On-the-fly reactive synthesis modulo theories (Work in progress)

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

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

Abstract. The Boolean abstraction technique translates (i.e., *Booleanizes*) an LTL modulo theories specification into an equi-realizable LTL specification. This solves the realizability modulo theories problem. However, synthesis modulo theories is a different problem: the system has to receive valuations in a first-order theory $\mathcal{T}$ and output valuations in $\mathcal{T}$. In this work in progress, we address how to meet this need without a pure synthesis method, but solving 'synthesis' *on-the-fly* by synthetising a Boolean controller from the Booleanized LTL specification and shipping it with a method that provides models in satisfiable instances of existential formulae (e.g., an SMT solver).

Keywords: LTL modulo theories, (Rich) reactive synthesis, (Rich) reactive realizability, Boolean abstraction, SMT solvers, Model finding, Quantifier elimination, On-the-fly symbolic computation

1 Introduction

1.1 Problem statement

Both reactive synthesis and realizability are decidable in classic LTL, but there is no decidable method for synthesis nor realizability in LTL modulo theories ($LTL_{\mathcal{T}}$): the extension of LTL where Boolean atomic propositions can be literals from a (multi-sorted) first-order theory $\mathcal{T}$. However, the Boolean abstraction method [1] translates (i.e., *Booleanizes*) specifications in reactive $LTL_{\mathcal{T}}$ into equi-realizable purely Boolean LTL ones. This solves the realizability modulo theories problem, since we can use classic off-the-shelf methods to decide realizability of Booleanized $LTL_{\mathcal{T}}$ formulae.

However, for synthesis modulo theories, it is not enough to synthetise a controller for the Booleanized specifications: we need to both receive from the environment a set of input valuations in the original $\mathcal{T}$ and the system to output another set of valuations in $\mathcal{T}$. This requires a new synthesis procedure. Nevertheless, we note that the pure correct-by-construction synthesis algorithm for $LTL_{\mathcal{T}}$ has some technical difficulties that need yet to be further researched: for instance, the synthesis method may vary substantially depending on $\mathcal{T}$ (e.g.,

with a different analytic tableaux method [2]), or some synthesis methods may be incomplete (e.g., need to compute Skolem functions from scratch [3]). Therefore, we propose an alternative general method that *mimicks* how a synthetised controller would behave by receiving and outputting valuations in $\mathcal{T}$. This is implemented using, *on-the-fly*, procedures that provide models of existential formulae of $\mathcal{T}$. Concretely, the method we propose (1) receives an $LTL_{\mathcal{T}}$ specification, (2) Booleanizes it, and makes use of (3) the controller synthetised from the Booleanized specification (4) shipped with a model-provider (e.g., an SMT solver); and both the controller and the provider are executed infinitely many often. To guarantee termination of the Booleanization process, the specification needs $\mathcal{T}$ to be decidable on its $\exists^*\forall^*$-fragment [3]. Also, once the Boolean controller is built, to guarantee that the embedded *controller + solver* architecture terminates infinitely many often (i.e., for each timestep of the potentially infinite trace), we need $\mathcal{T}$ to have a terminating procedure to provide models of existential fragments of $\mathcal{T}$. This approach does not guarantee termination using semi-decidable $\mathcal{T}$, but still yields a partial solution for such cases.

As far as we know, this is a novel method and the first successful decidable algorithm for reactive synthesis of $LTL_{\mathcal{T}}$. We explain the method and its limitations, together with a running example of it. We also raise some possible future research lines.

1.2 Boolean abstraction

We consider an approach that transforms a non-Boolean specification into a purely Boolean specification by (1) substituting theory literals by Boolean variables, and (2) computing an additional Boolean formula that captures the dependencies between the new variables imposed by the literals. This approach is called *Boolean abstraction*, and guarantees that the resulting specification can be passed to existing Boolean off-the-shelf synthesis and realizability tools, and is realizable if and only if the original specification is realizable. As for an example of reactive specifications in $\text{LTL}_{\mathcal{T}}$, let $\Box$ and $\bigcirc$ be the usual *globally* and *next* operators in LTL, respectively; and consider $\varphi_{\mathcal{T}} = \Box(R_0 \wedge R_1)$, where:

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

where its Booleanized version is $\varphi_{\mathbb{B}} = (\varphi'' \wedge (e_0 \leftrightarrow \neg e_1) \wedge (e_1 \leftrightarrow \neg e_2) \wedge (e_0 \leftrightarrow \neg e_2) \wedge \Box\varphi^{Extra})$, where $\varphi'' = (s_0 \rightarrow \bigcirc s_1) \wedge (\neg s_0 \rightarrow s_2)$ is a direct translation of $\varphi_{\mathcal{T}}$ (where s_0 models $(x < 2)$, s_1 models $(y > 1)$ and s_2 models $(y < x)$) that over-approximates the power of the system and φ^{Extra} is an additional formula

[3] This is the prenex-normal-form one-quantifier-alternation fragment in which a set of existentially quantified variables is followed by a set of universally quantified variables. This fragment includes, e.g., that all decidable theories such as linear integer arithmetic and real arithmetics

that makes $\varphi_{\mathbb{B}}$ to preserve the original power of each player in $\varphi_{\mathcal{T}}$:

$$\varphi^{Extra} : \begin{pmatrix} e_0 \rightarrow \big(c_1 \vee c_2 \vee c_3\big) \\ \quad \wedge \\ e_1 \rightarrow \big(c_5 \vee c_6\big) \\ \quad \wedge \\ e_2 \rightarrow \big(c_4 \vee c_5 \vee c_6\big) \end{pmatrix},$$

where $e_0, e_1, e_2 \in \mathbb{B}$ belong to the environment and where $c_1 = (s_0 \wedge s_1 \wedge \neg s_2)$, $c_2 = (s_0 \wedge \neg s_1 \wedge s_2)$, $c_3 = (s_0 \wedge \neg s_1 \wedge \neg s_2)$, $c_4 = (\neg s_0 \wedge s_1 \wedge s_2)$, $c_5 = (\neg s_0 \wedge s_1 \wedge \neg s_2)$ and $c_6 = (\neg s_0 \wedge \neg s_1 \wedge s_2)$, where $s_0, s_1, s_2 \in \mathbb{B}$ belong to the system. Also, the names c_1 to c_6 stand from *choice*, which means that, given a decision e_i of the environment, the system can *react* with one of the choices c_j in the disjunction implied by e_i; thus, we call *valid reactions* to the set of c_j that form the disjunction of choices, one valid reaction per e_i. Note that $e_0 \cup e_1 \cup e_2$ 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 $(x > 2)$. Also, note that *VR* is the set of valid reactions and F_r is the *reaction-formula*, i.e. the conjunction of reactions into a formula, which is needed for some calculi (more information about these can be found at [4]. It suffices for the reader to know that each valid reaction (thus, also each reaction-formula of a valid reaction) is related to its own e_k.

2 Work in progress method

2.1 Description of the architecture

Overall view For synthesis modulo theories it is not enough to synthetise a controller for the Booleanized $LTL_{\mathcal{T}}$ specifications: we need to both receive from the environment a set of input valuations in $\mathcal{T}$ and the system to output another set of valuations in $\mathcal{T}$. For instance, if the input is $x = 5$ such that $x \in \mathbb{Z}$, we cannot return a $w \in \mathbb{B}$: we need to return a concrete $y \in \mathbb{Z}$, e.g., if $\square(y > x)$ must hold, then $f(x) = x + 1$ is a suitable output that models $\forall x \exists y.\ (y > x)$, used infinitely many often. However, methods to construct such (e.g., Skolem) functions may vary depending on $\mathcal{T}$ (and may not exist for some $\exists^*\forall^*$-decidable $\mathcal{T}$), while we are interested in a more general method.

Fig. 1 shows that the *on-the-fly $LTL_{\mathcal{T}}$ specification-mimicking* method consists on shipping the Boolean controller synthetised from a $\phi_{\mathbb{B}}$ (which has been Booleanized from a $\phi_{\mathcal{T}}$) with a method that provides models in $\mathcal{T}$. At each instant of time, (1) given the valuation $[x \leftarrow v]$ of the environment (where $x, v \in \mathcal{T}$), then (2) the controller responds with a choice $c_i \in \mathbb{B}$. This choice is translated with *trans()* into its meaning in $\mathcal{T}$ in a DPLL(T) [10] fashion (e.g., in the example, if $\neg s_0 \wedge s_1 \wedge \neg s_2$ holds, then $\neg(x > 2) \wedge \neg(y > 1) \wedge (y < x)$ holds), getting $c_i^{\mathcal{T}}$; and valuation v substitutes x in $c_i^{\mathcal{T}}$. Then, (3) if a procedure (e.g., an SMT solver) uses *model()* and returns a model w of the existential formula $\varphi_m = \exists y.\ c_i^{\mathcal{T}}(y)$, then w is guaranteed to be a valuation $[y \leftarrow w]$ that

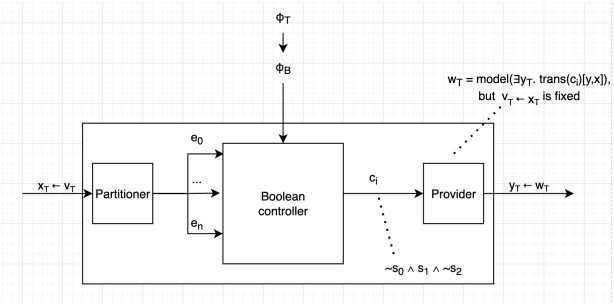

Fig. 1. The *on-the-fly* synthesis mimicking architecture.

does not violate the original $\phi_{\mathcal{T}}$ (where $y, w \in \mathcal{T}$). Note that φ_m is guaranteed to be satisfiable because of the nature of the Boolean abstraction algorithm [1]: since choices selected by the system in this step are associated to a valid (thus, satisfiable) predicate in $\mathcal{T}$.

Controller, partitioner and provider We now describe the parts of the architecture in more detail. Constructing the controller is straightforward: once we have the Booleanized specification, we can synthetise a model for it with classic tools (such as Strix or AbsSynthe). However, even if the discrete behaviour of the controller is semantically equivalent to the original one, we need the original system to output valuations in $\mathcal{T}$ (resp. environment input valuations in $\mathcal{T}$).

As for the partitioner, note the function $getVR : E_{\mathcal{T}} \rightarrow VR$, which receives a valuation of the environment variables $\overline{v}$ (the overline in $\overline{v}$ denotes one or more variable valuations) and returns the only valid reaction r for which $F_r(\overline{v})$ holds. In each timestep, the partitioner receives a valuation $\overline{v}$ of the environment variables $\overline{x}$ and its goal is to classify it in one of the discrete partitions $\overline{e}$ in order to provide an input Boolean variable $\overline{e}$ to the Boolean controller. Thus, the *partitioner* block is the infinite application of *getVR()*.

The *provider* is the function that, at the end, constructs the valuation $[\overline{y} \leftarrow \overline{w}]$ that satisfies the original specification. It is a simple technology: it receives a choice c_i (by means of its combination of fresh Boolean variables of the system $s_0, s_1, ...$), translates the meaning of these s_i into the original $\mathcal{T}$, leaving $c_i^{\mathcal{T}}$, and performs the query $\exists \overline{y} \in \mathcal{T}.c_i^{\mathcal{T}}$, which is guaranteed to be satisfiable. It could be the case, though, that obtaining the query is not fast enough for online safety critical contexts; so the current recommendation is to use this approach in offline tasks (e.g., offline monitoring of rich properties).

$\mathcal{T}$ must have a model-providing method. In the explanation above, we made a simplification for the sake of the comprehension: we assumed we had a procedure to extract models of existentially-quantified formulae of a given $\mathcal{T}$. However, even if it may seem so, not every $\exists^*\forall^*$-decidable fragment of $\mathcal{T}$

necessarily has a procedure that guarantees to return a model of existential formulae.

Thus, in order to use the on-the-fly approach with a termination guarantee, $\mathcal{T}$ must have a terminating model-provision method on existential formulae instances. Sometimes, these methods can be derived from quantifier-elimination methods, so this is also a desirable property for $\mathcal{T}$ (and also helps faster Boolean abstraction). If $\mathcal{T}$ does not meet these conditions, then there is no guarantee of finding any $\mathcal{T}$-valuation for witnessing of the Boolean strategy; yet we can try an incomplete method, such as using an SMT solver.

Note that, by construction, DPLL(T)-based SMT solving always finds a model in case of the instance being satisfiable, but there are other more challenging theories; e.g., theories that may be have been proven decidable through a quantifier elimination method, but do not have a constructive method to provide models.

2.2 Complete Running Example

If we interpret it in the theory of integers, the example of the introduction is unrealizable, so we cannot use the on-the-fly approach, because no Boolean machine is synthetisable. We still used it because we find interesting to show why it is unrealizable: it suffices for the environment to play any x such that $(x < 2)$ in the first timestep and $(x = 2)$ in the second timestep. Moreover, the environment can win in two steps no matter what the system plays. However, a slight modification in the specification turns it into realizable: we modify $(y < x)$ by $(y \leq x)$ and still $\mathcal{T} = \mathcal{T}_{\mathbb{Z}}$. Then, consider $\varphi'_{\mathcal{T}} = \square(R_0 \wedge R_1)$, where:

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

where its Booleanized version is $\varphi'_{\mathbb{B}} = (\varphi'' \wedge (e_0 \leftrightarrow \neg e_1) \wedge (e_1 \leftrightarrow \neg e_2) \wedge (e_0 \leftrightarrow \neg e_2) \wedge \square\varphi^{Extra})$, where $\varphi'' = (s_0 \rightarrow \bigcirc s_1) \wedge (\neg s_0 \rightarrow s_2)$ is, again, a direct translation of φ that over-approximates the power of the system and φ'_{Extra} is the additional formula (after some simplifications):

$$\varphi'_{Extra} : \begin{pmatrix} e_0 \rightarrow (c_1 \vee c_2) \\ \wedge \\ e_1 \rightarrow (c_4 \vee c_5 \vee c_6) \end{pmatrix},$$

where, again, $e_0, e_1, e_2 \in \mathbb{B}$ belong to the environment and where, again, $c_1 = (s_0 \wedge s_1 \wedge \neg s_2)$, $c_2 = (s_0 \wedge \neg s_1 \wedge s_2)$, $c_4 = (\neg s_0 \wedge s_1 \wedge s_2)$, $c_5 = (\neg s_0 \wedge s_1 \wedge \neg s_2)$ and $c_6 = (\neg s_0 \wedge \neg s_1 \wedge s_2)$, where $s_0, s_1, s_2 \in \mathbb{B}$ belong to the system. Note that, of the $e_0 \cup e_1 \cup e_2$ partition, where e_0 represents $(x < 2)$, e_1 represents $(x = 2)$ and e_2 represents $(x > 2)$, we did not consider e_2 for this example as an interesting case, since e_2 gives strictly more power to the system and the environment will never choose it [4]: this is because there will be no play that the environment can win using e_2 rather than e_1. Thus, we abuse notation and e_1 represents $(x \geq 2)$, but it is the particular case $x = 2$ the one that gives more power to the environment (i.e., restricts the system the most). Note that $\varphi_{\mathbb{B}}$ has the same partitions, but

with different choices implied by them: if the environment chooses e_0, then it leaves choices c_1 and c_2 to the system; if it chooses e_1, then it leaves c_4, c_5 and c_6.

We show a concrete execution (see Fig 2) and see that the $\mathcal{T}$-system always has the opportunity to respond, since the Boolean system is realizable. Note that a valid (positional) strategy of the system is to always play $y = 2$; which is equivalent to saying that a synthesised machine could have been a constant function $y = f(\overline{x}) = 2$.

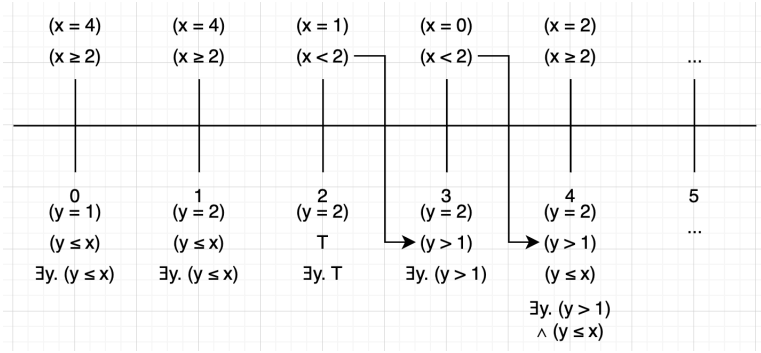

Fig. 2. Simulation of 5 steps, where we can see what values the environment chooses (and which is the predicate that this choice satisfies), together with the values that the system chooses according to the restrictions (of the current timestep or/and of the previous timestep).

Step 1: Environment forces instant response Let $x = 4$, which holds $(x \geq 2)$ and forces constraint $(y \leq x)$. We are in partition e_1, which implies choices $\{c_4, c_5, c_6\}$. Thus, the system can choose c_6 to imitate the $\overline{y} = [y = 1]$ of Fig 2. Further details are as follows:

1. $\exists \overline{y}.\ c_4(\overline{y}, \overline{x})$, with fixed $\overline{x}_{\mathcal{T}} = [x = 4]$.
   - $\mathbb{B} : \neg s_0 \wedge s_1 \wedge s_2$
   - $\mathcal{T} : (x \geq 2) \wedge (y > 1) \wedge (y \leq x)$
     $(4 \geq 2)^{\checkmark} \wedge (y > 1) \wedge (y \leq 4)$,

   where the ✓denotes that a ground predicate holds, because it is the environment decision itself.
   Thus, we have a finite set of models $m(\overline{y}) = \{2, 3, 4\}$. Take, e.g., the *least*: i.e., $\overline{y} = [y = 2]$, which holds the $(y \leq x)$ constraint of the specification. Thus, in this step, c_4 is not violating the specification; i.e., the system in $\mathbb{B}$ will potentially choose it in these plays. Note that, particularly c_4, can always be chosen.
2. $\exists \overline{y}.\ c_5(\overline{y}, \overline{x})$.

- $\mathbb{B} : \neg s_0 \wedge s_1 \wedge \neg s_2$
- $\mathcal{T} : (x \geq 2) \wedge (y > 1) \wedge (y > x)$
  $(4 \geq 2)^{\checkmark} \wedge (y > 1) \wedge (y > 4)$

Thus, we have infinite models $m(\overline{y}) = \{5, 6, 7, ...\}$. Take least, i.e., $\overline{y} = [y = 5]$, which does not hold the $(y \leq x)$ constraint of the specification. Thus, in this step, c_5 is violating the specification; i.e., the system in $\mathbb{B}$ will never choose it in these plays. Note that, indeed, c_5 can never be chosen without violating the specification.

3. $\exists \overline{y}.\ c_6(\overline{y}, \overline{x})$.
   - $\mathbb{B} : \neg s_0 \wedge \neg s_1 \wedge s_2$
   - $\mathcal{T} : (x \geq 2) \wedge (y \leq 1) \wedge (y \leq x)$
     $(4 \geq 2)^{\checkmark} \wedge (y \leq 1) \wedge (y \leq 4)$

   Thus, we have infinite models $m(\overline{y}) = \{1, 0, -1, ...\}$. Take, e.g., greatest: $\overline{y} = [y = 1]$, which holds the $(y \leq x)$ constraint of the specification. Thus, in this step, c_6 is not violating the specification; however, note that c_6 is not like c_4 in the sense that it cannot always be chosen to not violate the specification (see step 5).
   Also, note that the constraint $(y \leq 1)$ created by the Boolean abstraction algorithm is harder than the only constraint $(y \leq x)$ of the specification; thus, this suggests that the current algorithm of Boolean abstraction is *too exigent* in the sense that it considers more cases than necessary.

Step 2: Environment repeats the strategy. Environment plays the same and (see Fig 2) responds with $\overline{y} = [y = 2]$, so the system has to choose c_4 to imitate it.

Step 3: Environment changes its mind. Let $x = 1$, which holds $(x < 2)$ and forces constraint $\bigcirc(y > 1)$, whereas no constraint is further for the current timestep. We are in partition e_0, which implies choices $\{c_1, c_2\}$. The system can choose c_1 to imitate the $\overline{y} = [y = 2]$ of Fig 2 (which is the positional strategy). Note that this step has restriction $\top$ (i.e., no one), because it has forced a constraint only for the next timestep. Further details are as follows:

1. $\exists \overline{y}.\ c_1(\overline{y}, \overline{x})$, with fixed $\overline{x}_{\mathcal{T}} = [x = 1]$.
   - $\mathbb{B} : s_0 \wedge s_1 \wedge \neg s_2$
   - $\mathcal{T} : (x < 2) \wedge (y > 1) \wedge (y > x)$
     $(1 < 2)^{\checkmark} \wedge (y > 1) \wedge (y > 1)$

   Thus, we have infinite models $m(\overline{y}) = \{2, 3, 4, ...\}$. Take least, i.e., $\overline{y} = [y = 2]$, which holds the $\top$ constraint of the specification. Thus, in this step, c_1 is winning. Note that, particularly c_1, can always be chosen to hold the specification. Also, note that both $(y > 1)$ and $(y > x)$ are restrictions harder than $\top$.
2. $\exists \overline{y}.\ c_2(\overline{y}, \overline{x})$.
   - $\mathbb{B} : s_0 \wedge \neg s_1 \wedge s_2$
   - $\mathcal{T} : (x < 2) \wedge (y \leq 1) \wedge (y \leq x)$
     $(1 < 2)^{\checkmark} \wedge (y \leq 1) \wedge (y \leq 1)$

Thus, we have infinite models $m(\overline{y}) = \{1, 0, -1, ...\}$. Take greatest, i.e., $\overline{y} = [y = 3]$, which holds the $\top$ constraint of the specification. Thus, in this step, c_2 is winning; however, note that choosing c_2 (unlike c_1) does not always guarantee to hold the specification (see step 4). Again, note that both $(y \leq 1)$ and $(y \leq x)$ are restrictions harder than $\top$.

Step 4: Environment prepares its trap. Let $x = 0$, which holds $(x < 2)$ and forces constraint $\bigcirc(y > 1)$, and take into account that the system has constraint $(y > 1)$ forced by the previous timestep. We are in partition e_0, which implies, again, choices $\{c_1, c_2\}$. Further details are as follows:

1. $\exists\overline{y}.\ c_1(\overline{y}, \overline{x})$, with fixed $\overline{x}_{\mathcal{T}} = [x = 0]$.
   - $\mathbb{B} : s_0 \wedge s_1 \wedge \neg s_2$
   - $\mathcal{T} : (x < 2) \wedge (y > 1) \wedge (y > x)$
     $(0 < 2)^{\checkmark} \wedge (y > 1) \wedge (y > 0)$

   Thus, we have infinite models $m(\overline{y}) = \{2, 3, ...\}$. Take least, i.e., $\overline{y} = [y = 2]$, which holds the $\top$ constraint of the specification.
2. $\exists\overline{y}.\ c_2(\overline{y}, \overline{x})$.
   - $\mathbb{B} : s_0 \wedge \neg s_1 \wedge s_2$
   - $\mathcal{T} : (x < 2) \wedge (y \leq 1) \wedge (y \leq x)$
     $(1 < 2)^{\checkmark} \wedge (y \leq 1) \wedge (y \leq 0)$

   Thus, we have infinite models $m(\overline{y}) = \{0, -1, -2...\}$. However, no one of these holds the constraint $(y > 1)$ that was forced from previous timestep; thus, the system will never choose c_2 in this play.

Step 5: Environment strikes back! Let $x = 2$, which holds $(x \geq 2)$ and forces constraint $(y \leq x)$. Also, note that the system has constraint $(y > 1)$ from previous timestep. We are in partition e_1, which implies choices $\{c_4, c_5, c_6\}$. The system can (only) choose c_4 to imitate the $\overline{y} = [y = 1]$ of Fig 2. Further details are as follows:

1. $\exists\overline{y}.\ c_4(\overline{y}, \overline{x})$, with fixed $\overline{x}_{\mathcal{T}} = [x = 4]$.
   - $\mathbb{B} : \neg s_0 \wedge s_1 \wedge s_2$
   - $\mathcal{T} : (2 \geq 2) \wedge (y > 1) \wedge (y \leq 2)$
     $(4 \geq 2)^{\checkmark} \wedge (y > 1) \wedge (y \leq 4)$

   Thus, we have a finite model $m(\overline{y}) = \{2\}$. Take, $\overline{y} = [y = 2]$, which holds the $(y \leq x)$ constraint of the specification and also the previously forced constraint $(y > 1)$. Thus, c_4 is not violating the specification.
2. We skip $c_5^{(P)}$, since it will not be chosen, as remarked in step 1.
3. $\exists\overline{y}.\ c_6(\overline{y}, \overline{x})$.
   - $\mathbb{B} : \neg s_0 \wedge \neg s_1 \wedge s_2$
   - $\mathcal{T} : (x \geq 2) \wedge (y \leq 1) \wedge (y \leq x)$
     $(2 \geq 2)^{\checkmark} \wedge (y \leq 1) \wedge (y \leq 2)$

   Thus, we have infinite models $m(\overline{y}) = \{1, 0, -1, ...\}$. Take greatest: $\overline{y} = [y = 1]$, which holds the $(y \leq x)$ constraint of the specification. However, this model does not also hold the previously forced predicate $(y > 1)$; thus, c_6 is violating the specification in this step, and the system will not choose it.

Fig. 3 summarizes all the steps above.

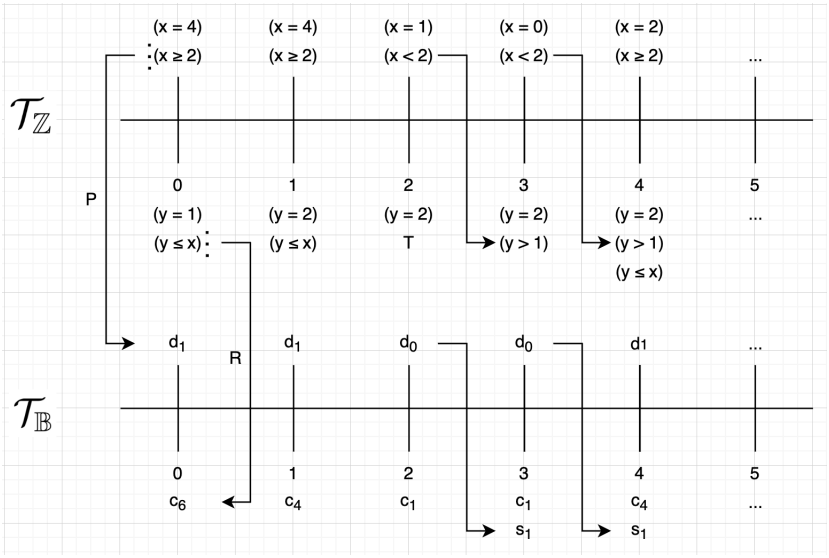

Fig. 3. We can see the $\mathcal{T}_{\mathbb{Z}}$ play above and the equi-satisfied $\mathcal{T}_{\mathbb{B}}$ play below. The P-labelled arrow between $(x \geq 2)$ and d_1 means that e_1 or e_2 is the *decision* of the environment (i.e., the partition $e_1 \cup e_2$); which represents all x such that $(x \geq 2)$ holds (resp. $(x < 2)$ with d_0). The R-labelled arrow between $(y \leq x)$ and c_6 means $\exists y.\ c_6$, where $c_6 \equiv (\neg s_0 \wedge \neg s_1 \wedge s_2)$; and, resp. $(y \leq x)$ with $\exists y.\ c_4$, $\top$ with $\exists y.\ c_1$, $(y > 1)$ with $\exists y.\ c_1$ and $((y > 1) \wedge (y \leq x))$ with $\exists y.\ c_6$. Note that, e.g., c_4 in the step 0 and c_2 in the step 2 were also possible choices of the system that the Boolean controller has decided not to play.

3 Conclusions

3.1 Why $LTL_{\mathcal{T}}$ and why Boolean abstraction?

The main difference of $LTL_{\mathcal{T}}$ with respect to other similar logics is that we do not allow predicates to compare variables at different timesteps, but we know decidability of Boolean abstraction for all theories with an $\exists^*\forall^*$-decidable fragment. For instance, [5] approaches LTL modulo theories, but for finite traces and where they allow temporal operators within predicates (making the logic undecidable).

As for Boolean abstraction, its main difference with respect to other similar approaches (e.g., [6,7,8]) is that Boolean abstraction generates a (Boolean) LTL specification so that existing tools can be used with any of their internal techniques and algorithms (bounded synthesis, for example) and will automatically benefit from further optimizations. Moreover, it preserves fragments like safety and GR(1) so specialized solvers can be used. On the contrary, all approaches above adapt one specific technique and implement it in a monolithic way. Note that the on-the-fly approach benefits from all advantages of using Boolean abstraction and is fully automatic (unlike [8]).

Note that, [9] uses terminology *on the fly synthesis*, but our work has several differences: (1) [9] is designed for LTL for finite traces, whereas our approach is for the whole LTL; (2) and [9] works for Booleans, whereas our approach works for theories. Moreover, (3) [9] addresses a very different problem: how to construct a controller on-the-fly for finite-trace LTL.

3.2 Conclusion

We acknowledge the problem of $LTL_{\mathcal{T}}$ synthesis is different from $LTL_{\mathcal{T}}$ 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 uses a Boolean controller synthetised from the Booleanized specification of $LTL_{\mathcal{T}}$ and shipped with a procedure that provides models of existential formulae of $\mathcal{T}$. The problem yields some challenges that are still to be solved, but this first approach seems promising.

Discussions When comparing the on-the-fly approach with a hypothetical pure synthesis method, it is easy to see that pure synthesis should be a faster method, since it is a set of synthetised functions and, thus, does not depend on the time that a procedure needs to provide models; whereas the on-the-fly method depends on how fast it finds models (and, if an SMT solver is used, this does not guarantee termination if theory is semi-decidable). However, this is also an advantage for the on-the-fly method, since it allows to mimic synthesis for theories that have semi-decision procedures that may provide models (e.g., by enumeration of candidate models), such as nonlinear integer arithmetic. Also, the on-the-fly approach allows using additional constraints in the architecture: i.e., we can change, on the fly, the queries that the provider must solve. For

instance, given $c_i^{\mathcal{T}}$, the provider will not solve $c_i^{\mathcal{T}}$, but the smallest model (if there is such) of $c_i^{\mathcal{T}}$; e.g., c_4 in step 1 of the running example. Pure synthesis would also allow some parameterized form of this idea, but not in such an open and *runtime* manner.

It still remains unclear how the pure synthesis will be constructed, and also whether it will include parts that are semantically equivalent to some parts used in this approach (e..g, a procedure similar to the partitioner). Moreover, we believe there may be some $\exists^*\forall^*$-decidable theories for which the realizability problem is decidable, but not the synthesis one, since it is not possible to compute the model provider (consider some fragments of reals).

As for future work, implementing a prototype of the on-the-fly approach is ongoing work, as well as delving into all discussions raised above. We also envision some applications of this approach to runtime verification (e.g., performing synthesis of runtime-monitoring specifications whose expressivity fits into $LTL_{\mathcal{T}}$). Obtaining reliable model providers is an upcoming challenge. Pure synthesis is also work in progress, but needs specific algorithms for each theory. The goal is to implement both the on-the-fly and pure synthesis approaches and compare them empirically.

References

1. A. Rodríguez, C. Sánchez, Boolean Abstractions for Realizability Modulo Theories, in: Proc. of the 35th International Conference on Computer Aided Verification (CAV 2023), Paris, France, July 17-22, 2015, Lecture Notes in Computer Science, Springer, 2023.
2. M. Hermo, P. Lucio, C. Sánchez, Tableaux for realizability of safety specifications in Proc. of the 25th International Symposium on Formal Methods (FM 2023). Lübeck, Germany, March 6-10, 2023, Proceedings, Vol. 14000 of Lecture Notes in Computer Science, Springer, 2023, pp. 495–513. URL `https://doi.org/10.1007/978-3-031-27481-7_28`
3. A. Katis, G. Fedyukovich, A. Gacek, J. D. Backes, A. Gurfinkel, M. W. Whalen, Synthesis from assume-guarantee contracts using skolemized proofs of realizability, CoRR abs/1610.05867. `arXiv:1610.05867`. URL `http://arxiv.org/abs/1610.05867`
4. A. Rodríguez, C. Sánchez, Reactive realizability modulo theories, under review (2023).
5. A. Gianola, N. Gigante, LTL modulo theories over finite traces: modeling, verification, open questions, in: L. Geatti, G. Sciavicco, A. Umbrico (Eds.), Short Paper Proceedings of the 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the 21st International Conference of the Italian Association for Artificial Intelligence (AIxIA 2022), Udine, Italy, November 28, 2022, Vol. 3311 of CEUR Workshop Proceedings, CEUR-WS.org, 2022, pp. 13–19. URL `https://ceur-ws.org/Vol-3311/paper3.pdf`
6. B. Finkbeiner, P. Heim, N. Passing, Temporal stream logic modulo theories, in: Proc. of the 25th Int'l Conf. on Foundations of Software Science and Computation Structures (FOSSACS 2022), Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7,

2022, Proceedings, Vol. 13242 of Lecture Notes in Computer Science, Springer, 2022, pp. 325–346. doi:10.1007/ 978-3-030-99253-8_17. URL https://doi.org/10.1007/978-3-030-99253-8_17

7. A. Katis, G. Fedyukovich, H. Guo, A. Gacek, J. Backes, A. Gurfinkel, M. W. Whalen, Validity-guided synthesis of reactive systems from assume- guarantee contracts, in: Proc. of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, (TACAS 2018), Held as Part of the European Joint Conferences on Theory and Prac- tice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Pro- ceedings, Part II, Vol. 10806 of Lecture Notes in Computer Science, Springer, 2018, pp. 176–193. doi:10.1007/978-3-319-89963-3_10. URL https://doi.org/10.1007/978-3-319-89963-3_10
8. A. Walker, L. Ryzhyk, Predicate abstraction for reactive synthesis, in: Proc. of the 14th Formal Methods in Computer-Aided Design, (FMCAD 2014), Lausanne, Switzerland, October 21-24, 2014, IEEE, 2014, pp. 219–226. doi: 10.1109/FMCAD.2014.6987617. URL https://doi.org/10.1109/FMCAD.2014.6987617
9. S. Xiao, J. Li, S. Zhu, Y. Shi, G. Pu, M. Y. Vardi, On-the-fly synthesis for LTL over finite traces, in: Proc. of the 35th AAAI Conference on Artificial Intelligence, (AAAI 2021), co-located with the Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021, AAAI Press, 2021, pp. 6530–6537. URL https://ojs.aaai.org/index.php/AAAI/article/view/16809
10. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In Proc. of the 16th International Conference in Computer Aided Verification, (CAV 2004), Boston, MA, USA, July 13-17, 2004, Proceedings, volume 3114 of Lecture Notes in Computer Science, pages 175–188. Springer, 2004. URL https://link.springer.com/chapter/10.1007/978-3-540-27813-9_14