Boolean Abstractions for Realizability Modulo Theories *

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

[1] IMDEA Software Institute, Madrid. Spain
[2] Universidad Politécnica de Madrid. Spain

Abstract. In this paper, we address the problem of the (reactive) realizability of specifications of theories richer than Booleans, including arithmetic theories. Our approach transforms specifications into purely Boolean specifications by (1) substituting theory literals by Boolean variables, and (2) computing an additional Boolean requirement that captures the dependencies between the new variables imposed by the literals. The resulting specification can be passed to existing Boolean off-the-shelf realizability tools, and is realizable if and only if the original specification is realizable. The first contribution is a brute-force version of our method, which requires a number of SMT queries that is doubly exponential in the number of input literals. Then, we present a faster method that exploits a nested encoding of the search for the extra requirement and uses SAT solving for faster traversing the search space and uses SMT queries internally. Another contribution is a prototype in Z3-Python. Finally, we report an empirical evaluation using specifications inspired in real industrial cases. To the best of our knowledge, this is the first method that succeeds in non-Boolean LTL realizability.

1 Introduction

Reactive synthesis [31,30] is the problem of automatically producing a system that is guaranteed to model a given temporal specification, where the Boolean variables (i.e., atomic propositions) are split into variables controlled by the environment and variables controlled by the system. Realizability is the related decision problem of deciding whether such a system exists. These problems have been widely studied [21,17], specially in the domain of Linear Temporal Logic (LTL) [29]. Realizability corresponds to infinite games where players alternatively choose the valuations of the Boolean variables they control. The winning condition is extracted from the temporal specification and determines which player wins a given play. A system is realizable if and only if the system player has a winning strategy, i.e., if there is a way to play such that the specification is satisfied in all plays played according to the strategy.

* This work was funded in part by the Madrid Regional Gov. Project "S2018/TCS-4339 (BLOQUES-CM)", by PRODIGY Project (TED2021-132464B-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union Next Generation EU/PRTR, and by a research grant from Nomadic Labs and the Tezos Foundation.

However, in practice, many practical and industrial specifications use complex data beyond Boolean atomic propositions, which precludes the direct use of realizability tools. These specifications cannot be written in (propositional) LTL, but instead use literals from a richer domain. We use $\text{LTL}_{\mathcal{T}}$ for the extension of LTL where Boolean atomic propositions can be literals from a (multi-sorted) first-order theory $\mathcal{T}$. The $\mathcal{T}$ variables (i.e., non-Boolean) in the specification are again split into those controlled by the system and those controlled by the environment. The resulting realizability problem also corresponds to infinite games, but, in this case, players chose valuations from the domains of $\mathcal{T}$, which may be infinite. Therefore, arenas may be infinite and positions may have infinitely many successors. In this paper, we present a method that transforms a specification that uses data from a theory $\mathcal{T}$ into an equi-realizable Boolean specification. The resulting specification can then be processed by an off-the-shelf realizability tool.

The main element of our method is a novel *Boolean abstraction* method, which allows to transform $\text{LTL}_{\mathcal{T}}$ specifications into (Boolean) LTL specifications. The method first substitutes all $\mathcal{T}$ literals by fresh Boolean variables controlled by the system, and then extends the specification with an additional subformula that constrains the combination values of these variables. This method is described in Section 3. The main idea is that, after the environment selects values for its (data) variables, the system responds with values for the variables it controls, which induces a Boolean value for all the literals. The additional formula we compute captures the set of possible valuations of literals and the precise power of each player to produce each valuation.

Example 1. Consider the following specification $\varphi = \Box(R_0 \wedge R_1)$, where:

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

where x is a numeric variable that belongs to the environment and y to the system. In the game corresponding to this specification, each player has an infinite number of choices at each time step. For example, in $\mathcal{T}_{\mathbb{Z}}$ (the theory of integers), the environment player chooses an integer for x and the system responds with an integer for y. This induces a valuation of all literals in the formula, which in turn induces (also considering the valuations of the literals at other time instants, according to the temporal operators) a valuation of the full specification.

In this paper, we exploit that, from the point of view of the valuations of the literals, there are only *finitely many* cases and provide a systematic manner to compute these cases. This allows us to reduce a specification into a purely Boolean specification that is equi-realizable. This specification encodes the (finite) set of decisions of the environment, and the (finite) set of reactions of the system. □

Ex. 1 suggests a naive algorithm to capture the powers of the environment and system to determine a combination of the valuations of the literals, by enumerating all these combinations and checking the validity of each potential

reaction. Checking that a given combination is a possible reaction requires an $\exists^*\forall^*$ query (which can be delegated to an SMT solver for appropriate theories).

In this paper, we describe and prove correct a Boolean abstraction method based on this idea. Then, we propose a more efficient search method for the set of possible reactions using SAT solving to speed up the exploration of the set of reactions. The main idea of this faster method is to learn from an invalid reaction which other reactions are guaranteed to be invalid, and from a valid reaction which other reactions are not worth being explored. We encode these learnt sets as a incremental SAT formula that allows to prune the search space. The resulting method is much more efficient than brute-force enumeration because, in each iteration, the learning can prune an exponential number of cases. An important technical detail is that computing the set of cases to be pruned from the outcome of a given query can be described efficiently using a SAT solver.

In summary, our contributions are: (1) a proof that realizability is decidable for all $\text{LTL}_\mathcal{T}$ specifications for those theories $\mathcal{T}$ with a decidable $\exists^*\forall^*$ fragment; (2) a simple implementation of the resulting Boolean abstraction method; (3) a much faster method based on a nested-SAT implementation of the Boolean abstraction method that efficiently explores the search space of potential reactions; and (4) an empirical evaluation of these algorithms, where our early findings suggest that Boolean abstractions can be used with specifications containing different arithmetic theories, and also with industrial specifications. We used Z3 [10] both as an SMT solver and a SAT solver, and Strix [27] as the realizability checker. To the best of our knowledge, this is the first method that succeeds (and efficiently) in non-Boolean LTL realizability.

2 Preliminaries

We study realizability of LTL [29,26] 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. The semantics of LTL associate traces $\sigma \in \Sigma^\omega$ with formulae as follows:

$\sigma \models T$	always	
$\sigma \models a$	iff	$a \in \sigma(0)$
$\sigma \models \varphi_1 \vee \varphi_2$	iff	$\sigma \models \varphi_1$ or $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$	iff	$\sigma \not\models \varphi$
$\sigma \models \bigcirc\varphi$	iff	$\sigma^1 \models \varphi$
$\sigma \models \varphi_1\, \mathcal{U}\, \varphi_2$	iff	for some $i \geq 0$ $\sigma^i \models \varphi_2$, and for all $0 \leq j < i, \sigma^j \models \varphi_1$

We use common derived operators like $\vee$, $\mathcal{R}$, $\diamondsuit$ and $\square$.

Reactive synthesis [33,28,5,14,4] is the problem of producing a system from an LTL specification, where the atomic propositions are split into propositions

that are controlled by the environment and those that are 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 according to an LTL formula φ if the trace of the play satisfies φ. A (memory-less) 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.

Depending on the fragment of LTL used, the synthesis problem has different complexities. The method that we present in this paper generates 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 formula is discharged into a solver capable to solve formulas in the right fragment. For simplicity in the presentation, we illustrate our method with safety formulae.

We use $\text{LTL}_{\mathcal{T}}$ as the extension of LTL where propositions are replaced by literals from a first-order theory $\mathcal{T}$. In realizability for $\text{LTL}_{\mathcal{T}}$, the variables that occur in the literals of a specification φ are split into those variables controlled by the environment (denoted by $\overline{v}_e$) and those controlled by the system ($\overline{v}_s$), where $\overline{v}_e \cap \overline{v}_s = \emptyset$. We use $\varphi(\overline{v}_e, \overline{v}_s)$ to remark that $\overline{v}_e \cup \overline{v}_s$ are the variables occurring in φ. The alphabet $\Sigma_{\mathcal{T}}$ is now a valuation of the variables in $\overline{v}_e \cup \overline{v}_s$. A trace is an infinite sequence of valuations, which induces an infinite sequence of Boolean values of the literals occurring in φ and, in turn, a valuation of the temporal formula.

Realizability for $\text{LTL}_{\mathcal{T}}$ 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, in Ex. 1 with $\mathcal{T} = \mathcal{T}_{\mathbb{Z}}$, valuation ranges over infinite values, and literal $(x \geq 2)$ can be satisfied with $x = 2$, $x = 3$, etc.

Arithmetic theories are a particular class of first-order theories. Even though our Boolean abstraction technique is applicable to any theory with a decidable $\exists^*\forall^*$ fragment, we illustrate our technique with 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). Both theories have a decidable $\exists^*\forall^*$ fragment. Note that the choice of theory influences the realizability of a given formula.

Example 2. Consider Ex. 1. The formula $\varphi := R_0 \wedge R_1$ is not realizable for $\mathcal{T}_{\mathbb{Z}}$, since, if at a given instant t, the environment plays $x = 0$ (and hence $x < 2$ is true), then y must be greater than 1 at time $t+1$. Then, if at $t+1$ the environment plays $x = 2$ then $(x \geq 2)$ is true but there is no y such that both $(y > 1)$ and $(y < 2)$. However, for $\mathcal{T}_{\mathbb{R}}$, φ is realizable (consider the system strategy to always play $y = 1.5$).

The following slight modifications of Ex. 1 alters its realizability (R_1' now has $y \leq x$ instead of $y < x$):

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

Now, $\varphi' = \square(R_0 \wedge R_1')$ is realizable for both $\mathcal{T}_{\mathbb{Z}}$ and $\mathcal{T}_{\mathbb{R}}$, as the strategy of the system to always pick $y = 2$ is winning. $\square$

3 Boolean Abstraction

We solve the realizability problem modulo theories by transforming the specification into an equi-realizable Boolean specification. Given a specification φ with literals l_i, we get a new specification $\varphi[l_i \leftarrow s_i] \wedge \square\varphi^{extra}$, where s_i are fresh Boolean variables and $\varphi^{extra} \in \mathrm{LTL}_{\mathbb{B}}$ is a Boolean formula (without temporal operators). The additional sub-formula φ^{extra} uses the freshly introduced variables s_i controlled by the system, as well as additional Boolean variables controlled by the environment $\overline{e}$, and captures the precise combined power of the players to decide the valuations of the literals in the original formula. We call our approach *Booleanization* or *Boolean abstraction*. The approach is summarized in Fig. 1: given an LTL specification $\varphi_{\mathcal{T}}$, it is translated into a Boolean $\varphi_{\mathbb{B}}$ which can be analyzed with off-the-shelf realizability checkers. Note that $\mathcal{G}^{\mathbb{B}}$ and $\mathcal{G}^{\mathcal{T}}$ are the games constructed from specifications $\varphi_{\mathbb{B}}$ and $\varphi_{\mathcal{T}}$, respectively. [20] shows that we can construct a game $\mathcal{G}$ from a specification φ and that φ is realizable if and only if $\mathcal{G}$ is winning for the system.

The Booleanization procedure constructs an extra requirement φ^{extra} and conjoins $\square\varphi^{extra}$ with the formula $\varphi[l_i \leftarrow s_i]$. In a nutshell, after the environment chooses a valuation of the variables it controls (including $\overline{e}$), the system responds with valuations of its variables (including s_i), which induces a Boolean value for all literals. Therefore, for each possible choice of the environment, the system has the power to choose a Boolean response among a specific collection of responses (a subset of all the possible combinations of Boolean valuations of the literals). Since the set of all possible responses is finite, so are the different cases. The extra requirement captures precisely the finite collection of choices of the environment and the resulting finite collection of responses for each case.

3.1 Notation

In order to explain the construction of the extra requirement, we introduce some preliminary definitions. We will use Ex. 1 as the running example.

A literal is an atom or its negation, no matter the atom is a Boolean variable or a predicate. Let $Lit(\varphi)$ be the collection of literals that appear in φ (or Lit, if

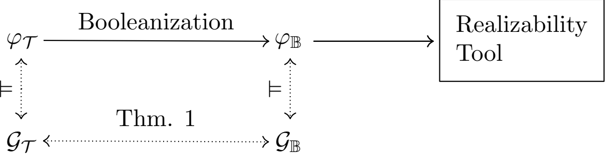

Fig. 1: The tool chain with the correctness argument.

the formula is clear from the context). For simplicity, we assume that all literals belong the same theory, but each theory can be Booleanized in turn, as each literal belongs to exactly one theory and we assume in this paper that literals from different theories do not share variables. We will use $\overline{x}$ as the environment controlled variables occurring in $Lit(\varphi)$ and $\overline{y}$ for the variables controlled by the system.

In Ex. 1, we first translate the literals in φ. Since $(x < 2)$ is equivalent to $\neg(x \geq 2)$, we use a single Boolean variable for both. The substitutions is:

$$\begin{array}{lll} (x < 2) \leftarrow s_0 & (y > 1) \leftarrow s_1 & (y < x) \leftarrow s_2 \\ (x \geq 2) \leftarrow \neg s_0 & (y \leq 1) \leftarrow \neg s_1 & (y \geq x) \leftarrow \neg s_2 \end{array}$$

After the substitution we obtain $\varphi'' = \square(R_0^{\mathbb{B}} \wedge R_1^{\mathbb{B}})$ where

$$R_0^{\mathbb{B}} : s_0 \rightarrow \bigcirc s_1 \qquad\qquad R_1^{\mathbb{B}} : \neg s_0 \rightarrow s_2$$

Note that φ'' may not be equi-realizable to φ, as we may be giving too much power to the system if s_0, s_1 and s_2 are chosen independently without restriction. Note that φ'' is realizable, for example by always choosing s_1 and s_2 to be true, but φ is not realizable in $LTL_{\mathcal{T}_{\mathbb{Z}}}$.

Definition 1 (Choice). *A choice $c \subseteq Lit(\varphi)$ is a subset of the literals of φ.*

The intended meaning of a choice is to capture what literals are true in the choice, while the rest (i.e., $Lit \setminus c$) are false. Once the environment picks values for $\overline{x}$, the system can realize some choice c by selecting $\overline{y}$ and making the literals in c true (and the rest false). However, for some values of $\overline{x}$, some choices may not be possible for the system for any $\overline{y}$. Given a choice c, we use $f(c(\overline{x}, \overline{y}))$ to denote the formula:

$$\bigwedge_{l \in c} l \wedge \bigwedge_{l \notin c} \neg l$$

which is a formula with variables $\overline{x}$ and $\overline{y}$ that captures logically the set of values of $\overline{x}$ and $\overline{y}$ that realize precisely choice c. We use $\mathcal{C}$ for the set of choices. Note that there are $|\mathcal{C}| = 2^{|Lit|}$ different choices. We call the elements of $\mathcal{C}$ choices because they may be at the disposal of the system to choose by picking the right values of its variables.

A given choice c can act as *potential* (meaning that the response is possible) or as *antipotential* (meaning that the response is not possible). A potential is a formula (that depends only on $\overline{x}$) that captures those values of $\overline{x}$ for which the system can respond and make precisely the literals in c true (and the rest of the literals false). The negation of the potential captures precisely those values of $\overline{x}$ for which there are no values of $\overline{y}$ that lead to c.

Definition 2 (Potential and Antipotential). *Given a choice c, a potential is the following formula c^p and a antipotential is the following formula c^a:*

$$c^p(\overline{x}) = \exists \overline{y}. f(c(\overline{x}, \overline{y})) \qquad\qquad c^a(\overline{x}) = \forall \overline{y}. \neg f(c(\overline{x}, \overline{y}))$$

Example 3. We illustrate two choices for Ex. 1. Consider choices $c_0 = \{(x < 2), (y > 1), (y < x)\}$ and $c_1 = \{(x < 2), (y > 1)\}$. Choice c_0 corresponds to $f(c_0) = (x < 2) \wedge (y > 1) \wedge (y < x)$, that is literals $(x < 2)$, $(y > 1)$ and $(y < x)$ are true. Choice c_1 corresponds to $f(c_1) = (x < 2) \wedge (y > 1) \wedge (y \geq x)$, that is literals $(x < 2)$, $(y > 1)$ being true and $(y < x)$ being false (i.e., $(y \geq x)$ being true). It is easy to see the meaning of c_2, c_3 etc. Then, the potential and antipotential formulae of e.g., choices c_0 and c_1 from Ex. 1 are as follows:

$$c_0^p = \exists y.(x < 2) \wedge (y > 1) \wedge (y < x) \qquad c_0^a = \forall y.\neg\big((x < 2) \wedge (y > 1) \wedge (y < x)\big)$$
$$c_1^p = \exists y.(x < 2) \wedge (y > 1) \wedge (y \geq x) \qquad c_1^a = \forall y.\neg\big((x < 2) \wedge (y > 1) \wedge (y \geq x)\big)$$

Note that potentials and antipotentials have $\overline{x}$ as the only free variables. □

Depending on the theory, the validity of potentials and antipotentials may be different. For instance, consider c_0^p and theories $\mathcal{T}_{\mathbb{Z}}$ and $\mathcal{T}_{\mathbb{R}}$:

- In $\mathcal{T}_{\mathbb{Z}}$: $\exists y.(x < 2) \wedge (y > 1) \wedge (y < x)$ is equivalent to *false*.
- In $\mathcal{T}_{\mathbb{R}}$: $\exists y.(x < 2) \wedge (y > 1) \wedge (y < x)$ is equivalent to $(x < 2)$.

These equivalences can be obtained by Cooper's algorithm [9] for $\mathcal{T}_{\mathbb{Z}}$ and Tarski's method [32] for $\mathcal{T}_{\mathbb{R}}$.

A reaction is a description of the specific choices that the system has the power to choose.

Definition 3 (Reaction). *Let P and A be a partition of $\mathcal{C}$ that is: $P \subseteq \mathcal{C}$, $A \subseteq \mathcal{C}$, $P \cap A = \emptyset$ and $P \cup A = \mathcal{C}$. The reaction $react_{(P,A)}$ is*

$$react_{(P,A)}(\overline{x}) \stackrel{def}{=} \bigwedge_{c \in P} c^p \wedge \bigwedge_{c \in A} c^a$$

The reaction $react_{(P,A)}$ is equivalent to:

$$react_{(P,A)}(\overline{x}) = \bigwedge_{c \in P} \big(\exists \overline{y}. f(c(\overline{x}, \overline{y}))\big) \wedge \bigwedge_{c \in A} \big(\forall \overline{y}. \neg f(c(\overline{x}, \overline{y}))\big).$$

There are $2^{2^{|Lit|}}$ different reactions.

A reaction r is called valid whenever there is a move of the environment for which r captures precisely the power of the system, that is exactly which choices the system can choose. Formally, a reaction is valid whenever $\exists \overline{x}. r(\overline{x})$ is a valid formula. We use $\mathcal{R}$ for the set of reactions and VR for the set of valid reactions. It is easy to see that, for all possible valuations of $\overline{x}$ the environment can pick, the system has a specific power to respond (among the finitely many cases). Therefore, the following formula is valid:

$$\varphi_{VR} = \forall \overline{x}. \bigvee_{r \in VR} r(\overline{x}).$$

Example 4. In Ex. 1, for theory $\mathcal{T}_\mathbb{Z}$, we find there are two valid reactions (using choices from Ex. 3):

$$\begin{array}{l} r_1 : \exists x. c_0^A \wedge c_1^P \wedge c_2^P \wedge c_3^P \wedge c_4^A \wedge c_5^A \wedge c_6^A \wedge c_7^A \\ r_2 : \exists x. c_0^A \wedge c_1^A \wedge c_2^A \wedge c_3^A \wedge c_4^A \wedge c_5^P \wedge c_6^P \wedge c_7^A, \end{array}$$

where reaction r_1 models the possible responses of the system after the environment picks a value for x with $(x < 2)$, whereas r_2 models the responses to $(x \geq 2)$. On the other hand, for $\mathcal{T}_\mathbb{R}$, there are three valid reactions:

$$\begin{array}{l} r_1 : \exists x. c_0^A \wedge c_1^P \wedge c_2^P \wedge c_3^P \wedge c_4^A \wedge c_5^A \wedge c_6^A \wedge c_7^A \\ r_2 : \exists x. c_0^P \wedge c_1^P \wedge c_2^P \wedge c_3^A \wedge c_4^A \wedge c_5^A \wedge c_6^A \wedge c_7^A \\ r_3 : \exists x. c_0^A \wedge c_1^A \wedge c_2^A \wedge c_3^A \wedge c_4^P \wedge c_5^P \wedge c_6^P \wedge c_7^A \end{array}$$

Note that there is one valid reaction more, since in $\mathcal{T}_\mathbb{R}$ there is one more case: $x \in (1, 2]$. Also, note that c_4 cannot be a potential in $\mathcal{T}_\mathbb{Z}$ (not even with a collaboration between environment and system), whereas it can in $\mathcal{T}_\mathbb{R}$. □

3.2 The Boolean Abstraction Algorithm

Boolean abstraction is a method to compute $\varphi_\mathbb{B}$ from $\varphi_\mathcal{T}$. In this section we describe and prove correct a basic brute-force version of this method, and later in Section 4, we present faster algorithms. All Boolean abstraction algorithms that we present on this paper first compute the extra requirement, by visiting the set of reactions and computing a subset of the valid reactions that is sufficient to preserve realizability. The three main building blocks of our algorithms are (1) the stop criteria of the search for reactions; (2) how to obtain the next reaction to consider; and (3) how to modify the current set of valid reactions (by adding new valid reactions to it) and the set of remaining reactions (by pruning the search space). Finally, after the loop, the algorithm produces as φ^{extra} a conjunction of cases, one per valid reaction (P, A) in VR. We introduce a fresh variable $e_{(P,A)}$, controlled by the environment for each valid reaction (P, A), to capture that the environment plays values for $\overline{x}$ that correspond to the case where the system is left with the power to choose captured precisely by (P, A). Therefore, there is one additional environment Boolean variable per valid reaction (in practice we can enumerate the number of valid reactions and introduce only a logarithmic number of environment variables). Finally, the extra requirement uses P for each valid reaction (P, A) to encode the potential moves of the systems as a disjunction of the literals described by each choice in P. Each of these disjunction contains precisely the combinations of literals that are possible for the concrete case that (P, A) captures.

Alg. 1: Brute-force

```
1 Input: φ_T
2 φ' ← φ_T[l_i ← s_i]  VR ← {}
3 C ← choices(literals(φ_T))
4 R ← 2^C
5 for (P, A) ∈ R do
6     if ∃x̄.react_(P,A)(x̄) then
7         VR ← VR ∪ {(P, A)}
8 φ^extra ← getExtra(VR)
9 return φ_B ← φ' ∧ φ^extra
```

A brute-force algorithm that implements Boolean Abstraction method by exhaustively searching all reactions is shown in Alg 1. The building blocks of this algorithm are:

(1) It stops when the remaining set of reactions is empty.
(2) It traverses the set $\mathcal{R}$ according to some predetermined order.
(3) To modify the set of valid reactions, if (P, A) is valid it adds (P, A) to the set VR (line 7). To modify the set of remaining reactions, it removes (P, A) from the search.

Finally, the extra sub-formula φ^{extra} is generated by *getExtra* (line 8) defined as follows:

$$getExtra(VR) = \bigwedge_{(P,A)\in VR} (e_{(P,A)} \rightarrow \bigvee_{c\in P} (\bigwedge_{l_i\in c} s_i \wedge \bigwedge_{l_i\notin c} \neg s_i))$$

Note that there is an $\exists^*\forall^*$ validity query in the body of the loop (line 6) to check whether the candidate reaction is valid. This is why decidability of the $\exists^*\forall^*$ fragment is crucial because it captures the finite partitioning of the environment moves (which is existentially quantified) for which the system can react in certain ways (i.e., potentials, which are existentially quantified) by picking appropriate valuations but not in others (i.e., antipotentials, which are universally quantified). In essence, the brute-force algorithm iterates over all the reactions, one at a time, checking whether each reaction is valid or not. In case the reaction (characterized by the set of potential choices[3]) is valid, it is added to VR.

Example 5. Consider again the specification in Ex. 1, with $\mathcal{T}_\mathbb{Z}$ as theory. Note that the valid reactions are r_1 and r_2, as shown in Ex. 4, where the potentials of r_1 are $\{c_1, c_2, c_3\}$ and the potentials of r_2 are $\{c_5, c_6\}$. Now, the creation of φ^{extra} requires two fresh variables d_0 and d_1 for the environment (they correspond to environment decisions $(x < 2)$ and $(x \geq 2)$, respectively), resulting into:

$$\varphi^{extra}_{\mathcal{T}_\mathbb{Z}} : \begin{pmatrix} d_0 \rightarrow \big((s_0 \wedge s_1 \wedge \neg s_2) \vee (s_0 \wedge \neg s_1 \wedge s_2) \vee (s_0 \wedge \neg s_1 \wedge \neg s_2)\big) \\ \wedge \\ d_1 \rightarrow \big((\neg s_0 \wedge s_1 \wedge \neg s_2) \vee (\neg s_0 \wedge \neg s_1 \wedge s_2)\big) \end{pmatrix}$$

For example $c_2 = \{s_0\}$ is a choice that appears as potential in valid reaction r_1, so it appears as a disjunct of d_0 as $(s_0 \wedge \neg s_1 \wedge \neg s_2)$. The resulting *Booleanized* specification is as follows:

$$\varphi^{\mathbb{B}}_{\mathcal{T}_\mathbb{Z}} = (\varphi'' \wedge (A_\mathbb{B} \rightarrow \Box\varphi^{extra}_{\mathcal{T}_\mathbb{Z}})) \qquad \square$$

Note that the Boolean encoding is extended with an assumption formula $A_\mathbb{B} = (d_0 \leftrightarrow \neg d_1) \wedge (d_0 \vee d_1)$ that restricts environment moves to guarantee that exactly one environment decision variable is picked. Also, note that a Boolean abstraction algorithm will output three (instead of two) decisions for the environment, but we ackowledge that one of them will never be played by it, since it gives strictly more power to the system. The complexity of this brute-force Booleanization algorithm is doubly exponential in the number of literals.

[3] The potentials in a choice characterize the precise power of the system player, because the potentials correspond with what the system can respond.

3.3 From Local Simulation to Equi-Realizability

The intuition about the correctness of the algorithm is that the extra requirement encodes precisely all reactions (i.e., collections of choices), for which there is a move of the environment that leaves the system with precisely that power to respond. As an observation, in the extra requirement, the set of potentials in valid reactions cannot be empty. This is stated in Lemma 1.

Lemma 1. *Let $C \in \mathcal{C}$ be such that $react_C \in VR$. Then $C \neq \emptyset$.*

Proof. Bear in mind $react_C \in VR$ is valid. Let $\overline{v}$ be such that $react_C[\overline{x} \leftarrow \overline{v}]$ is valid. Let $\overline{w}$ be an arbitrary valuation of $\overline{y}$ and let c be a configuration and l a literal. Therefore:

$$\bigwedge_{l[\overline{x} \leftarrow \overline{v}, \overline{y} \leftarrow \overline{w}] \text{ is true}} l \wedge \bigwedge_{l[\overline{x} \leftarrow \overline{v}, \overline{y} \leftarrow \overline{w}] \text{ is false}} \neg l$$

It follows that $I[\overline{x} \leftarrow \overline{v}]\exists \overline{y}.c$, so $c \in C$. □

Lemma 1 is crucial, because it ensures that once a Boolean abstraction algorithm is executed, for each fresh $\overline{e}$ variable in the extra requirement, at least one reaction with one or more potentials can be responded by the system.

Therefore, in each position in the realizability game, the system can respond to moves of the system leaving to precisely corresponding positions in the Boolean game. In turn, this leads to equi-realizability because each move can be simulated in the corresponding game. Concretely, is easy to see that we can define a simulation between the positions of the games for $\varphi_{\mathcal{T}}$ and $\varphi_{\mathbb{B}}$ such that (1) each literal l_i and the corresponding variable s_i have the same truth value in related positions, (2) the extra requirement is always satisfied, and (3) moves of the system in each game from related positions in each game can be mimicked in the other game. This is captured by the following theorem:

Theorem 1. *System wins $\mathcal{G}^{\mathcal{T}}$ if and only if System wins the game $\mathcal{G}^{\mathbb{B}}$. Therefore, $\varphi_{\mathcal{T}}$ is realizable if and only if $\varphi_{\mathbb{B}}$ is realizable.*

Proof. Since realizability games are memory-less determined, it is sufficient to consider only local strategies. Given a strategy $\rho_{\mathbb{B}}$ that is winning in $\mathcal{G}^{\mathbb{B}}$ we define a strategy $\rho_{\mathcal{T}}$ in $\mathcal{G}^{\mathcal{T}}$ as follows. Assuming related positions, $\rho_{\mathcal{T}}$ moves in $\mathcal{G}^{\mathcal{T}}$ to the successor that is related to the position where $\rho_{\mathbb{B}}$ moves in $\mathcal{G}^{\mathbb{B}}$. By (3) above, it follows that for every play played in $\mathcal{G}^{\mathbb{B}}$ according to $\rho_{\mathbb{B}}$ there is a play in $\mathcal{G}^{\mathcal{T}}$ played according to $\rho_{\mathcal{T}}$ that results in the same trace, and vice-versa: for every play played in $\mathcal{G}^{\mathcal{T}}$ according to $\rho_{\mathcal{T}}$ there is a play in $\mathcal{G}^{\mathbb{B}}$ played according to $\rho_{\mathbb{B}}$ that results in the same trace. Since $\rho_{\mathbb{B}}$ is winning, so is $\rho_{\mathcal{T}}$. The other direction follows similarly, because again $\rho_{\mathbb{B}}$ can be constructed from $\rho_{\mathcal{T}}$ not only guaranteeing the same valuation of literals and corresponding variables, but also that the extra requirement holds in the resulting position. □

The following corollary of Thm. 1 follows immediately.

Theorem 2. *Let $\mathcal{T}$ be a theory with a decidable $\exists^*\forall^*$-fragment. Then,* $\text{LTL}_{\mathcal{T}}$ *realizability is decidable.*

4 Efficient algorithms for Boolean Abstraction

4.1 Quasi-reactions

The basic algorithm presented in Section 3 exhaustively traverses the set of reactions, one at a time, checking whether each reaction is valid. Therefore the body of the loop is visited $2^{|\mathcal{C}|}$ times. In practice, the running time of this basic algorithm quickly becomes unfeasible.

We now improve Alg. 1 by exploiting the observation that every SMT query for the validity of a reaction reveals information about the validity of other reactions. We will exploit this idea by learning uninteresting subsequent sets of reactions and pruning the search space. The faster algorithms that we present below encode the remaining search space using a SAT formula, whose models are further reactions to explore.

To implement the learning-and-pruning idea we first introduce the notion of quasi-reaction.

Definition 4 (Quasi-reaction). *A quasi-reaction is a pair* (P, A) *where* $P \subseteq \mathcal{C}$, $A \subseteq \mathcal{C}$ *and* $P \cap A = \emptyset$.

Quasi-reactions remove from reactions the constraint that $P \cup A = \mathcal{C}$. A quasi-reaction represents the set of reactions that would be obtained from choosing the remaining choices that are neither in P nor in A as either potential or antipotential. The set of quasi-reactions is:

$$\mathcal{Q} = \{(P, A) | P, A \subseteq \mathcal{C} \textit{ and } P \cap A = \emptyset\}$$

Note that $\mathcal{R} = \{(P, A) \in \mathcal{Q} | P \cup A = \mathcal{C}\}$.

Example 6. Consider a case with four choices c_0, c_1, c_2 and c_3. The quasi-reaction $(\{c_0, c_2\}, \{c_1\})$ corresponds to the following formula:

$$\exists \overline{x}.\ \big(\exists \overline{y}.\ f(c_0(\overline{x}, \overline{y})) \wedge \forall \overline{y}.\ \neg f(c_1(\overline{x}, \overline{y})) \wedge \exists \overline{y}.\ f(c_2(\overline{x}, \overline{y}))\big)$$

Note that nothing is stated in this quasi-reaction about c_3 (it neither acts as a potential nor as an antipotential). □

Consider the following order between quasi-reactions: $(P, A) \preceq (P', A')$ holds if and only if $P \subseteq P'$ and $A \subseteq A'$. It is easy to see that $\preceq$ is a partial order, that $(\emptyset, \emptyset)$ is the lowest element and that for every two elements (P, A) and (P', A') there is a greatest lower bound (namely $(P \cap P', A \cap A')$). Therefore $(P, A) \sqcap (P', A') \stackrel{\text{def}}{=} (P \cap P', A \cap A')$ is a meet operation (it is associative, commutative and idempotent). Note that $q \preceq q'$ if and only if $q \sqcap q' = q$. Formally:

Proposition 1. $(\mathcal{Q}, \sqcap)$ *is a lower semi-lattice.*

The quasi-reaction semi-lattice represents how *informative* a quasi-reaction is. Given a quasi-reaction (P, A), removing an element from either P or A results in a strictly less informative quasi-reaction. The lowest element $(\emptyset, \emptyset)$ contains the least information.

Given a quasi-reaction q, the set $\mathcal{Q}_q = \{q' \in \mathcal{Q} | q' \preceq q\}$ of the quasi-reactions below q form a full lattice with join $(P, Q) \sqcup (P', Q') \stackrel{\text{def}}{=} (P \cup P', Q \cup Q')$. This is well defined because P' and Q, and P and Q' are guaranteed to be disjoint.

Proposition 2. *For every q, $(\mathcal{Q}_q, \sqcap, \sqcup)$ is a lattice.*

As for reactions, quasi-reactions correspond to a formula in the theory as follows:

$$qreact_{(P,A)}(\overline{x}) = \bigwedge_{c \in P} \big(\exists \overline{y}. c(\overline{x}, \overline{y})\big) \wedge \bigwedge_{c \in A} \big(\forall \overline{y}. \neg c(\overline{x}, \overline{y})\big)$$

Again, given a quasi-reaction q, if $\exists \overline{x}. qreact_q(\overline{x})$ is valid we say that q is valid, otherwise we say that q is invalid. The following holds directly from the definition (and the fact that adding conjuncts makes a first-order formula "less satisfiable").

Proposition 3. *Let q, q' be two quasi-reactions with $q \preceq q'$. If q is invalid then q' is invalid. If q' is valid then q is valid.*

These results enable the following optimizations.

4.2 Quasi-reaction-based Optimizations

A Logic-based Optimization. Consider that, during the search for valid reactions in the main loop, a reaction (P, A) is found to be invalid, that is $react_{(P,A)}$ is unsatisfiable. If the algorithms explores the quasi-reactions below (P, A), finding $(P', A') \preceq (P, A)$ such that $qreact_{(P',A')}$, then by Prop. 3, every reaction (P'', A'') above (P', A') is guaranteed to be invalid. This allows to prune the search in the main loop by computing a more informative quasi-reaction q after an invalid reaction r is found, and skipping all reactions above q (and not only r). For example, if the reaction corresponding to $(\{c_0, c_2, c_3\}, \{c_1\})$ is found to be invalid, and by exploring quasi-reactions below it, we find that $(\{c_0\}, \{c_1\})$ is also invalid, then we can skip all reactions above $(\{c_0\}, \{c_1\})$. This includes for example $(\{c_0, c_2\}, \{c_1, c_3\})$ and $(\{c_0, c_3\}, \{c_1, c_2\})$. In general, the lower the invalid quasi-reaction in $\preceq$, the more reactions will be pruned. This optimization resembles a standard choosing of max/min elements in an anti-chain.

A Game-based Optimization. Consider now two reactions $r = (P, A)$ and $r' = (P', A')$ such that $P \subseteq P'$ and assume that both are valid reactions. Since r' allows more choices to the system (because the potentials P determine these choices), the environment player will always prefer to play r than r'. Formally, if there is a winning strategy for the environment that chooses values for $\overline{x}$ (corresponding to a model of $react_r$), then choosing values for $\overline{x}'$ instead (corresponding to a model of $react_{r'}$) will also be winning.

Therefore, if a reaction r is found to be valid, we can prune the search for reactions r' that contain strictly more potentials, because even if r' is also valid, it will be less interesting for the environment player. For instance, if $(\{c_0, c_3\}, \{c_1, c_2\})$ is valid, then $(\{c_0, c_1, c_3\}, \{c_2\})$ and $(\{c_0, c_1, c_3, c_2\}, \{\})$ become uninteresting to be explored and can be pruned from the search.

4.3 A Single Model-loop Algorithm (Alg. 2)

We present now a faster algorithm that replaces the main loop of Alg. 1 that performs exhaustive exploration with a SAT-based search procedure that prunes uninteresting reactions. In order to do so, we use a SAT formula ψ with one variable z_i per choice c_i, in a DPLL(T) fashion. An assignment $v : \mathit{Vars}(\psi) \to \mathbb{B}$ to these variables represents a reaction (P, A) where

$$P = \{c_i | v(z_i) = \mathit{true}\} \qquad A = \{c_j | v(z_j) = \mathit{false}\}$$

Similarly, a partial assignment $v : \mathit{Vars}(\psi) \rightharpoonup \mathbb{B}$ represents a quasi-reaction. The intended meaning of ψ is that its models encode the set of interesting reactions that remain to be explored. This formula is initialized with $\psi = \mathit{true}$ (note that $\neg(\bigwedge_{z_i} \neg z_i)$ is also a correct starting point because the reaction where all choices are antipotentials is invalid). Then, a SAT query is used to find a satisfying assignment for ψ, which corresponds to a (quasi-)reaction r whose validity is interesting to be explored. Alg. 2 shows the Model-loop algorithm. The three main building blocks of the model-loop algorithm are:

(1) Alg. 2 stops when ψ is invalid (line 14).
(2) To explore a new reaction, Alg. 2 obtains a satisfying assignment for ψ (line 15).
(3) Alg. 2 checks the validity of the reaction (line 16) and enriches ψ o prune according to what can be learned, as follows:
   - If the reaction is invalid (as a result of the SMT query in line 16), then it checks the validity of quasi-reaction $q = (\emptyset, A)$ in line 23. If q is invalid, add the negation of q as a new conjunction of ψ (line 26). If q is valid, add the negation of the reaction (line 24). This prevents all SAT models that agree with one of these q, which correspond to reactions $q \preceq r'$, including r.

Alg. 2: Model-loop

```
10 Input: φ_T
11 φ' ← φ_T[l_i ← s_i] ; VR ← {}
12 C ← choices(literals(φ_T))
13 R ← 2^C ; ψ ← ⊤
14 while SAT(ψ) do
15     m = model(ψ)
16     if ∃x̄. (toTheory(m, C))
       then
17         P ← posVars(m)
18         ψ ← ψ ∧ ¬(⋀_{p∈P} p)
19         VR ← VR ∪ (e_t, P)
20     else
21         N ← negVars(m)
22         fh ← ⋀_{n∈N} n
23         if ∃x̄. toTheory(fh, C)
           then
24             ψ ← ψ ∧ ¬m
25         else
26             ψ ← ψ ∧ ¬fh
27 φ^extra ← getExtra(VR)
28 return φ_B ← φ' ∧ φ^extra
```

– If the reaction is valid, then it is added to the set of valid reactions VR and the corresponding quasi-reaction that results from removing the antipotentials is added (negated) to ψ (line 18), preventing the exploration of uninteresting cases, according to the game-based optimization.

As for the notation in Alg. 2 (also in Alg. 3 and Alg. 4), *model(ψ)* in line 15 is a function that returns a satisfying assignment of the SAT formula ψ, *posVars(m)* returns the positive variables of m (e.g., c_i, c_j etc.) and *negVars(m)* returns the negative variables. Finally, $\mathit{toTheory}(m, \mathcal{C}) = \bigwedge_{m_i} c_i^p \wedge \bigwedge_{\neg m_i} c_i^a$ (in lines 16 and 23) translates a Boolean formula into its corresponding formula in the given $\mathcal{T}$ theory. Note that unsatisfiable m can be minimized finding cores.

If r is invalid and $(\emptyset, A)$ is found also to be invalid, then exponentially many cases can be pruned. Similarly, if r is valid, also exponentially many cases can be pruned. The following result shows the correctness of Alg. 2:

Theorem 3. *Alg. 2 terminates and outputs a correct Boolean abstraction.*

Proof. (sketch) Alg. 2 terminates because, at each step in the loop, ψ removes at least one satisfying assignment and the total number is bounded by $2^{|\mathcal{C}|}$. Also, the correctness of the generated formula is guaranteed because, for every valid reaction in Alg. 1, either there is a valid reaction found in Alg. 2 or a more promising reaction found in Alg. 2. □

4.4 A Nested SAT algorithm (Alg. 3)

We now present an improvement of Alg. 2 that performs a more detailed search for a promising collection of invalid quasi-reactions under an invalid reaction r.

Alg. 3: Nested-SAT

```
29 Input: φ_T
30 φ' ← φ_T[l_i ← s_i] ; VR ← {}
31 C ← choices(literals(φ_T))
32 R ← 2^C ; ψ ← ⊤
33 while SAT(ψ) do
34     m = model(ψ)
35     if ∃x̄. (toTheory(m, C))
        then
36         P ← posVars(m)
37         ψ ← ψ ∧ ¬(⋀_{p∈P} P)
38         VR ← VR ∪ (e_t, P)
39     else
40         N ← negVars(m)
41         ψ ← ψ ∧ ¬m
42         I ← inner_loop(m, C)
43         ψ ← ψ ∧ ¬(⋀_{i∈I} i)
44 φ^extra ← getExtra(VR)
45 return φ_B ← φ' ∧ φ^extra
```

Note that it is not necessary to find the precise collection of all the smallest quasi-reactions that are under an invalid reaction r, as long as at least one quasi-reaction under r is calculated (perhaps, r itself). Finding lower quasi-reactions allow to prune more, but its calculation is more costly, because more SMT queries need to be performed. The Nested-SAT algorithm (Alg. 3) explores (using an inner SAT encoding) this trade-off between computing more exhaustively better invalid quasi-reactions and the cost of the search. The three main building blocks of the nested-SAT algorithm (see Alg. 3) are:

(1) It stops when ψ is invalid (as in Alg. 2), in line 33.
(2) To get the reaction, obtain a satisfying assignment m for ψ (as in Alg. 2), in line 34.

(3) Check the validity of the corresponding reaction and prune ψ according to what can be learned as follows. If the reaction is valid, then we proceed as in Alg. 2. If $r = (P, A)$ is invalid (as a result of the SMT query), then an inner SAT formula encodes whether a choice is masked (eliminated from P or A). Models of the inner SAT formula, therefore, correspond to quasi-reactions below r. If a quasi-reaction q found in the inner loop is invalid, the inner formula is additionally constrained and the set of invalid quasi-reactions is expanded. If a quasi-reaction q found is valid, then the inner SAT formula is pruned eliminating all quasi-reactions that are guaranteed to be valid. At the end of the inner loop, a (non-empty) collection of invalid quasi-reactions are added to ψ.

The inner loop, shown in Alg. 4 (where VQ stands for *valid quasi-reactions*), explores a full lattice. Also, note that $\neg(\bigwedge_{z_i} \neg z_i)$ is, again, a correct starting point. Consider, for example, that the outer loop finds $(\{c_1, c_3\}, \{c_0, c_2\})$ to be invalid and that the inner loop produces assignment $w_0 \wedge w_1 \wedge w_2 \wedge \neg w_3$. This corresponds to c_3 being masked producing quasi-reaction $(\{c_1\}, \{c_0, c_2\})$. The pruning system is the following:

Alg. 4: Inner loop

```
46 Input: m, C
47 VQ ← {} ; β ← ⊤
48 while SAT(β) do
49     u = model(β)
50     if
         ∃x̄. (toTheory_inn(u, m, C))
       then
51         P ← posVars(u)
52         β ← β ∧ ¬(⋀_{p∈P} p)
53     else
54         N ← negVars(u)
55         β ← β ∧ ¬(⋀_{n∈N} n)
56         VQ ← VQ ∪ u
57 return VQ
```

- If quasi-reaction q is valid then the inner SAT formula is pruned eliminating all inner models that agree with the model in the masked choices. In our example, we would prune all models that satisfy $\neg w_3$ if q is valid (because the resulting quasi-reactions will be inevitably valid).
- If quasi-reaction q is invalid, then we prune in the inner search all quasi-reactions that mask less than q, because these will be inevitably invalid. In our example, we would prune all models satisfying $\neg(w_0 \wedge w_1 \wedge w_2)$.

Note that $toTheory_inn(u, m, \mathcal{C}) = \bigwedge_{m_i \wedge u_j} c_i^p \wedge \bigwedge_{\neg m_i \wedge u_j} c_i^a$ is not the same function as the *toTheory()* used in Alg. 2 and Alg. 3, since the inner loops needs both model m and mask u (which makes no sense to be negated) to translate a Boolean formula into a $\mathcal{T}$-formula. Also, note that there is again a trade-off in the inner loop because an exhaustive search is not necessary. Thus, in practice, we also used some basic heuristics: (1) entering the inner loop only when $(\emptyset, A)$ is invalid; (2) fixing a maximum number of inner model queries per outer model with the possibility to decrement this amount dynamically with a decay; and (3) reducing the number of times the inner loop is exercised (e.g., *enter the inner loop only if the number of invalid outer models so far is even*).

Example 7. We explore the results of Alg. 3. A possible execution for 2 literals can be as follows:

1. Reaction $(\{c_0, c_3\}, \{c_1, c_2\})$ is obtained in line 34, which is declared invalid by the SMT solver in line 35. The inner loop called in line 42 produces $(\{c_0\}, \{c_1\})$, $(\{c_3\}, \{c_2\})$ and $(\{\}, \{c_1, c_2\})$ as three invalid quasi-reactions, and their negations are added to the SAT formula of the outer loop in line 43.
2. A second reaction $(\{c_0, c_1\}, \{c_3, c_4\})$ is obtained from the SAT solver in line 34, and now the SMT solver query is valid in line 35. Then, $\neg(c_0 \wedge c_1)$ is added to the outer SAT formula in line 37.
3. A third reaction $(\{c_2, c_3\}, \{c_0, c_1\})$ is obtained in line 33 , which is again valid in line 35. Similarly, $\neg(c_2 \wedge c_3)$ is added the outer SAT formula in line 37.
4. A fourth reaction $(\{c_1, c_2\}, \{c_0, c_3\})$ is obtained in line 33, which is now invalid (line 35). The inner loop called in line 42 generates the following cores: $(\{c_1\}, \{c_0\})$ and $(\{c_2\}, \{c_3\})$. The addition of the negation of these cores leads to an unsatisfiable outer SAT formula, and the algorithm terminates.

The execution in this example has performed 4 SAT+SMT queries in the outer loop, and 3+2 SAT+SMT queries in the inner loops. The brute-force Alg. 1 would have performed 16 queries. Note that the difference between the exhaustive version and the optimisations soon increases exponentially when we consider specifications with more literals.

□

5 Empirical evaluation

We perform an empirical evaluation on six specifications inspired by real industrial cases: *Lift* (*Li.*), *Train* (*Tr.*), *Connect* (*Con.*), *Cooker* (*Coo.*), *Usb* (*Usb*) and *Stage* (*St.*), and a synthetic example (*Syn.*) with versions from 2 to 7 literals. For the implementation, we used used Python 3.8.8 with Z3 4.11.

It is easy to see that "clusters" of literals that do not share variables can be Booleanized independently, so we split into clusters each of the examples. We report our results in Fig. 2. Each row contains the result for a cluster of an experiment (each one for the fastest heuristic). Each benchmark is split into clusters, where we show number of variables (*vr.*) and literals (*lt.*) per cluster. We also show running times of each algorithm against each cluster; concretely, we test Alg. 1 (*BF*), Alg. 2 (*SAT*) and Alg. 3 (*Doub.*). For Alg. 2 and Alg. 3, we show the number of queries performed; in the case of Alg. 3, we also show both outer and inner queries. Alg. 1 and Alg. 2 require no heuristics. For Alg. 3, we report, left to right: maximum number of inner loops (*MxI.*), the modulo division criteria (*Md.*)[4], the number of queries after which we perform a decay of 1 in the maximum number of inner loops (*Dc.*), and if we apply the invalidity of $(\emptyset, A)$ as a criteria to enter the inner loop (*A.*), where ✓ means that we do and × means the contrary. Also, ⊥ means timeout (or *no data*).

[4] This means that the inner loop is entered if and only if the number of invalid models so far is divisible by *Md*.

Bn.	Cls.	Time (s)			Queries (out+inn)		Heuristics (doub)				$\varphi^{\mathbb{B}}$	
(nm.)	(vr, lt)	BF	SAT	Doub.	SAT	Doub.	MxI.	Md.	Dc.	*A.*	Val.	Tme.
Li.	(1, 7)	⊥	6740	**31.77**	30375	**72/1040**	40	2	0	✓	1	4.41
	(2, 4)	3911	**0.70**	0.91	**27**	25/20	10	2	0	×	16	
	(1, 3)	3.64	1.19	**0.52**	46	**10/20**	10	2	0	×	4	
	(1, 2)	0.23	**0.09**	0.14	4	**4/3**	3	3	0	×	3	
Tr.	(1, 3)	3.18	**0.04**	0.96	**16**	26/20	10	2	0	✓	5	5.13
	(2, 1)	0.05	**0.04**	**0.04**	**2**	**2/0**	1	1	0	✓	2	
	(1, 3)	3.10	1.64	**0.21**	74	**2/10**	10	2	0	✓	1	
	(1, 1)	**0.04**	0.06	0.11	**3**	3/2	1	1	0	✓	1	
	(3, 6)	⊥	1269	**112.5**	13706	**1170/4716**	100	20	40	×	15	
	(4, 5)	⊥	5251	**4144**	**44177**	52623/12332	100	20	40	×	24	
	(3, 5)	⊥	2044	**359.3**	31363	**9123/10158**	100	20	40	×	9	
	(4, 12)	⊥	⊥	**6571**	⊥	**2728/40920**	100	20	40	×	104	
Con.	(2, 2)	0.23	**0.09**	**0.09**	**4**	**4/0**	3	3	0	✓	4	4.37
Coo.	(3, 5)	⊥	1356	**2.81**	27883	**16/160**	20	2	0	✓	1	3.64
Usb.	(2, 3)	3.40	0.21	**0.17**	**8**	**8/0**	3	3	0	✓	8	3.93
	(3, 5)	⊥	**231.9**	364.4	**5638**	**5638/0**	20	2	0	✓	32	
St.	(8, 8)	⊥	**18.19**	18.20	**256**	**256/0**	40	2	0	✓	256	6.06
	(3, 6)	⊥	1311	**194.8**	14994	**1697/6536**	100	20	40	×	45	
Syn.	(2, 2)	0.21	0.24	**0.18**	11	**4/3**	3	3	0	✓	2	4.12
	(2, 3)	3.42	2.69	**1.24**	119	**14/40**	10	2	0	✓	3	4.11
	(2, 4)	2842	108.6	**16.51**	3982	**188/620**	10	2	0	✓	3	4.28
	(2, 5)	⊥	7151	**68.90**	44259	**380/2800**	20	2	0	✓	11	4.53
	(2, 6)	⊥	⊥	**402.2**	⊥	**4792/9941**	100	20	40	×	24	4.85
	(2, 7)	⊥	⊥	**3596**	⊥	**7344/139440**	40	2	0	✓	1	5.30
	(2, 7)	⊥	⊥	**3862**	⊥	**24311/40615**	200	20	40	×	45	5.99

Fig. 2: Empirical evaluation results of the different Boolean abstraction algorithms, where the best results are in **bold** and $\varphi_{\mathbb{B}}$ only refers to best times.

The brute-force (BF) Alg. 1 performs well with 3 or fewer literals, but the performance dramatically decreases with 4 literals. Alg. 2 (single SAT) performs well up to 4 literals, and it can hardly handle cases with 6 or more literals. An exception is *Lift (1,7)* which is simpler since it has only one variable (and this implies that there is only one player). The performance improvement of SAT with respect to BF is due to the decreasing of queries. For example, *Train (3,6)* performs 13706 queries, whereas BF would need $2^{2^6} = 1.844 \cdot 10^{18}$ queries.

All examples are Booleanizable when using Alg. 3 (two SAT loops), particularly when using a combination of concrete heuristics. For instance, in small cases (2 to 5 literals) it seems that heuristic-setups like 3/3/3/0/✓[5] are fast, whereas in bigger cases other setups like 40/2/0/✓ or 100/40/20/× are faster.

[5] This means: we only perform 3 inner loop queries per outer loop query (and there is no decay, i.e., $decay = 0$), we enter the inner loop once per 3 outer loops and we only enter the inner loop if $(\emptyset, A)$ is invalid.

Lits	Alg.	Performed queries (out+inn)	Out of	Needed queries ($\simeq$ %)
2	Alg 2	4	16	25
3	Alg 2	8	256	3.125
4	Alg 3	$83 + 380$	65536	0.709
5	Alg 3	$380 + 2800$	4294967296	$7.404 \cdot 10^{-5}$
6	Alg 3	$4792 + 9941$	$1.844 \cdot 10^{19}$	$1 \cdot 10^{-13}$
...	...	...	...	...
12	Alg 3	$2728 + 40920$	∞	0

Fig. 3: Best numbers of queries for Alg. 2 and 3 relative to brute-force (Alg.1).

We conjecture that a non-zero decay is required to handle large inputs, since inner loop exploration becomes less useful after some time. However, adding a decay is not always faster than fixing a number of inner loops (see *Syn (2,7)*), but it always yields better results in balancing the number of queries between the two nested SAT layers. Thus, since balancing the number of queries typically leads to faster execution times, we recommend to use decays. Note that we performed all the experiments reported in this section running all cases several times and computing averages, because Z3 exhibited a big volatility in the models it produces, which in turn influenced the running time of our algorithms. This significantly affects the precise reproducibility of the running times. For instance, for *Syn(2,5)* the worst case execution was almost three times worst than the average execution reported in Fig. 2. Studying this phenomena more closely is work in progress. Note that there are cases in which the number of queries of *SAT* and *Doub.* are the same (e.g., *Usb(3,5)*), which happened when the *A.* heuristic had the effect of making the search not to enter the inner loop.

In Fig. 2 we also analyzed the constructed $\varphi_{\mathbb{B}}$, measuring the number of valid reactions from which it is made (*Val.*) and the time (*Tme.*) that a realizability checker takes to verify whether $\varphi_{\mathbb{B}}$ (hence, $\varphi_{\mathcal{T}}$) is realizable or not (expressed with dark and light gray colours, respectively). We used Strix [27] as the realizability process. As we can see, there is a correspondence between the expected realizability in $\varphi_{\mathcal{T}}$ and the realizability result that Strix returns in $\varphi_{\mathbb{B}}$. Indeed, we can see all instances can be solved in less than 7 seconds, and the length of the Boolean formula (characterized by the number of valid reactions) hardly affect performance. This suggests that future work should be focused on reducing time necessary to produce Boolean abstraction to scale even further.

Also, note that Fig. 2 shows remarkable results as for ratios of queries required with respect to the (doubly exponential) brute-force algorithm: e.g., $4792 + 9941$ (outer + inner loops) out of the $1.844 \cdot 10^{19}$ queries that the brute-force algorithm would need, which is less than its $1 \cdot 10^{-13}\%$ (see Fig. 3 for more details). We also compared the performance and number of queries for two different theories $\mathcal{T}_{\mathbb{Z}}$ and $\mathcal{T}_{\mathbb{R}}$ for *Syn (2,3)* to *Syn (2,6)*. Note, again, that the realizability result may vary if a specification is interpreted in different theories, but this is not relevant for the experiment in Fig. 4.

Lits	Heuristic setup	$\mathcal{T}_{\mathbb{Z}}$ Time (s)	$\mathcal{T}_{\mathbb{Z}}$ Queries (ou/in)	$\mathcal{T}_{\mathbb{R}}$ Time (s)	$\mathcal{T}_{\mathbb{R}}$ Queries (ou/in)
3	10/2/0/✓	0.63	8/30	0.90	14/40
4	10/2/0/✓	16.14	308/500	11.19	125/560
5	20/2/0/✓	62.44	408/3220	88.55	357/3460
6	40/2/0/✓	678.71	2094/32760	722.64	1862/35840

Fig. 4: Comparison of $\mathcal{T}_{\mathbb{Z}}$ and $\mathcal{T}_{\mathbb{R}}$ for *Syn (2,3)* to *Syn (2,6)*.

6 Related Work and Conclusions

Related work. Constraint LTL [11] extends LTL with the possibility of expressing constraints between variables at bounded distance (of time). The theories considered are a restricted form of $\mathcal{T}_{\mathbb{Z}}$ with only comparisons with additional restrictions to overcome undecidability. In comparison, we do not allow predicates to compare variables at different timesteps, but we prove decidability for all theories with an $\exists^*\forall^*$ decidable fragment. LTL modulo theories is studied in [19,12] for finite traces and where they allow temporal operators within predicates, leading to undecidability.

As for works closest to ours, [7] proposes numerical LTL synthesis using an interplay between an LTL synthesizer and a non-linear real arithmetic checker. However, [7] overapproximates the power of the system and hence it is not precise for realizability. Linear arithmetic games are studied in [13] introducing algorithms for synthesizing winning strategies for non-reactive specifications. [22] considers infinite theories (like us), but it does not guarantee success or termination, whereas our Boolean abstraction is complete. They only consider safety, while our approach considers all LTL. The follow-up [23] has still similar limitations: only liveness properties that can be reduced to safety are accepted, and guarantees termination only for the unrealizability case. Similarly, [18] is incomplete, and requires a powerful solver for many quantifier alternations, which can be reduced to 1-alternation, but at the expense of the algorithm being no longer sound for the unrealizable case (e.g., depends on Z3 not answering "unknown"). As for [34], it (1) only considers safety/liveness GR(1) specifications, (2) is limited to the theory of fixed-size vectors and requires (3) quantifier elimination (4) and guidance. We only require $\exists^*\forall^*$-satisfiability (for Boolean abstraction) and we consider multiple infinite theories. The usual main difference 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.

Temporal Stream Logic (TSL) [16] extends LTL with complex data that can be related accross time, making use of a new *update* operator $[\![y \leftarrowtail f x]\!]$, to indicate that y receives the result of applying function f to variable x. TSL is later

extended to theories in [15,25]. In all these works, realizability is undecidable. Also, in [8] reactive synthesis and syntax guided synthesis (SyGuS) [1] collaborate in the synthesis process, and generate executable code that guarantees reactive and data-level properties. It also suffers from undecidability: both due to the undecidability of TSL [16] and of SyGus [6]. In comparison, we cannot relate values accross time but we provide a decidable realizability procedure.

Comparing TSL with $\text{LTL}_{\mathcal{T}}$, TSL is undecidable already for safety, the theory of equality and Presburger arithmetic. More precisely, TSL is only known to be decidable for three fragments (see Thm. 7 in [15]). TSL is (1) semi-decidable for the reachability fragment of TSL (i.e., the fragment of TSL that only permits the next operator and the eventually operator as temporal operators); (2) decidable for formulae consisting of only logical operators, predicates, updates, next operators, and at most one top-level eventually operator; and (3) semi-decidable for formulae with one cell (i.e., controllable outputs). All the specifications considered for empirical evaluation in Section 5 are not within the considered decidable or semi-decidable fragments. Also, TSL allows (finite) uninterpreted predicates, whereas we need to have predicates well defined within the semantics of theories of specifications for which we perform Boolean abstraction.

Conclusion. The main contribution of this paper is to show that $\text{LTL}_{\mathcal{T}}$ is decidable via a Boolean abstraction technique for all theories of data with a decidable $\exists^*\forall^*$ fragment. Our algorithms create, from a given $\text{LTL}_{\mathcal{T}}$ specification where atomic propositions are literals in such a theory, an equi-realizable specification with Boolean atomic propositions. We also have introduced efficient algorithms using SAT solvers for efficiently traversing the search space. A SAT formula encodes the space of reactions to be explore and our algorithms reduce this space by learning uninteresting areas from each reaction explores. The fastest algorithm uses a two layer SAT nested encoding, in a DPLL(T) fashion. This search yields dramatically more efficient running times and makes Boolean abstraction applicable to larger cases. We have performed an empirical evaluation of implementations of our algorithms. We found empirically that the best performances are obtained when there is a balance in the number of queries made by each layer of the SAT-search. To the best of our knowledge, this is the first method to propose a solution (and efficient) to realizability for general $\exists^*\forall^*$ decidable theories, which include, for instance, the theories of integers and reals.

Future work includes first how to improve scalability further. We plan to leverage quantifier elimination procedures [9] to produce candidates for the sets of valid reactions and then check (and correct) with faster algorithms. Also, optimizations based in quasi-reactions can be enhanced if state-of-the-art tools for satisfiability core search (e.g., [24,3,2]) are used. Another direction is to extend our realizability method into a synthesis procedure by synthesizing functions in $\mathcal{T}$ to produces witness values of variables controlled by the system given (1) environment and system moves in the Boolean game, and (2) environment values (consistent with the environment move). Finally, we plan to study how to extend $\text{LTL}_{\mathcal{T}}$ with controlled transfer of data accross time preserving decidability.

References

1. Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *In Proc. of Formal Methods in Computer-Aided Design, (FMCAD) 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8. IEEE, 2013.
2. Jaroslav Bendík and Kuldeep S. Meel. Counting maximal satisfiable subsets. In *In Proc. of the 35th AAAI Conf. on Artificial Intelligence, (AAAI'21)*, pages 3651–3660. AAAI Press, 2021.
3. Jaroslav Bendík and Kuldeep S. Meel. Counting minimal unsatisfiable subsets. In *In Proc. of the 33rd Int'l Conf. in Computer Aided Verification, (CAV'21), Part II*, volume 12760 of *LNCS*, pages 313–336. Springer, 2021.
4. Roderick Bloem, Hana Chockler, Masoud Ebrahimi, and Ofer Strichman. Vacuity in synthesis. *Formal Methods Syst. Des.*, 57(3):473–495, 2021.
5. Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.
6. Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. What's decidable about syntax-guided synthesis? *CoRR*, abs/1510.08393, 2015.
7. Chih-Hong Cheng and Edward A. Lee. Numerical LTL synthesis for cyber-physical systems. *CoRR*, abs/1307.3722, 2013.
8. Wonhyuk Choi, Bernd Finkbeiner, Ruzica Piskac, and Mark Santolucito. 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 (PLD'22)*, pages 229–243. ACM, 2022.
9. Dennis W. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7(2):91–100, 1972.
10. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4693 of *LNCS*, pages 337–340. Springer, 2008.
11. Stéphane Demri and Deepak D'Souza. An automata-theoretic approach to constraint LTL. *Inf. Comput.*, 205(3):380–415, 2007.
12. Rachel Faran and Orna Kupferman. LTL with arithmetic and its applications in reasoning about hierarchical systems. In *Proc. of the 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, (LPAR-22.), Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 343–362. EasyChair, 2018.
13. Azadeh Farzan and Zachary Kincaid. Strategy synthesis for linear arithmetic games. *Proc. ACM Program. Lang.*, 2(POPL):61:1–61:30, 2018.
14. Bernd Finkbeiner. Synthesis of reactive systems. In Javier Esparza, Orna Grumberg, and Salomon Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 72–98. IOS Press, 2016.
15. Bernd Finkbeiner, Philippe Heim, and Noemi Passing. 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*, pages 325–346. Springer, 2022.
16. Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the Bools. In Isil Dillig and Serdar Tasiran, editors, *Proc. of the 31st Int'l Conf. on Computer Aided Verification (CAV'19), Part I*, volume 11561 of *LNCS*, pages 609–629. Springer, 2019.

17. Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):519–539, 2013.
18. Andrew Gacek, Andreas Katis, Michael W. Whalen, John Backes, and Darren D. Cofer. Towards realizability checking of contracts using theories. In *Proc. of the 7th International Symposium NASA Formal Methods (NFM'15)*, volume 9058 of *LNCS*, pages 173–187. Springer, 2015.
19. Alessandro Gianola and Nicola Gigante. LTL modulo theories over finite traces: modeling, verification, open questions. In *Proc. of the 4th Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis*, volume 3311 of *CEUR Workshop Proceedings*, pages 13–19. CEUR-WS.org, 2022.
20. Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Automata, logics, and infinite games: A guide to current research [outcome of a dagstuhl seminar, february 2001]. volume 2500 of *LNCS*. Springer, 2002.
21. Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur, and Leander Tentrup. 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*, pages 116–143, 2017.
22. Andreas Katis, Grigory Fedyukovich, Andrew Gacek, John D. Backes, Arie Gurfinkel, and Michael W. Whalen. Synthesis from assume-guarantee contracts using skolemized proofs of realizability. *CoRR*, abs/1610.05867, 2016.
23. Andreas Katis, Grigory Fedyukovich, Huajun Guo, Andrew Gacek, John Backes, Arie Gurfinkel, and Michael W. Whalen. 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*, pages 176–193. Springer, 2018.
24. Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.
25. Benedikt Maderbacher and Roderick Bloem. Reactive synthesis modulo theories using abstraction refinement. In *22nd Formal Methods in Computer-Aided Design, (FMCAD'22)*, pages 315–324. IEEE, 2022.
26. Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.
27. Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 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*, pages 578–586. Springer, 2018.
28. Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Proc. of VMCAI'06*, pages 364–380. Springer, 2006.
29. Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symp. on Foundations of Computer Science (FOCS'77)*, pages 46–67. IEEE CS Press, 1977.
30. Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proc. of the 16th Annual ACM Symp. on Principles of Programming Languages (POPL'89)*, pages 179–190. ACM Press, 1989.
31. Amir Pnueli and Roni Rosner. 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*, pages 652–671. Springer, 1989.
32. Alfred Tarski. Theorem proving in arithmetic without multiplication. *University of California Press.*, 1951.

33. Wolfgang Thomas. 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*, pages 635–655. Springer, 2008.
34. Adam Walker and Leonid 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*, pages 219–226. IEEE, 2014.