Adaptive Reactive Synthesis for LTL and LTL$_f$ Modulo Theories

Andoni Rodríguez[1,2] **and César Sánchez**[1]

[1]IMDEA Software Institute
[2]Universidad Politécnica de Madrid
{andoni.rodriguez,cesar.sanchez}@imdea.org

Abstract

Reactive synthesis is the process of generate correct controllers from temporal logic specifications. Typically, synthesis is restricted to Boolean specifications in *LTL*. Recently, a Boolean abstraction technique allows to translate *LTL*$_\mathcal{T}$ specifications that contain literals in theories into equi-realizable *LTL* specifications, but no full synthesis procedure exists yet. In synthesis modulo theories, the system receives valuations of environment variables (from a first-order theory $\mathcal{T}$) and outputs valuations of system variables from $\mathcal{T}$. In this paper, we address how to syntheize a full controller using a combination of the static Boolean controller obtained from the *Booleanized LTL* specification together with on-the-fly queries to a solver that produces models of satisfiable existential $\mathcal{T}$ formulae. This is the first synthesis method for *LTL* modulo theories. Additionally, our method can produce adaptive responses which increases explainability and can improve runtime properties like performance. Our approach is applicable to both *LTL* modulo theories and LTL$_f$ modulo theories.

Introduction

Reactive synthesis is the problem of automatically producing a system that models a given temporal specification, where the Boolean variables (i.e., atomic propositions) are split into those controlled by the environment and those controlled by the system. Realizability is the related decision problem of deciding whether such a system exists. These problems have been widely studied (Pnueli and Rosner 1989), specially in the domain of Linear Temporal Logic (*LTL*) (Pnueli 1977). Realizability corresponds to an infinite game where players alternatively choose the valuations of the Boolean variables they control. A specification is realizable if and only if the system has a strategy such that the specification is satisfied in all plays played according to the strategy. The system is extracted from a winning system strategy. Both reactive synthesis and realizability are decidable for *LTL* (Pnueli and Rosner 1989). *LTL* modulo theories (*LTL*$_\mathcal{T}$) is the extension of *LTL* where Boolean atomic propositions can be literals from a (multi-sorted) first-order theory $\mathcal{T}$. Realizability of *LTL*$_\mathcal{T}$ specifications is decidable under certain conditions over $\mathcal{T}$ (Rodriguez and Sánchez 2023), using a Boolean abstraction or *Booleanization* method that translates a specification $\varphi_\mathcal{T}$ in *LTL*$_\mathcal{T}$ into an equi-realizable *LTL* $\varphi_\mathbb{B}$ formulae: i.e. $\varphi_\mathcal{T}$ is realizable if and only if $\varphi_\mathbb{B}$ is realizable.

However, to perform *LTL*$_\mathcal{T}$ reactive synthesis it is not enough to synthesize a controller for the Booleanized specifications because the system will receive and will have to produce values from theory variables. Some previous synthesis methods try to create statically a controller that always produces the same outputs for the same inputs (e.g. (Katis et al. 2016)) but are incomplete for many $\mathcal{T}$, since they need to compute Skolem functions. In this paper, we propose a general method that uses *on-the-fly* procedures to dynamically produce outputs as the results of computing models of existential $\mathcal{T}$ formulae. Concretely, the method we propose statically receives an *LTL*$_\mathcal{T}$ specification φ, Booleanizes φ using (Rodriguez and Sánchez 2023) and synthesizes a controller S using standard methods. Then, dynamically S is combined with a tool that can produce models of satisfiable $\mathcal{T}$ formulae (e.g., an SMT solver) which collaborate in tandem at each step of the execution. To guarantee that the reaction is produced at every step, we require that $\mathcal{T}$ has an efficient procedure to provide models of existential fragments of $\mathcal{T}$. Our approach does not guarantee termination using semi-decidable $\mathcal{T}$, but still yields a partial solution for such cases. We also use an additional component, called *partitioner*, which discretizes the environment $\mathcal{T}$-input providing a suitable input for the Boolean controller (but this is an easy by-product of the Booleanization procedure and can be computed statically). As far as we know, this is the first successful reactive synthesis procedure for *LTL*$_\mathcal{T}$ specifications.

Preliminaries

LTL, LTL$_f$ and reactive synthesis. We use synthesis of LTL (Pnueli 1977) specifications. The syntax of LTL is:

$$\varphi ::= T \mid a \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where a ranges from an atomic set of proposition AP, $\vee$, $\wedge$ and $\neg$ are the usual Boolean disjunction, conjunction and negation, and $\bigcirc$ and $\mathcal{U}$ are the *next* and *until* temporal operators (other derived operators include $\mathcal{R}$, $\Diamond$ and $\Box$). The semantics of LTL associate traces $\sigma \in \Sigma^\omega$ with formulae. Our technique also applies to LTL$_f$ specifications (Manna and Pnueli 1995; Giacomo and Vardi 2013), that borrow the

syntax from LTL, but whose semantics is given in terms of finite traces represented as finite words over the alphabet.

Reactive synthesis (Thomas 2008; Finkbeiner 2016) is the problem of producing a system from an LTL specification, where the atomic propositions in the formula are split into propositions controlled by the environment and those controlled by the system. Synthesis corresponds to a turn-based game where, in each turn, the environment produces values of its variables (inputs) and the system responds with values of its variables (outputs). A play is an infinite sequence of turns. The system player wins a play if the trace of the play satisfies the specification φ. A strategy of a player is a map from positions into a move for the player. A play is played according to a strategy if all the moves of the corresponding player are played according to the strategy. A strategy is winning for a player if all the possible plays played according to the strategy are winning for the player. Note that, depending on the fragment of LTL used, the synthesis problem has different complexities (with general 2EXPTIME complete).

***LTL*$_\mathcal{T}$ and Boolean abstraction.** We use *LTL*$_\mathcal{T}$ as the extension of LTL where propositions are replaced by literals from a first-order theory $\mathcal{T}$ (for example Presburger arithmetic or linear real arithmetic). In realizability and synthesis for *LTL*$_\mathcal{T}$, the (theory) variables that occur in the literals of a specification φ are split into those variables controlled by the environment (denoted by $\overline{x}$) and those controlled by the system ($\overline{y}$), where $\overline{x} \cap \overline{y} = \emptyset$. We use $\varphi(\overline{x}, \overline{y})$ to remark that $\overline{x} \cup \overline{y}$ are the variables occurring in φ. The alphabet $\Sigma_\mathcal{T}$ is now a valuation of the variables in $\overline{x} \cup \overline{y}$, which can be infinite in many theories. A trace is an infinite sequence of valuations, which induces an infinite sequence of Boolean valuation of the literals occurring in φ and, in turn, an evaluation of the temporal formula. Deciding *LTL*$_\mathcal{T}$ realizability corresponds to an infinite game with an infinite arena where positions may have infinitely many successors if the ranges of the variables controlled by the system and the environment are infinite. For instance, literal $(x \geq 2)$ interpreted in integers can be satisfied with $x = 2$, $x = 3$, $x = 4$, etc.

In this paper we introduce a synthesis algorithm for *LTL*$_\mathcal{T}$ based on the Boolean abstraction method (Rodriguez and Sánchez 2023), which transforms an *LTL*$_\mathcal{T}$ specification into an equi-realizable *LTL* Boolean specification. The Boolean abstraction method requires that the $\exists^*\forall^*$ fragment of the theory is decidable. The method used in this paper produces a formula in the same temporal fragment as the original formula (e.g., starting from a safety formula another safety formula is generated). The generated *LTL* formula can be discharged into a synthesis engine that can generate a controller for the system player if the specification is realizable. For simplicity in the presentation, we illustrate our method with safety formulae and arithmetic specifications. Concretely, we will consider $\mathcal{T}_\mathbb{Z}$ (i.e., linear integer arithmetic) and $\mathcal{T}_\mathbb{R}$ (i.e., non-linear real arithmetic), which are decidable.

The Boolean abstraction procedure takes an input formula $\varphi_\mathcal{T}$ with literals l_i and produces a new specification $\varphi_\mathbb{B} = \varphi[l_i \leftarrow s_i] \wedge \Box\varphi^{extra}$, where s_i are fresh Boolean variables and $\varphi^{extra} \in \mathbb{B}$. The core of the algorithm is the additional subformula φ^{extra} which uses the freshly introduced variables s_i—controlled by the system—as well as additional Boolean variables $\overline{e}_k$ controlled by the environment and captures that, for each possible $\overline{e}_k$, the system has the power to choose a response among a specific s_i. The extra requirement captures precisely the finite collection of input decisions of the environment (partitions of the environment space of valuations) and the resulting (finite) choices of the system to respond (partitions of the system choices that results in the same Boolean valuations of the literals).

Motivating running example. As for an example of reactive specifications in LTL$_\mathcal{T}$, let $\Box$ be the usual *globally* operator in LTL and $\bigcirc$ the *next* operator. Consider $\varphi_\mathcal{T} = \Box(R_0 \wedge R_1)$ as the running example for the paper, where

$$R_0 : (x < 2) \rightarrow \bigcirc(y > 1) \qquad R_1 : (x \geq 2) \rightarrow (y < x)$$

In $\varphi_\mathcal{T}$, $x \in \mathcal{T}$ belongs to the environment and $y \in \mathcal{T}$ belongs to the system. Note that $\varphi_\mathcal{T}$ is not realizable for $\mathcal{T} = \mathcal{T}_\mathbb{Z}$, since, if at a given time instant t, the environment plays $x = 0$, and hence $(x < 2)$ holds, then y must be greater than 1 at time $t + 1$. Then, if at $t + 1$ the environment plays $x = 2$, then $(x \geq 2)$ holds but there is no y such that both $(y > 1)$ and $(y < 2)$. However, for $\mathcal{T} = \mathcal{T}_\mathbb{R}$, φ is realizable (consider the system strategy to always play $y = 1.5$). The Boolean abstraction method transforms $\varphi_\mathcal{T}$ into a purely Boolean specification $\varphi_\mathbb{B}$ that allows to perform automatic *LTL* realizability checking. For instance, for $\mathcal{T} = \mathcal{T}_\mathbb{Z}$, the Booleanized version of $\varphi_\mathcal{T}$ is the following:

$$\varphi_\mathbb{B} = \varphi'' \wedge \Box(\varphi^{legal} \rightarrow \varphi^{extra}),$$

where φ^{legal} encodes that e_0, e_1 and e_2 characterize a partition of the input decisions of the environment $(e_0 \vee e_1 \vee e_2) \wedge (e_0 \rightarrow \neg(e_1 \wedge e_2)) \wedge (e_1 \rightarrow \neg(e_0 \wedge e_2)) \wedge (e_2 \rightarrow \neg(e_0 \wedge e_1))$. Also $\varphi'' = (s_0 \rightarrow \bigcirc s_1) \wedge (\neg s_0 \rightarrow s_2)$ is a direct translation of $\varphi_\mathcal{T}$, where s_0 abstracts the literal $(x < 2)$, s_1 abstracts $(y > 1)$ and s_2 abstracts $(y < x)$. Solely replacing literals with fresh system variables over-approximates the power of the system, therefore we need an additional formula φ^{extra} that encodes the original power of each player in $\varphi_\mathcal{T}$:

$$\varphi^{extra} : \begin{pmatrix} & (e_0 & \rightarrow & s_{01\overline{2}} \vee s_{0\overline{1}2} \vee s_{0\overline{12}}) \\ \wedge & (e_1 & \rightarrow & s_{\overline{0}1\overline{2}} \vee s_{\overline{0}\overline{1}2}) \\ \wedge & (e_2 & \rightarrow & s_{\overline{0}12} \vee s_{\overline{0}1\overline{2}} \vee s_{\overline{0}\overline{12}}) \end{pmatrix},$$

where $e_0, e_1, e_2 \in \mathbb{B}$ belong to the environment and where $s_{01\overline{2}} = (s_0 \wedge s_1 \wedge \neg s_2)$, $s_{0\overline{1}2} = (s_0 \wedge \neg s_1 \wedge s_2)$, $s_{0\overline{12}} = (s_0 \wedge \neg s_1 \wedge \neg s_2)$, $s_{\overline{0}12} = (\neg s_0 \wedge s_1 \wedge s_2)$, $s_{\overline{0}1\overline{2}} = (\neg s_0 \wedge s_1 \wedge \neg s_2)$ and $s_{\overline{0}\overline{1}2} = (\neg s_0 \wedge \neg s_1 \wedge s_2)$, where $s_0, s_1, s_2 \in \mathbb{B}$ belong to the system. Sub-formulae $s_{01\overline{2}}, s_{0\overline{1}2}, s_{0\overline{12}}, s_{\overline{0}12}, s_{\overline{0}1\overline{2}}$ and $s_{\overline{0}\overline{12}}$ represent the *choices* of the system, that is, given a decision e_k of the environment, the system can *react* with one of the choices c_i in the disjunction implied by e_k. Note that φ^{legal} encodes that e_0, e_1, e_2 is a (finite) partition in the domain of the (infinite) valuations of the environment, where e_0 abstracts its decision x such that $(x < 2)$, e_1 represents x such that $(x = 2)$ and e_2 represents $(x > 2)$. Note that if the considered $\mathcal{T}$ is different, $\varphi_\mathbb{B}$ may also differ.

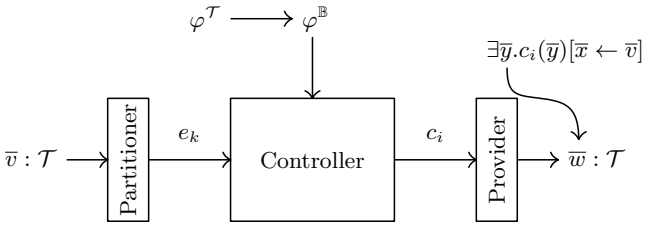

Figure 1: The *on-the-fly* synthesis architecture.

Description of the Approach

On-the-fly Architecture

Overall Architecture. For synthesis modulo theories it is not enough to synthesize a controller for the Booleanized $LTL_{\mathcal{T}}$ specifications, because the actual controller will receive inputs in $\mathcal{T}$ from the environment and produce outputs from complex values in $\mathcal{T}$. For instance, consider the environment controls an integer variable x and the system controls an integer variable y in the specification $\varphi^{\mathcal{T}} = \square(y > x)$, then the system can satisfy $\varphi^{\mathcal{T}}$ by always returning any y such that $(y > x)$. In this paper we propose a general approach to do so, shown in Fig. 1, which we call *on-the-fly $LTL_{\mathcal{T}}$ synthesis*. This method consists on computing statically a Boolean controller for $\varphi^{\mathbb{B}}$ (which has been Booleanized from $\varphi^{\mathcal{T}}$), and dynamically combine it with a method to provide models from formulae in $\mathcal{T}$. At runtime, at each instant of time, (1) given the valuations $[\overline{x} \leftarrow \overline{v}]$ of the environment (where $\overline{v}$ are actual input values for each environment variable $x \in \mathcal{T}$), then (2) the *partitioner* discretizes this valuation generating a Boolean input for the Boolean controller; (3) the controller responds with a choice $c_i \in \mathbb{B}$ (which corresponds to a verdict on the Boolean valuations of literals in the formula). Our controller still needs to produce actual values of the output variables that make the verdict of the literals be as in c_i, for which a formula of the form $\exists\overline{y}.\ c_i^{\mathcal{T}}(\overline{y})$ is generated (where $c_i^{\mathcal{T}}(\overline{y})$ is the $\mathcal{T}$ formula that contains one conjunction per literal, and the input variables replaced by their values). This formula represents all the values that the system controls, that result in the choice c_i that the Boolean controller has output. By the correctness of the Booleanization process this formula must be satisfiable. Stage (4), called *provider*, uses an SMT solver to produce a model $\overline{w}$ of $\exists\overline{y}.\ c_i^{\mathcal{T}}(\overline{y})$ so $[\overline{y} \leftarrow \overline{w}]$ will guarantee the original specification $\varphi^{\mathcal{T}}$. Note that we replace $\overline{x}$ by the input valuation $\overline{v}$ in $c_i^{\mathcal{T}}(\overline{y})$, so $c_i^{\mathcal{T}}(\overline{y})$ only has $\overline{y}$ as variables. We now describe in detail the three parts of the architecture.

Partitioner. At each timestep, the partitioner receives a valuation $\overline{v} \in E_{\mathcal{T}}$ of the environment variables $\overline{x}$. The goal of the partitioner is to find the discrete partition of the input universe of values where $\overline{x}$ belongs in order to provide an input Boolean variable $\overline{e}_k$ to the Boolean controller. The *partitioner* is realized by a function *getE* : $E_{\mathcal{T}} \to \mathcal{D}$ that finds the discretization $\overline{e}_k \in \mathcal{D}$ that corresponds to $\overline{v}$ (see (Rodriguez and Sánchez 2023) for details). We now describe designs of *getE()*, but first we need some clarification. The set *VR* is the set of valid reactions, where each reaction is associated with a partition identified by $\overline{e}_k$. A choice, as described above is a concrete valuation of the literals to true or false. A reaction r is a conjunction of subformulae, one for each choice, stating that either the choice is possible (there are valutaions of the variables of the system that makes the choice hold) or impossible (all valuations of the variables of the system inevitably make the choice false). A reaction is *valid* (see (Rodriguez and Sánchez 2023)) whenever for some value of the environment variables, the system is left with the exact combination of possible and impossible choices described by r. That is, a valid reaction captures values of the environment for which a system can choose a specific subset of valuations of the literals in the formula. The Booleanization algorithm essentially computes the set *VR* of valid reactions, which characterizes (in a discrete manner) every decision e_k that the environment can take and every choice c_i with which the system can respond to each e_k. Thus, partitioning $\overline{x}$ is always possible in any $\mathcal{T}$ supported by the abstraction method.

A brute force design of *getE()* finds the valid reaction r that corresponds to $\overline{v}$, by substituting $\overline{x}$ in the first order formula that characterizes r and checks the validity. Note that a faster method can use quantifier elimination (QE) to compute the checker of r by evaluating $r(\overline{v})$, if $\mathcal{T}$ accepts QE, which happens in arithmetic theories (which have great applicability in the area). More technically, using QE the partitioning is pre-computed: since reactions are formulae $\exists\overline{x}.\varphi$ where φ is a conjunction of formulae $\forall\overline{y}.\varphi_i(\overline{x},\overline{y})$ and $\exists\overline{y}.\varphi_j(\overline{x},\overline{y})$. Eliminating $\overline{y}$ results in a formula $\exists\overline{x}.\varphi'$ with $\overline{x}$ as only variable in φ'. Therefore, after QE, deciding validity for a given input $\overline{v}$ for $\overline{x}$ boils down to substitution and simple evaluation. Also, when QE is not available (e.g., EUF) one has to use a (slower) validity check, to decide that $\exists\overline{x}.\varphi$ is valid (again φ contains $\forall\overline{y}$ and $\exists\overline{y}$). This method is effective but slower (need on-the-fly solving a validity query). Alg. 1 shows the brute force method. It receives the valuation $\overline{v}$, the set of valid reactions *VR* (line 1) and a list *dec_ls* of decisions e_k, associated to each $r \in$ *VR* in the same position. Then, it creates a set $\mathcal{S} =$ *VR* (line 2) that will be decremented in line 8 until it is empty (line 3). Line 4 picks a reaction $r \in \mathcal{S}$ and line 5 replaces the environment variables by $\overline{v}$. If ψ holds (line 6) then $\overline{v}$ is captured by r (and cannot be captured by any other r), so we return the associated Boolean decision of the environment (line 7). The fact that *VR* is the set of valid reactions guarantees that exactly one r captures $\overline{v}$, so line 9 is never reached in a correct design. Note that lines 3-7 describe an in-between function *getVR* : $E_{\mathcal{T}} \to$ *VR* that gets r from $\overline{v}$. Then, the partitioner will use *getE()* at each state of the execution.

Controller. Constructing the Boolean controller is straightforward: once we have the Booleanized specification, we can synthesize a Boolean controller using off-the-self synthesis tools (Jacobs et al. 2017) which exist for *LTL* and LTL_f whenever the instance is realizable.

Boolean controllers can be stimulated with environment inputs and will produce system outputs, generating plays of the form $\epsilon, \overline{e}_k^{t_0} c_i^{t_0}, \overline{e}_k^{t_0} c_i^{t_0} \overline{e}_k^{t_1} c_i^{t_1}, ...$ where the environment has selected inputs $\overline{e}_k^{t_0}, \overline{e}_k^{t_1}, ...$ and then the system reacts to them with $c_i^{t_0}, c_i^{t_1}, ...$ taking into account the current $\overline{e}_k$ and the memory encoded in the states of the controller.

Algorithm 1: Exhaustive *getE()*.

Input: $\overline{v}$, *VR*, dec_ls
$\mathcal{S} \leftarrow$ *VR*
while $\mathcal{S} \neq \emptyset$ **do**
  $r \leftarrow \mathcal{S}$, $pos \leftarrow$ *getPos*(r, VR)
  $\psi \leftarrow$ *subst*$(r, \overline{v})$
  if ψ *is satisfied* **then**
    return $\overline{e} \leftarrow dec\_ls(pos)$
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{r\}$
return *Error*

} This is *getVR*().

Provider. The discrete behaviour of the Boolean controller requires an additional component, that we call *provider* to construct a valuation $[\overline{y} \leftarrow \overline{w}]$ of the system variables that guarantee the satisfaction of the original specification. The provider receives a choice c_i by means of its combination of fresh Boolean variables of the system $s_0, s_1, ...$ that abstract the literals. Then, it translates the meaning of each s_j of c_i to $\mathcal{T}$ using the literal, creating the formula $c_i^{\mathcal{T}}$, where it substitutes the values of environment variables $\overline{x}$ for the corresponding input valuations $\overline{v}$. The resulting formula $\exists \overline{y} \in \mathcal{T}.c_i^{\mathcal{T}}$ is guaranteed to be satisfiable and it is discharged into a solver to produce a concrete model. For model provision, we can use constructive methods like the ones underlying SMT solvers, like DPLL(T) and CDCL(T).

Adaptivity in On-the-fly Synthesis

Max SMT. A key strength of the the on-the-fly approach is that it allows using additional *soft* constraints in the architecture. The queries solved by the provider can often produce different results (as long as all literals have the same valuation). Therefore, the provider can—even dynamically—produce different results. Given the first order formula $c_i^{\mathcal{T}}$, the provider may seek different models of $c_i^{\mathcal{T}}$ depending on additional constraints provided. For example, given $c_i^{\mathcal{T}} \equiv (y > x)$, then, adding soft constraint $\forall y'.(y' < y) \rightarrow (y' < x)$ to $c_i^{\mathcal{T}}$ returns the smallest model (value of y). This means that we can add (and remove) soft constraints $\varphi_{\mathcal{T}}^{add}$ on-the-fly without the need of re-synthesizing a different controller.

An important observation is that this dynamic controller will always stay within the winning region of $\varphi_{\mathcal{T}}$ because the play of the Booleanized version $\varphi_{\mathbb{B}}$ remains identical (all literals have the same valuation at every step independently of the soft constraint). For instance, a provider that is required to stay within limits $[-0.2, +0.2]$ may choose a sequence of values that follow some smoothness criteria: i.e., try to avoid abrupt changes in the values of y through time.

More use cases. The adaptivity described above is more useful whenever the architecture is running in an environment where resources are limited and it is provided with some oracle (e.g., a machine-learning model) that chooses soft constraints dynamically, depending on the optimization criteria that is decided to be followed in each timestep. These oracles can also be deterministic, like in modes of an execution. For example, consider the case in which the engineer has designed different modes to optimize energy or times, like a robot with an *econ* and *fast* speed mode or a thermostat with higher and more moderate heating modes etc.

As a more concrete use case, consider the following synthetic specification of a fast-cars road $\varphi^{car} = \square(R_A \wedge R_B)$:

$$R_A : (x < 120) \rightarrow \diamondsuit_{[0,10]}[\square(y < x)\, \mathcal{U}\, (x \geq 120)]$$
$$R_B : (x \geq 120) \rightarrow \diamondsuit_{[0,10]}[\square(y \geq x)\, \mathcal{U}\, (x < 120)],$$

where x and y are positive numeric variables that belong to the environment and the system respectively. Note that the specification φ^{car} is considerably under-specified and a controller with an unrestricted on-the-fly provider could provide y valuations that are not interesting. However, we can refine the behaviour by adding extra constraints, without needing to re-synthesize the underlying Boolean controller. For instance, consider we want to drive efficiently in case $(x \geq 120)$ requiring the minimum y such that $y \leq x$ would be the desired behaviour. Alternatively, consider some timesteps it is unsafe for the road to drive k units below x whenever $(x < 120)$ so the soft constraint would be $(y > x - k)$, where k is provided by some oracle.

Empirical Evaluation

Demonstration. We first illustrate using the running example $\varphi_{\mathcal{T}}$ how the on-the-fly approach behaves in practise. Specification $\varphi_{\mathcal{T}}$ is unrealizable for $\mathcal{T}_{\mathbb{Z}}$, but a slight modification makes it realizable. If we replace $(y < x)$ with $(y \leq x)$ we obtain $\varphi'_{\mathcal{T}} = \square(R_0 \wedge R'_1)$, where:

$$R_0 : (x < 2) \rightarrow \bigcirc(y > 1) \qquad R'_1 : (x \geq 2) \rightarrow (y \leq x)$$

Specification $\varphi'_{\mathcal{T}}$ is realizable in $\mathcal{T}_{\mathbb{Z}}$ (consider the strategy of the system to always play $y = 2$). The Booleanized version of $\varphi'_{\mathcal{T}}$ is $\varphi'_{\mathbb{B}} = \varphi'' \wedge \square([(e_0 \vee e_1) \wedge (e_0 \leftrightarrow \neg e_1)] \rightarrow \varphi^{extra\prime})$, where $\varphi'' = (s_0 \rightarrow \bigcirc s_1) \wedge (\neg s_0 \rightarrow s_2)$ and $\varphi^{extra\prime}$ is:

$$\left[e_0 \rightarrow \left(s_{01\overline{2}} \vee s_{0\overline{1}2}\right)\right] \wedge \left[e_1 \rightarrow \left(s_{\overline{0}12} \vee s_{\overline{0}1\overline{2}} \vee s_{\overline{01}2}\right)\right]$$

where $e_0, e_1 \in \mathbb{B}$ belong to the environment and represent $(x < 2)$ and $(x \geq 2)$, respectively. Note that in $\varphi'_{\mathbb{B}}$ there are no separated e_k for $(x = 2)$ and $(x > 2)$. Also, note that a complete abstraction would also produce a decision $e_{1.5}$ with choices $\{s_{01\overline{2}}, s_{0\overline{1}2}, s_{0\overline{1}\overline{2}}\}$ that represents $(x < 1)$, but an intelligent environment will never play $e_{1.5}$, since it offers strictly more power to the system (more choices), so e_0 already characterizes those plays. All choices from $s_{01\overline{2}}$

Step	x	$\overline{e}$	$\overline{c}$	$\overline{y}$
1	..., 3, 4, 5, ...	e_0, e_1	$s_{\overline{0}12}$, $s_{\overline{0}1\overline{2}}$, $s_{\overline{01}2}$	3, 2, ...
2	..., 3, 4, 5, ...	e_0, e_1	$s_{\overline{0}12}$, $s_{\overline{0}1\overline{2}}$, $s_{\overline{01}2}$	3, 2, ...
3	.., 0, 1, 2, ...	e_0, e_1	$s_{01\overline{2}}$, $s_{0\overline{1}2}$	1, 0, ...
4	.., −1, 0, 1, ...	e_0, e_1	$s_{01\overline{2}}$, $s_{0\overline{1}2}$	2, 3, ..
5	.. 2, 3, 4, . . .	e_0, e_1	$s_{\overline{0}12}$, $s_{\overline{0}1\overline{2}}$, $s_{\overline{01}2}$	2

Table 1: Modified running example $\varphi'_{\mathcal{T}}$ executed for 5 steps. Gray colour indicates the selected values among the infinitely many $\overline{x}$ and $\overline{y}$ or the finitely many $\overline{e}_k$ and c_i options.

Bn. (nm.)	Cls. (vr, lt)	*BA* Tme.	*BA* Quer.	*BA* Dc.	*BA* $\mathbb{B}$.	$1K$ sims. *Pa.*	$1K$ sims. *Cn.*	$1K$ sims. *Pr.*	$10K$ sims. *Pa.*	$10K$ sims. *Cn.*	$10K$ sims. *Pr.*	$10K$ sims. (sft.) *m/m.*	$10K$ sims. (sft.) *pc.*	$10K$ sims. (sft.) *sm.*
Li.	(1, 7)	28.66	1008	1	3.71	0.01	2.17	240	0.01	2.14	2.32	216	204	202
	(2, 4)	0.72	44	16		0.01			0.01					
	(1, 3)	0.51	30	4		0.01			0.01					
	(1, 2)	0.13	7	3		0.01			0.01					
Tr.	(1, 3)	0.87	45	5	5.01	0.01	3.41	272	0.01	3.39	262	294	270	306
	(2, 1)	0.04	2	2		0.01			0.01					
	(1, 3)	0.19	12	9		0.01			0.01					
	(1, 1)	0.10	5	4		0.01			0.01					
	(3, 6)	104.5	5367	15		0.02			0.01					
	(4, 5)	3871	52666	24		0.02			0.02					
	(3, 5)	328.4	18390	9		0.02			0.02					
	(4, 12)	4909	37083	104		0.03			0.02					
Con.	(2, 2)	0.09	4	4	4.34	0.01	1.17	104	0.01	1.17	104	107	129	102
Coo.	(3, 5)	2.60	161	1	3.56	0.01	1.21	171	0.01	1.20	168	168	168	173
Usb	(2, 3)	0.16	8	8	3.93	0.01	1.80	302	0.01	1.76	304	329	313	358
	(3, 5)	342.2	5638	32		0.01			0.01					
St.	(8, 8)	17.9	256	256	2.86	0.02	2.86	295	0.02	2.86	291	299	298	260
	(3, 6)	164.2	6138	45		0.02			0.02					
Syn.	(2, 2)	0.18	7	2	3.82	0.01	1.01	106	0.01	1.03	119	124	129	128
	(2, 3)	1.15	53	3	3.89	0.01	1.95	112	0.01	1.90	118	118	110	103
	(2, 4)	14.51	625	3	3.72	0.01	1.98	113	0.01	1.98	113	112	115	127
	(2, 5)	59.6	2707	11	3.95	0.01	2.07	170	0.01	2.06	167	144	142	152
	(2, 6)	377.7	9042	24	3.91	0.02	2.14	194	0.01	2.09	183	184	191	191
	(2, 7)	3008	12290	45	4.29	0.02	2.40	209	0.02	2.21	207	247	222	436

Table 2: Empirical evaluation of controller construction (*BA*) and execution performance (*1K,10K...*) of different benchmarks (last group refers to adaptive synthesis). All times are measured in seconds, except for *Cn.* and *Pr.*, which are microseconds.

to $s_{\overline{012}}$ have the same meaning as in $\varphi_{\mathcal{T}}$. We show a concrete execution in Tab. 1, where we see how the $\mathcal{T}$-controller responds to a few $\mathcal{T}$-inputs. For instance, in the first step, the input $x = 4$ is discretized into the Boolean decision e_1 which is passed to the Boolean controller. The controller responds $s_{\overline{012}} = \neg s_0 \wedge s_1 \wedge s_2$ to this input, which is translated into $s^{\mathcal{T}}_{\overline{012}} = \neg(x < 2) \wedge (y > 1) \wedge (y \leq x)$. Then, the provider substitutes valuation $[x \leftarrow 5]$ in $s^{\mathcal{T}}_{\overline{012}}$, and solves $\exists y.s^{\mathcal{T}}_{\overline{012}}(y)[x \leftarrow 5]$, i.e., $\exists y.\neg(5 < 2) \wedge (y > 1) \wedge (y \leq 5)$ which is guaranteed to succeed. A possible model is $y = 2$.

To show adaptivity, we could have enriched the query by adding e.g., soft $\varphi^{add} = \nexists z.(x > 2) \wedge (z > 1) \wedge (z < x) \wedge (z < y)$ that states that, there is no z that holds the same safety constraints than y and is smaller than y. In other words $s^{\mathcal{T}}_{\overline{012}} \wedge_{\text{soft}} \varphi^{add}$ implies that y is the smallest possible value that satisfies $s^{\mathcal{T}}_{\overline{012}}$. This way, the controller will always output $y = 2$, except when $s_{\overline{012}}$ is chosen. We can also design adaptivity criteria that changes over time. For instance, consider a point p which is the sum of y in previous timesteps and we want to always provide y as closest as possible to p. This way, the results in the same execution of Tab. 1 are $y = 2$, $y = 4$, $y = 1$, $y = 6$ and $y = 8$ instead of (hardly) always $y = 2$. We cannot encode such adaptivities within LTL itself (since allowing arbitrary valuation transmission across-time would make the realizability problem undecidable), but we can leverage the usage of monitors for such calculations or use linear regression if the theory is some linear arithmetic. Also, note that, since these constraints are soft, safety is not compromised. In the example, since the theory is linear arithmetic obtaining for example minimal or maximals values of bounded domains is always possible.

Moreover, we tested the performance of the realizable $\varphi'_{\mathcal{T}}$ above with the increasing p value and the same cyclic 100000 stimuli of the environment: $4, 4, 1, 0, 2, 4, 4, 1, 0, 2, ...$ The total running time of the experiment was $17079s$ and more detailed results were the following: the average time for the partitioner was $30\ ms$, the average time for the Boolean controller execution was 2.17 μs, and the average time for the provider was: 170 μs. Note that all the y outputs were $y = 2$, $y = 4$ or an increasing y, where the last value produced was $y = 499998$.

Experimental setting. We perform an empirical evaluation on six specifications based on real industrial specifications: Cooker (*Coo.*) and Usb (*Usb.*) are original benchmarks from (Rodriguez and Sánchez 2023), whereas Lift (*Li.*), Train (*Tr.*), Connect (*Con.*) and Stage (*St.*) are modifications of original specifications that were unrealizable to make them realizable. The meanings of the specifications are as follows: *Li.* describes the functioning of a freight elevator system, *Tr.* describes the functioning of an autonomous train

driving system, *Con.* describes the functioning of an electric vehicle charging and discharging system, *Coo.* describes the operation of a food processor with various functions, *Usb.* describes the operation of a system that prevents the loss of information during the interaction between a USB and a machine and *St.* describes the operation of a system that combines the use of different sensors for use in aviation. *Syn.* is a synthetic example (also from the same source) with versions from 2 to 7 literals whose aim is to test scalability.

Tab. 2 shows the main group of experiments. It is easy to see that "clusters" of literals that do not share variables can be Booleanized independently and they can be composed thereafter, so we split into clusters each of the examples, where we show number of variables (*vr.*) and literals (*lt.*) per cluster. Boolean controller synthesis time (*BA*) reports the time needed for the Boolean abstraction (*Tme.*) for each cluster, together with the number of input decisions available for the environment (*Dc.*) and the shared Boolean controller synthesis time ($\mathbb{B}$). This is performed at compile time.

The time needed for computing the components that are executed at runtime (e.g., the partitioner must be constructed from a regular expression based on the controller) is negligible. The following two groups show results of the execution of 1000 (1K sims.) and 10.000 (10K sims.) timesteps of input-output simulations. We measure, for each group: (1) the average time of the partitioner (*Pa.*) to respond with a discrete e_k from $\overline{x}$, (2) the average time of the Boolean controller (*Cn.*) to respond with a Boolean c_i and (3) the average time of the provider (*Pr.*) to respond with a $\mathcal{T}$-valuation associated to c_i. The first column corresponds to the name of the benchmarks (*nm.*). We used used Python $3.8.8$ with Z3 $4.12.2$ for the implementation. We used Strix by (Meyer, Sickert, and Luttenberger 2018) as the synthesis engine and `aigsim.c` to simulate the controller with stimulous. We ran the experiments on a MacBook Air 12.4 with the M1 processor and 16 GB of memory. All experiments can be replicated for the corresponding LTL$_f$ benchmarks, except that we would need to use another more appropriate synthesis engine for such fragment (e.g. Nike by (Favorito 2023)).

Results. Our results suggest that there is not much volatility between different simulations. This is due to the heuristics both in the Boolean controllers and the SMT solvers (which make them regularly choose the same solution paths), also the partitioner does not show volatility, as predicted. Less than 1% of the times the input-output results were different. As for hardest benchmarks, *St.* is the case study that takes the most average time for *Pa.*, since it contains more e_k (see *Dc.*) and *Tr.* is the benchmark that takes the most average time for the controller, since this controller has many more states. *Tr.* is also the benchmark that takes the most average time for the provider, since it contains more constraints to solve. Finally, differences are seem negligible when different theories $\mathcal{T}$ are used (we tested $\mathcal{T}_{\mathbb{R}}$ and $\mathcal{T}_{\mathbb{Z}}$).

In order to guarantee response time is bounded and fast enough for the targeted applications (specially in case the on-the-fly approach is to be used in safety critical contexts), we kept track of possible *time outliers*; i.e., cases in which the SMT of the provider took too much time to solve a $\mathcal{T}$-output in a given timestep. As expected, this did not happen, because current SMT solvers are able to solve instances that are much larger than the ones we used here. For example, if the Boolean abstraction receives 10 literals in a cluster, then the SMT solver will have to solve a constraint of 10 literals, which is extremely fast. It is not probable that the provider will get stuck in its task, because the current state of the art has its bottlenecks in other components (say, Boolean abstraction and classic controller synthesis), but these problems are detected before the production phase of the Boolean controller, at compile time. Note that in Tab. 2 the whole input-output process never took more than $1ms$ to respond in any given timestep. An unexpected result is that the more simulations we perform, the time for the Boolean controller is usually slightly smaller, so handling memory does not look like a problem for the controller's latches and it seems it somehow handles input repetition in an optimal way.

In addition, we used adaptivity to make systems more efficient with respect to three different criteria: (1) returning minimum and maximum valuation possible (*m/m*) knowing there was at least one of such bounds to stop the search, (2) returning valuation closest to a p point (*pc.*) that is randomly generated and (3) the smoothness valuation; i.e., providing with values as close as possible to the previous ones (*sm.*), for which we used an external method in Python to calculate values runtime. We made tests with 10000 stimuli and input-output response times remained in the order of ms. Note that the times for the partitioner and controller do not change, thus we only show the times of the providers (see the three rows in $10K$ *sims (sft.)*). There are also some unexpected cases in which looking for adaptive responses makes the provider perform faster. One possibility is that Z3's *optimize()* function for soft constraints is more regular in some cases compared to the usual *solve()* function. There is not a clear response on when one adaptive response will be faster than another. In our empirical evaluation, it strongly depended on the literals (e.g., how large the bounds are).

We also tested that, when providing y generated by ex-

Reqs.	(Wu et al.)		*Ours*			
	$\mathbb{B}$	$\mathcal{T}$	$\mathbb{B}$	$\mathcal{T}$	$\mathcal{T}^{sm}$	$\mathcal{T}^{pc}$
Pw. $26, 27$	0.3	631	0.21	171	194	198
Pw. $32, 33$	0.41	590	0.24	190	188	211
Pw. All	0.8	520	0.25	214	205	215
Driv. 1	0.45	340	0.22	106	106	105
Driv. 2	0.5	640	0.22	118	114	123
Driv. $1, , 2$	0.8	520	0.23	124	123	130
Cr. $1, 2, 3$	0.37	370	0.18	141	145	168
Cr. $2, 3, 4$	0.49	710	0.22	163	180	174
Cr. All	0.45	580	0.22	133	138	101
Quad. All	0.18	610	0.15	170	188	188
Ctr. All	0.5	690	0.26	173	185	186
Glc. All	0.31	530	0.21	177	173	169
Wtr. All	0.57	510	0.30	156	162	191

Table 3: Comparison of (Wu et al. 2019) and our approach measured in (μs), where we also test two adaptivity criteria.

ternal tools it is easy to check if a given y satisfies all literals in the choice c_i provided by the controller. This corresponds to using our technique for *LTL*$_\mathcal{T}$ shielding. Since there are no previous results in synthesis modulo theories, we adapted our experiments to make them comparable to the experiments in (Wu et al. 2019). We acknowledge that both solutions are not the same: we synthetise a controller whereas (Wu et al. 2019) synthetizes a winning region with a correction step. However, in Tab. 3 we compare the approaches in the two tasks they share: computing the Boolean output ($\mathbb{B}$) and producing the final value ($\mathcal{T}$). For the first task, our approach requires around $0.2\mu s$ and their approach needed $0.5\mu s$, which means that using a Boolean controller is a good choice. For the second task, we required approx. $0.17ms$ to provide a y whereas they needed about $0.5ms$. We conjecture that we could produce outputs faster using static techniques to synthesize a provider, in a similar way (Wu et al. 2019) uses regression. Also, we used Python for our prototype except `aigsim.c`, whereas they use all components in C. Note that (Wu et al. 2019) does not provide adaptivity results, while we do for the smoothness criteria ($\mathcal{T}^{sm}$) and moving p criteria ($\mathcal{T}^{pc}$). Also, note that (Wu et al. 2019) can only be used with specifications containing linear real arithmetic, whereas our method allows all $\exists^*\forall^*$ decidable fragments of theories (a requirement for the Boolean abstraction), which includes $\mathcal{T}_\mathbb{R}$. Hence, if we modified the $\mathcal{T}$ used in the table for $\mathcal{T}_\mathbb{Z}$, (Wu et al. 2019) could not provide responses. Also, due to its shield nature, (Wu et al. 2019) is restricted to safety, while our method produces a controller for all LTL, preserving the fragment of the original $\varphi_\mathcal{T}$.

Related Work and Conclusions

Related Work. Recently, (Rodriguez and Sánchez 2023) introduced *LTL*$_\mathcal{T}$, and showed that the realizability problem for *LTL*$_\mathcal{T}$ is decidable via a Boolean abstraction technique. We extended this approach here to reactive synthesis modulo theories. *LTL*$_f^{\text{MT}}$ is introduced in (Geatti, Gianola, and Gigante 2022) as an alternative to LTL modulo theories but for finite traces and allowing temporal operators within predicates (*lookbacks*), and only study satisfiability (which is already undecidable). We study synthesis for *LTL*$_\mathcal{T}$, which is based on realizability (Rodriguez and Sánchez 2023). Our results apply both to enriched LTL and LTL$_f$ for the restricted fragment with no lookbacks, which are decidable. Studying decidable extensions between *LTL*$_\mathcal{T}$ and LTL$^{\text{MT}}$ (reconciling the two logics) is future work.

Some works (Katis et al. 2016, 2018; Gacek et al. 2015) consider synthesis for first-order theories, but without termination guarantees and only considering some temporal fragments. We guarantee termination of the controller synthesis if the theory is decidable in the $\exists^*\forall^*$ fragment and SMT solver supports the theory. Moreover, all approaches above adapt one specific technique and implement it in a monolithic way, whereas Boolean abstraction allows to use of our on-the-fly architecture, since it generates an equi-realizable (Boolean) LTL specification. This will benefit from all future improvements of synthesis from Boolean abstractions and is fully automatic (unlike (Walker and Ryzhyk 2014)). In no previous work, adaptivity has been considered, except in (Wu et al. 2019) as compared above.

Temporal Stream Logic (TSL) (Finkbeiner et al. 2019) extends LTL with complex data that can be related accross time and (Finkbeiner, Heim, and Passing 2022; Maderbacher and Bloem 2022; Choi et al. 2022) use extensions of TSL to theories, but realizability (and thus synthesis) is undecidable in all these works. In comparison, our method cannot relate values accross time but provides a decidable synthesis procedure; e.g., all the specifications of the empirical evaluation in Tab. 2 are not within the decidable nor semi-decidable fragments of TSL. Similar undecidability is reported in (Faran and Kupferman 2018). Other approaches (e.g., (Demri and D'Souza 2007; Cheng and Lee 2013; Farzan and Kincaid 2018)) restrict expressivity whether temporal-wise, theory-wise or both. For further information on difference between LTL extensions and their expressivity compared to *LTL*$_\mathcal{T}$, we refer the reader to (Rodriguez and Sánchez 2023). Note that, (Xiao et al. 2021) uses terminology *on the fly synthesis*, but it addresses a different problem: how to construct a controller on-the-fly for Boolean LTL$_f$.

Conclusion. We have studied the problem of *LTL*$_\mathcal{T}$ synthesis which is more challenging than *LTL*$_\mathcal{T}$ realizability modulo theories, since synthesis implies computing a system that receives valuations in $\mathcal{T}$ and provides valuations in $\mathcal{T}$. We propose an *on-the-fly* approach that first discretizes the input from the environment, then uses a Boolean controller synthesized from the Booleanized specification of *LTL*$_\mathcal{T}$, and finally produces a reaction using a procedure that provides models of existential formulae of $\mathcal{T}$. We performed an empirical evaluation that proves the idea useful in practise. We also showed how adaptivity within this framework fits naturally and offers more interesting reactions. This is the first solution to reactive synthesis modulo theories.

Exploiting capabilities of the on-the-fly approach for adaptivity and semi-decidable theories is an immediate follow up. As for further future work, it still remains unclear how a static provider based in Skolem functions (Fedyukovich, Gurfinkel, and Gupta 2019) can be synthetised, and whether it will share some parts used in the on-the-fly approach: for instance, in $\varphi^\mathcal{T} = \square(y > x)$, a Skolem function $f(x) = x + 1$ serves as a witness for y in $\forall x.\exists y.\ (y > x)$ and can be used infinitely many often to provide integer values for $y \in \mathcal{T}$. However, there may be some $\exists^*\forall^*$-decidable theories for which the realizability problem is decidable, but no static provider can be constructed. We plan to compare both approaches. We also envision some applications of *LTL*$_\mathcal{T}$ to (runtime) verification and shielding.

Acknowledgments

This work was funded in part by PRODIGY Project (TED2021-132464B-I00)—funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/ PRTR—by DECO Project (PID2022-138072OB-I00)—funded by MCIN/AEI/10.13039/ 501100011033 and by the ESF+—and by a research grant from Nomadic Labs and the Tezos Foundation.

References

Cheng, C.; and Lee, E. A. 2013. Numerical LTL Synthesis for Cyber-Physical Systems. *CoRR*, abs/1307.3722.

Choi, W.; Finkbeiner, B.; Piskac, R.; and Santolucito, M. 2022. Can reactive synthesis and syntax-guided synthesis be friends? In *Proc. of the 43rd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI'22)*, 229–243. ACM.

Demri, S.; and D'Souza, D. 2007. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3): 380–415.

Faran, R.; and Kupferman, O. 2018. LTL with Arithmetic and its Applications in Reasoning about Hierarchical Systems. In *Proc. of the 22nd Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning, (LPAR-22)*, volume 57 of *EPiC Series in Computing*, 343–362. EasyChair.

Farzan, A.; and Kincaid, Z. 2018. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.*, 2(POPL): 61:1–61:30.

Favorito, M. 2023. Forward LTLf Synthesis: DPLL At Work. *CoRR*, abs/2302.13825.

Fedyukovich, G.; Gurfinkel, A.; and Gupta, A. 2019. Lazy but Effective Functional Synthesis. In *Proc. of the 20th Int'l Conf. in Verification, Model Checking, and Abstract Interpretation, (VMCAI'19)*, volume 11388 of *LNCS*, 92–113. Springer.

Finkbeiner, B. 2016. Synthesis of Reactive Systems. In *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, 72–98. IOS Press.

Finkbeiner, B.; Heim, P.; and Passing, N. 2022. Temporal Stream Logic modulo Theories. In *Proc. of the 25th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS'22)*, volume 13242 of *LNCS*, 325–346. Springer.

Finkbeiner, B.; Klein, F.; Piskac, R.; and Santolucito, M. 2019. Temporal Stream Logic: Synthesis Beyond the Bools. In *Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV'19), Part I*, volume 11561 of *LNCS*, 609–629. Springer.

Gacek, A.; Katis, A.; Whalen, M. W.; Backes, J.; and Cofer, D. D. 2015. Towards Realizability Checking of Contracts Using Theories. In *Proc. of the 7th International Symposium NASA Formal Methods (NFM'15)*, volume 9058 of *LNCS*, 173–187. Springer.

Geatti, L.; Gianola, A.; and Gigante, N. 2022. Linear Temporal Logic Modulo Theories over Finite Traces. In *Proc. of the 31st Int'l Joint Conf. on Artificial Intelligence, (IJCAI'22)*, 2641–2647. ijcai.org.

Giacomo, G. D.; and Vardi, M. Y. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proc. of the 23rd Int'l Joint Conference on Artificial Intelligence (IJCAI'13)*, 854–860. IJCAI/AAAI.

Jacobs, S.; Basset, N.; Bloem, R.; Brenguier, R.; Colange, M.; Faymonville, P.; Finkbeiner, B.; Khalimov, A.; Klein, F.; Michaud, T.; Pérez, G. A.; Raskin, J.; Sankur, O.; and Tentrup, L. 2017. The 4th Reactive Synthesis Competition (SYNTCOMP 2017): Benchmarks, Participants & Results. In *Proc. of the 6th Workshop on Synthesis (SYNT@CAV 2017)*, volume 260 of *EPTCS*, 116–143.

Katis, A.; Fedyukovich, G.; Gacek, A.; Backes, J. D.; Gurfinkel, A.; and Whalen, M. W. 2016. Synthesis from Assume-Guarantee Contracts using Skolemized Proofs of Realizability. *CoRR*, abs/1610.05867.

Katis, A.; Fedyukovich, G.; Guo, H.; Gacek, A.; Backes, J.; Gurfinkel, A.; and Whalen, M. W. 2018. Validity-Guided Synthesis of Reactive Systems from Assume-Guarantee Contracts. In *Proc. of the 24th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'18), Part II*, volume 10806 of *LNCS*, 176–193. Springer.

Maderbacher, B.; and Bloem, R. 2022. Reactive Synthesis Modulo Theories using Abstraction Refinement. In *22nd Formal Methods in Computer-Aided Design, (FMCAD'22)*, 315–324. IEEE.

Manna, Z.; and Pnueli, A. 1995. *Temporal verification of reactive systems - safety*. Springer. ISBN 978-0-387-94459-3.

Meyer, P. J.; Sickert, S.; and Luttenberger, M. 2018. Strix: Explicit Reactive Synthesis Strikes Back! In *Proc. of the 30th Int'l Conf on Computer Aided Verification (CAV'18) Part I*, volume 10981 of *LNCS*, 578–586. Springer.

Pnueli, A. 1977. The temporal logic of programs. In *Proc. of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, 46–67. IEEE CS Press.

Pnueli, A.; and Rosner, R. 1989. On the synthesis of an asynchronous reactive module. In *Proc. of the 16th Int'l Colloqium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *LNCS*, 652–671. Springer.

Rodriguez, A.; and Sánchez, C. 2023. Boolean Abstractions for Realizabilty Modulo Theories. In *Proc. of the 35th International Conference on Computer Aided Verification (CAV'23)*, volume 13966 of *LNCS*. Springer, Cham.

Thomas, W. 2008. Church's Problem and a Tour through Automata Theory. In *In Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, volume 4800 of *LNCS*, 635–655. Springer.

Walker, A.; and Ryzhyk, L. 2014. Predicate abstraction for reactive synthesis. In *Proc. of the 14th Formal Methods in Computer-Aided Design, (FMCAD'14))*, 219–226. IEEE.

Wu, M.; Wang, J.; Deshmukh, J.; and Wang, C. 2019. Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems. In *Proc. of 19th Formal Methods in Computer Aided Design, (FMCAD'19)*, 129–137. IEEE.

Xiao, S.; Li, J.; Zhu, S.; Shi, Y.; Pu, G.; and Vardi, M. Y. 2021. On-the-fly Synthesis for LTL over Finite Traces. In *Proc. of the 35th AAAI Conference on Artificial Intelligence, (AAAI'21)*, 6530–6537. AAAI Press.