_{ap} \in p_{e0}(j)$ if $r_j \cap i_j = ap$ and $p_{ap} \in p_{e0}(j)$ iff $u \in i_j$. Clearly, every extended trace corresponds to a Boolean trace, but the opposite is not true, for instance, because two predicates contradict each other, or because the updates and the predicates do not match.

The LTL specifications obtained from the propositional encoding can be realized by standard Mealy machines or shown unrealizable by standard Moore machines. To make the meaning of the input and output variables clearer we will call them predicates $P$ and updates $U$ and refer to the machines as Boolean Mealy and Moore Machines. We use $U$ and $2^P$ and leave the translation into vectors of Boolean variables implicit.

A Boolean Mealy machine is a tuple $(Q, P, U, q_0, \delta, \mu_B)$ where $Q$ is a set of states, $U$ is a set of updates, $P$ is a set of predicates, $\delta_B \in Q \times 2^P \rightarrow Q$ is the transition function, and $\mu_B \in Q \times 2^P \rightarrow U$ is the update function.

A run $\sigma_B$ of a Boolean Mealy machine induced by a sequence of predicate sets $P = P_0, P_1, \ldots$ is an infinite sequence of states, updates, and predicate sets $(q_0, u_0, P_0), (q_1, u_1, P_1), \ldots$, where $q_{i+1} = \delta_B(q_i, P_i)$ and $u_{i+1} = \mu_B(q_i, P_i)$. The corresponding Boolean trace $\rho_B$ is $(u_0, P_0), (u_1, P_1), \ldots$.

A Boolean Mealy machine $M_B$ is theory consistent with respect to theory $T$ iff every trace $\rho_B$ induced by a consistent sequence $P$ has a corresponding extended trace $\rho_E$.

A Boolean Moore machine is a tuple $M_B = (Q, P, U, q_0, \delta, \mu, o)$ where $Q$ is a set of states, $P$ is a set of predicates, $q_0 \in Q$ is the initial state, $\delta_B \in Q \times U \rightarrow Q$ is the transition function, and $o \in Q \rightarrow 2^P$ is the output function.

A run $\sigma_B$ of a Boolean Moore machine induced by a sequence $\bar{u} = u_0, u_1, \ldots$ is an infinite sequence of states and predicate sets $(q_0, u_0, P_0), (q_1, u_1, P_1), \ldots$, where $q_{i+1} = \delta_B(q_i, u_i)$ and $P_{i+1} = o(q_i, u_i)$. We call a Boolean Moore machine theory consistent with respect to theory $T$ iff every trace $\rho_B = P_0, P_1, \ldots$ can be extended to an extended trace $\rho_E$.

C. Theory Consistency Analysis

We propose three criteria to locally analyze Boolean Moore machines $M_B = (Q, P, U, q_0, \delta, \mu)$ and sets of theory inputs variables $I$ for theory consistency.

- Every state must be inhabited by at least one concrete state i.e. $\text{sat}(o(q))$ for every $q \in Q$.
- Every transition must be valid for at least one pair of concrete pre and post states i.e. $\text{sat}(o(q)) \land (u \land o(q'))$ for every $(q_i, u, q_j) \in \delta$.
- Every transition must be valid for all concrete pre-states i.e. $o(q_i) \rightarrow wp(u, \exists i'. o(q_j))$ for every $(q_i, u, q_j) \in \delta$. 

Lemma 1. If these local consistency criteria are satisfied for a Boolean Moore machine \( M_B \), it is (globally) theory consistent and there exists a theory Moore machine \( M_T \) whose extended traces are consistent with the traces of \( M_B \).

Proof. Assuming the criteria are all satisfied. Every output function is satisfiable, this includes the initial state which contains an initial value \( r_0 \) of a theory machine. For every transition \( (\varphi_i, u, \varphi_j) \) where there exists a model of \( \varphi_i \) all of the models map to a model of \( \varphi_j \), by the transition property checked by the algorithm. Therefore by induction, \( r \) contains an initial value \( i \), values \( o \) traces are consistent with the traces of \( i \).

Example 3. Our approach checks consistency on a local level, the environment strategy can still be theory consistent if the third criterion is violated. The Boolean Moore machine in Figure 3 has two transitions (orange) that do not satisfy the third criterion, if the third criterion is violated. The Boolean Moore machine level, the environment strategy can still be theory consistent.

Algorithm 1: Check Theory Consistency.

```plaintext
def isconsistent(m):
    Data: Boolean Moore machine
    m = (Q, P, U, q_0, δ, o), set of variables I
    Result: ⊤ or (possibly) ⊥ with additional assumptions and predicates.
    foreach q ∈ reachable(Q) ;                      // Case 1
        if ¬sat(o(q)) then
            yield ⊥, □¬o(q);
        foreach (q_i, u, q_j) ∈ reachable(δ) ;       // Case 2
            if ¬sat(o(q_j) ∧ u ∧ o(q_i)) then
                yield ⊥, □(o(q_j) ∧ u → X¬o(q_j));
        foreach (q_i, u, q_j) ∈ reachable(δ) ;       // Case 3
            wp := weakest precondition (u, ∃!o(q_j));
            wp := quantelim(wp);
            if sat(o(q_i) ∧ ¬wp) then
                yield ⊥, □(¬wp ∧ u → X¬o(q_j));
        return ⊤;
```

Figure 3. Theory consistent machine that is not locally consistent.

SMT solver to check if the output assignment is consistent with the theory. For example, the two variables \( p_{<0} \) and \( p_{>0} \) cannot be true in the same state. If such a problem is found we generate a new assumption that rules out this assignment in every state. In the previous example, this would generate the assumption □(¬p_{<0} ∨ ¬p_{>0}).

b) Transition Consistency: Once all states produce consistent outputs and there still exists a counterstrategy, we turn towards transitions. As of now, there are no assumptions that link the state before an update was performed to the state afterward. This step checks if there are impossible transitions.

We again use an SMT solver to perform this analysis. Let’s look at the transition \( \{p_{>0}\}|x ← x + 1\} \{¬p_{>0}\} \. To check it the following SMT problem is generated \( x ≥ 5 \land x’ = x + 1 \land ¬(x’ ≥ 5) \), this is unsatisfiable and we can generate an assumption to eliminate it □(p_{>0} ∧ [x ← x + 1] → Xp_{>0}).

c) New Predicates: Another case is that a transition is possible for some, but not all of the values. For instance, the triple \( \{p_{<0}\}|x ← x + 1\} \{p_{>0}\} \) does not hold for all values of \( x \). This shows that our current abstraction might not be precise enough to correctly describe this transition and we need an additional predicate. We calculate the weakest precondition of the post state given the updates of the transition. This gives us the predicate \( x ≥ -1 \). If states in the pre-state are not included in the weakest precondition they can not take the transition. The weakest precondition can also be used as the new predicate to distinguish which concrete states may take a transition. In case there are input variables the future inputs are existentially quantified in the post-condition. This results in a natural extension of the weakest precondition, it contains all states that can reach one of the valid post-conditions.

Lemma 2. If for a Boolean Moore machine \( M_B \) Algorithm 1 returns inconsistent together with assumptions \( ψ \) every Theory Moore machine \( M_T \) — where \( M_B \) and \( M_T \) share the inputs, state variables, and updates — satisfies \( ψ \).
Proof. Algorithm 1 can produce three different types of assumptions $\psi_s, \psi_t, \psi_p$ corresponding to the three cases of the algorithm. Let $M_T$ be an arbitrary theory Moore machine. Let $\psi_s$ be $\Box \neg o(q)$ for an unsatisfiable $o(q)$. $M_T$ must define an output valuation for every state because $o(q)$ is empty no state in any $M_T$ can produce such an output. Therefore all $M_T$ satisfy $\psi_s$.

Let $\psi_t$ be $\Box (o(q_i) \land u \rightarrow X \neg o(q_j))$ where $\neg sat(o(q_i) \land u \land o(q_j)')$ and $sat(o(q_j))$. None of the values satisfying $o(q_i)$ have a successor in $o(q_j)$ after performing $u$. The added constraint is equivalent to $\neg o(q_i) \land \neg u \land o(q_j)' \leftrightarrow \neg o(q_i) \lor \neg u \lor o(q_j)' \leftrightarrow o(q_i) \land u \rightarrow X \neg o(q_j)$. All transitions in all $M_T$ satisfy this property at all points in time.

Let $\psi_p$ be $\Box (\neg p \land u \rightarrow X \neg o(q_j))$ where $p$ is the weakest precondition of $o(q_j)$ under $u$ and $sat(o(q_i) \land u \land \neg o(q_j))$. By the definition of weakest precondition, no value in $\neg p$ leads to $o(q_j)$ when performing $u$. This also holds in the presence of inputs. The quantifier elimination procedure leads to the weakest precondition for unknown inputs at the next time step. All transitions in all $M_T$ will lead from $\neg p$ to $\neg o(q_j)$ when performing $u$.

All added constraints are satisfied by all states and transitions in all $M_T$. The constraints only talk about individual states and transitions therefore also all traces in $M_T$ satisfy these constraints and $\forall M_T, M_T = \psi$. $\square$

D. Generalizing Counterexamples

The counterexamples generated by Algorithm 1 only block the exact state or transition present in the counterstrategy. To achieve faster and better convergence it is necessary to generalize these counterexamples. Generalization of counterexamples is done using an algorithm to find an unsatisfiable core, i.e., a small (not necessarily minimal) subset of clauses such that their conjunction is unsatisfiable.

This is done for each assumption returned from Algorithm 1 as follows. For $\Box \neg o$ we compute $o_{usc}$ = unsatcore($o$) and produce the generalized assumption $\Box \neg o_{usc}$. For $\Box (o1 \land u \rightarrow X \neg o2)$ we compute $o_{usc}, o_{usc}2 = unsatcore(o1 \land u \land o2')$ by keeping track of where each conjunct originated. This is then turned back into the generalized assumption $\Box (o1_{usc} \land u_{usc} \rightarrow X \neg o_{usc})$.

Using unsat cores in this way allows us to find smaller counterexamples that do not depend on superficial information. Therefore, the counterexamples also block situations where unrelated predicates or updates are different.

V. SYNTHESIS ALGORITHM

A. Synthesis

Our synthesis procedure is shown in Algorithm 2. The procedure starts with a specification in TSL(T) that is translated to an LTL specification as described in Section IV-A. The LTL specification is given to a synthesis tool for propositional LTL. If the synthesis tool finds a realizing system, this system encodes a solution for the TSL(T) synthesis problem. If not, the LTL synthesizer gives us a counterstrategy, which we analyze to find any inconsistencies with the theory. If the counterstrategy is theory-consistent, it gives us a counterexample to the realizability of the TSL(T) formula. If the theory solver shows that the counterexample is theory-inconsistent, we refine the specification, strengthening it to exclude the inconsistency observed. We illustrate the approach using an example and then prove its correctness.

Example 4. We apply Algorithm 2 to synthesize a system from the specification $\phi$ given in Example 1. To better illustrate the algorithm we will only introduce one new assumption per iteration. The machines created during execution (5 counter strategies in the form of Moore machines and one strategy in the form of a Mealy machine) are depicted in Figure 4. We refer to the specification in step n as $\phi_n \triangleq \Box \bigwedge_{k=1}^{n+1} \psi_k \rightarrow \phi$ where $\psi_k$ are the assumptions added in step $k$.

The LTL encoding of $\phi_1 = \phi$ is $\phi_B$. (See Example 2) Propositional synthesis results in the counterstrategy shown in Figure 4.1. Algorithm 1 reveals that the second state of this counterexample (shown in red) is inconsistent, because $\neg(0 \leq x) \land \neg(x < 100)$ is unsatisfiable. We obtain the new assumption $\psi_{1} = \Box(0 \leq x \lor x < 100)$, Note that this is a more general assumption than just the negated state formula. Attempting to synthesize a system for $\phi_1$ (we will omit the LTL encoding step from now on) results in the counterstrat-
Any Boolean system shown in Figure 4.2. This time, all outputs are consistent, but the transition \([x \leftarrow x - 1]\) is inconsistent: If \(x < 100\) and \(0 \geq i\) and we apply the update \([x \leftarrow x - i]\), it cannot be that \(x \not< 100\) in the following state. We obtain the assumption 
\[
\psi_2 \triangleq \square (x < 100 \land 0 \leq i \land [x \leftarrow x - i] \rightarrow X' x < 100).
\]

Boolean synthesis for \(\phi_2\) results in a similar counterstrategy (Figure 4.3), this time the transition \([x \leftarrow x + i]\) is inconsistent and \(\psi_3 \triangleq \square (0 \leq x \land 0 \leq i \land [x \leftarrow x + i] \rightarrow X' 0 \leq x).

Synthesis for \(\phi_3\) leads to the counterstrategy shown in Figure 4.4. The first two consistency checks pass, but case 3 reports that the transition \([x \leftarrow x - i]\) is only valid for some of the possible values in the first state. We learn the new predicate 
\[
0 \leq x - i \text{ and the assumption } \psi_4 \triangleq \square (0 \leq x - i \land [x \leftarrow x - i] \rightarrow X' 0 \leq x) \land \square \neg (0 \leq i \land 0 \leq x - i \land 0 \leq x).
\]
This includes the assumption obtained from the transition as well as one to prevent state inconsistencies with the new predicate.  

Running the Boolean synthesis algorithm again results in the counterstrategy in 4.5. The transition \([x \leftarrow x + i]\) is inconsistent and we add 
\[
\psi_5 \triangleq \square (0 \not\geq x - i \land i < 5 \land [x \leftarrow x + i] \rightarrow X' x < 100).
\]

Boolean synthesis is executed for the last time on \(\phi_5\). This time, the synthesis tool produces a Boolean system satisfying the specification this corresponds to the system shown in Figure 1.

**Theorem 1.** Any Boolean system \(M_B\) returned by Algorithm 2 can be converted to a theory system \(M_T\) that satisfies \(\phi\).

**Proof.** To obtain \(M_T\) an initial valuation \(r_0\) that satisfies \(\phi\) at point 0 has to be chosen. All other components are the same as in \(M_B\). Let \(\phi' = \psi \rightarrow \phi\) be the last specification used in the algorithm and \(\psi\) all the added assumptions. We know \(M_B \models \phi'_B\) and that \(M_B\) and \(M_T\) share the same extended traces. Therefore, \(M_T\) satisfies \(\phi'\). From Lemma 2 follows that \(M_T \models \psi\). Thus \(M_T \models \phi\). \(\square\)

Even though our algorithm is not guaranteed to terminate it can prove unrealizability in certain cases. For \(x = 0 \rightarrow \square ([x \leftarrow x + 1] \land x < 3)\) we can perform two refinement steps and learn the new predicates \(x \geq 2\) and \(x \geq 1\). Using these the propositional synthesis tool can build a consistent environment strategy. There are no conflicts that could be used to further refine the specification. This shows that the specification is unrealizable.

**Theorem 2.** If Algorithm 2 returns unrealizable there is no \(M_T \models \phi\) and \(\phi\) is unrealizable by machines using the updates \(U\).

**Proof.** If there exists a machine \(M'_T \models \neg \phi\) there is no machine \(M_T \models \phi\). The propositional synthesis tool provides us with a machine \(M_B \models \neg \phi'\) where \(\phi' = \psi \rightarrow \phi\) for assumptions \(\psi\). The consistency check results in consistent, so by Lemma 1 there exists a \(M'_T \models \neg \phi'\). According to Lemma 2 \(\psi\) is satisfied by \(M'_T\). Therefore, \(M'_T \models \neg \phi\) and no \(M_T \models \phi\). \(\square\)

---

1 We include this here instead of in its own step to simplify the example.

---

**B. Limitations**

Our algorithm is not guaranteed to terminate. In Section VI we discuss multiple specifications that can be successfully synthesized or where unrealizability can be shown. In this section, we show two exemplar cases for which our algorithm will not terminate.

Our algorithm cannot handle reachability properties where the number of required steps depends on the concrete value of a state variable and is unbounded. The specification \(0 \leq x \rightarrow (\diamond (x < 0) \land \square ([x \leftarrow x + 1] \lor [x \leftarrow x - 1]))\) with the state variable \(x\) is an example of this. The specification is obviously realized by a system always using the update \([x \leftarrow x - 1]\). However, we would add the new predicates \(x \geq 1, x \geq 2,\ldots\) without terminating.

A similar problem can occur for unrealizability. For \(x = 1 \rightarrow (\diamond (x = 0) \land \square ([x \leftarrow x + 1]))\) we learn the predicates \(x = -1, x = -2,\ldots\) without terminating. However, the predicate \(x > 1\) would allow us to prove unrealizability.

**C. Decidable fragment**

**Theorem 3.** The TSL synthesis problem for the theory of equality is decidable.

Algorithm 2 will always terminate if the set of predicates that Algorithm 1 can generate is finite. For a finite set of predicates the assumptions that can be added by cases, one and two are also finite. Since the assumptions block the counterstrategy from reappearing this means there can not be infinitely many counter strategies and the synthesis algorithm will terminate with the correct answer. The theory of equality only allows updates to move values. Iterating the weakest precondition can only create finitely many predicates, for the equality theory also the quantifier elimination cannot introduce new constants. Thus the number of possible predicates is finite and the problem is decidable.

---

**VI. EXPERIMENTAL EVALUATION**

We implemented our algorithm in our tool Raboniel\(^2\). Our implementation relies on several external tools: ts tools [6] is used for parsing TSL and to perform the propositional encoding, strix [8] is used for LTL synthesis, and Z3 [11] is used as the SMT solver. When performing counterexample analysis using Algorithm 1 we add all assumptions from the same case before we start the next iteration. The obtained theory Mealy machine can be compiled into a Python program.

**A. Extended running example**

The first experiment is an extension of Example 4. We change two parameters in the specification. The system is no longer allowed to change between the two updates at every step. Instead after changing the update, it has to use the new update for the next \(c\) steps. This shows how our algorithm deals with more complex temporal properties. We also varied the size of the intervals for \(x\) and \(i\) demonstrating that our algorithm is independent of the size of the concrete state

\(^2\)https://doi.org/10.5281/zenodo.5647461
space. The results are listed in Table I including the used parameters \((c, x_{\text{max}}, i_{\text{max}})\), the number of refinements, the number of states in the minimized system, the number of learned predicates during the whole execution and the total run time in seconds. The table includes realizable as well as unrealizable configurations. The different ranges for \(x\) and \(i\) show how our approach can handle state spaces symbolically, it behaves the same whether there are 100 or 100000 concrete states, these number of concrete states is also far above what can be solved with explicit states in LTL. The larger values of \(c\) require the system to plan further ahead by limiting how often it can switch its output, this requires a second state and a few additional predicates.

### B. Elevator

A classic example for reactive synthesis is a controller for an elevator. The single state variable \(floor\) represents the current position of the elevator. It can start anywhere between the first floor and the maximum floor and is not allowed to leave this interval. The controller has three options: move the elevator up or down or stay in the same position. Every floor has to be visited infinitely often. The results are shown in Table II as type simple. We varied the number of floors of the building to show how our algorithm scales with more complex specifications. No new predicates are learned as a sufficient number of predicates is already included in the specification (equality tests for every floor are part of the liveness properties). The required time seems to grow exponentially with the growing number of floors. This leads to growing propositional synthesis problems (which is worst case double-exponential). The number of states stays constant because most of the complexity is part of the predicates e.g. the position of the elevator. The overall time is still reasonable even for a large number of floors.

A different version of this specification is shown in Table II as type signal. In this version, the environment controls a variable \(signal\) to select the floor the elevator has to reach which is stored in the state variable \(target\). This results in a more complex specification with worse run time, which is dominated by propositional synthesis.

### C. Cyber-Physical Systems

The previous examples all used linear integer arithmetic. We can also use other SMT theories like linear real arithmetic (LRA). Using reals allows us to model linear cyber-physical systems. This example is inspired by Belta et al. [12] chapter 9, a system of two coupled water tanks with linear dynamics; one tank drains \((x2)\) and the other one \((x1)\) is refilled by the controller. An illustration is depicted in Figure 5. We discretize the inputs (refill tank \(x1\)) with two values \((0\) and \(0.0003\), represented as different updates. We created two variants of the system. The first one is a safety specification where the water level of both tanks has to be kept between \(0.1\) and \(0.7\).

Synthesis of this system takes 31 seconds and 4 refinements. The resulting system only has a single state, but 13 new predicates where required.

The second version consists of only one water tank, but requires the liveness property: whenever the water level falls below \(0.1\) it has to eventually exceed \(0.4\). A system realizing that specification can be synthesized using 18 refinements in 95 seconds (9 new predicates), it consists of 2 states. These examples demonstrate that our tool can handle updates with more complex operations. This leads to a large number of new predicates, but can still be synthesized in less than two minutes.

### D. Comparison with Related Work

The TSL paper by Finkbeiner et al. [6] contains various examples of TSL specifications. However, most of them do not require any refinement and the first LTL approximation is already realizable. For these examples, our tool would perform the same, because no theory refinement is used. A small number of examples required refinement. We converted two of them by replacing the uninterpreted functions for increment and decrement with native integer operations. The implementation of their refinement approach is not publicly available, we thus compare our results to the numbers reported in their paper. The experiment “TwoCountersInRange” took our tool 8 refinements and 1677s compared to their 173s. The experiment “OneCounterGUI” took our tool only 4 refinements and 17.2s, this is a factor 100 faster than their 1767s. These results suggest that both tools can outperform the other by an order of magnitude, depending on the example.

We also compared our tool using a benchmark set of safety games on infinite grid worlds that was introduced by Neider and Topcu [13]. We compare with the following tools: the

---

**Table I: Results for the extended running example.**

<table>
<thead>
<tr>
<th>(c)</th>
<th>(x_{\text{max}})</th>
<th>(i_{\text{max}})</th>
<th># refine.</th>
<th># states</th>
<th># new pred.</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1.3</td>
</tr>
<tr>
<td>3</td>
<td>100 000</td>
<td>50</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>100 000</td>
<td>50</td>
<td>10</td>
<td>2</td>
<td>4</td>
<td>3.1</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>110</td>
<td>6</td>
<td>unreal.</td>
<td>3</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>60</td>
<td>4</td>
<td>unreal.</td>
<td>2</td>
<td>0.8</td>
</tr>
</tbody>
</table>

**Table II: Results for the elevator.**

<table>
<thead>
<tr>
<th>type</th>
<th># floors</th>
<th># refinements</th>
<th># states</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>3</td>
<td>13</td>
<td>2</td>
<td>3.1</td>
</tr>
<tr>
<td>simple</td>
<td>4</td>
<td>11</td>
<td>3</td>
<td>3.7</td>
</tr>
<tr>
<td>simple</td>
<td>5</td>
<td>15</td>
<td>4</td>
<td>8.2</td>
</tr>
<tr>
<td>simple</td>
<td>8</td>
<td>21</td>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>simple</td>
<td>10</td>
<td>24</td>
<td>4</td>
<td>185</td>
</tr>
<tr>
<td>signal</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>signal</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>217</td>
</tr>
<tr>
<td>signal</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>1424</td>
</tr>
</tbody>
</table>

![Fig. 5. Water tanks system](image-url)
logic-based synthesis tools ConSynth [14], JSyn-VG [15], and GenSys [16]; the automata-learning-based tools SAT-Synth and RPNI-Synth [13]; as well as the decision-tree-learning tool DT-Synth [17]. These experiments are shown in Table III our tool is listed as Raboniel, the results for the other tools are reproduced from [16]. Our tool is able to solve 6 out of the 7 benchmarks within 15 minutes. The execution time of our tool is on the lower end of the spectrum. However, TSL(T) allows us to express and handle more sophisticated specifications. Most other tools (except ConSynth and JSyn-VG) only support safety properties and would not be able to handle the other examples shown in this paper.

Table III

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C</th>
<th>J</th>
<th>D</th>
<th>S</th>
<th>R</th>
<th>G</th>
<th>Raboniel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Box</td>
<td>3.7</td>
<td>0.6</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
<td>1.9</td>
</tr>
<tr>
<td>Box Limited</td>
<td>0.4</td>
<td>1.7</td>
<td>0.1</td>
<td>0.4</td>
<td>0.5</td>
<td>0.2</td>
<td>0.6</td>
</tr>
<tr>
<td>Diagonal</td>
<td>1.9</td>
<td>4.0</td>
<td>2.4</td>
<td>1.3</td>
<td>0.5</td>
<td>0.2</td>
<td>10.9</td>
</tr>
<tr>
<td>Evasion</td>
<td>1.5</td>
<td>0.5</td>
<td>0.2</td>
<td>81</td>
<td>0.1</td>
<td>0.7</td>
<td>5.4</td>
</tr>
<tr>
<td>Follow</td>
<td>—</td>
<td>1.2</td>
<td>0.3</td>
<td>88.9</td>
<td>—</td>
<td>0.7</td>
<td>—</td>
</tr>
<tr>
<td>Solitary Box</td>
<td>0.4</td>
<td>0.9</td>
<td>0.1</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
<td>0.5</td>
</tr>
<tr>
<td>Square 5x5</td>
<td>—</td>
<td>6.5</td>
<td>2.5</td>
<td>0.6</td>
<td>0.2</td>
<td>0.3</td>
<td>75.1</td>
</tr>
</tbody>
</table>

VII. RELATED WORK

The first paper on TSL [6] introduces this logic as a way to do synthesis while separating control flow and data processing. Reactive synthesis is used to build a control flow model which describes how the uninterpreted functions are combined and which of them is used when based on a logic circuit. This model can then be instantiated and translated to a functional reactive program (FRP)[18]. Our approach has less separation of data and control, by supporting theories we can reason about a lot of operations and construct systems that would not be possible using uninterpreted functions. We also directly create executable code without the intermediary FRP. Another major difference is the analysis of counter strategies. Finkbeiner et al. use an algorithm specific to uninterpreted functions that checks all possible traces up to a certain length. We have shown that consistency checking can also be done by local checks in a theory-independent way. That way we can also learn new predicates which allow for the synthesis of otherwise impossible specifications and prove unrealizability.

The extension to TSL(T) was first done by Finkbeiner et al. [7] they study uninterpreted functions and Presburger arithmetic and provide a search-based algorithm to check satisfiability. However, they did not look into synthesis.

Another recent extension of TSL is by Choi et al. [19]. They describe a different approach to adding arithmetic to TSL using syntax-guided synthesis (SyGuS) [20]. The TSL formula is translated into sequential SyGuS problems and the solutions are used to create assumptions. This technique cannot create new predicates and thus will not be able to solve problems such as our running example. Their solution was developed independently and in parallel to our approach.

Other techniques for reactive synthesis beyond Booleans are: Reactive synthesis from register automata specifications has been studied[2], [4], [3], [21]. These models allow comparison (equality/inequalities) of data values, but no operations. Multiple decidable fragments have been identified. Another approach uses variable automata [5], [22] specifications these can perform arithmetic the authors also identified a decidable fragment. While for both register and variable automata strong theoretical results have been achieved we are not aware of any empirical evaluations. There are also synthesis tools that specifically target cyber-physical systems [23], [24] these often rely on finite or receding horizons instead of infinite traces. counterexample guided methods have also been used for program synthesis [25] and model synthesis [26].

VIII. CONCLUSION AND FUTURE WORK

The algorithm presented in this paper performs specification refinement in a pure lazy way. That is new assumptions are only added when they are encountered in a counterexample. Performing some analysis upfront and after learning new predicates has the potential to significantly improve the run time. Testing for incompatibilities between predicates would be an obvious target for this. Another extension would be new strategies for learning predicates and heuristics to prevent learning unnecessary predicates (slowing down propositional synthesis).

We presented a synthesis procedure for temporal stream logic modulo theories. Our algorithm is based on a CEGAR [10] loop and translation to propositional LTL synthesis. The synthesis problem for TSL modulo theories, in general, is undecidable. However, we can synthesize systems or prove unrealizability in many cases. Huge state spaces can be handled by using a symbolic representation during synthesis. Some specifications require new predicates, in many cases, we are able to automatically find these.

REFERENCES


