Efficient Handling of Obligation Constraints in Synthesis from Omega-Regular Specifications

Saqib Sohail and Fabio Somenzi
University of Colorado at Boulder

***Abstract*—A finite state reactive system (for instance a hardware controller) can be specified through a set of ω-regular properties, most of which are often safety properties. In the game-based approach to synthesis, the specification is converted to a game between the system and the environment. A deterministic implementation is obtained from the game graph and a system's winning strategy. However, there are obstacles to extract an efficient implementation from the game in hardware. On the one hand, a large space must be explored to find a strategy that has a concise representation. On the other hand, the transition structure inherited from the game graph may correspond to a state encoding that is far from optimal.**

In the approach presented in this paper, the game is formulated as a sequence of Boolean equations. That leads to significant improvements in the quality of the implementation compared to existing automata-based techniques. It is also shown discussed to extend this approach to the synthesis from obligation properties.

I. Introduction

Synthesizing reactive systems from ω-regular specifications [20] allows designers to focus on intended behavior rather than implementation details. Acceptance of automated techniques, however, is in proportion to their ability to deliver designs that meet cost and performance targets and are comparable to those produced by humans.

We present techniques that increase the performance of synthesis algorithms and lead to more compact implementations. While the algorithms are general, our implementation is geared towards the synthesis of hardware controllers. We assume that the specification of the reactive system to be synthesized is given by a list of ω-regular properties (system guarantees) that must hold when the environment satisfies another list of ω-regular properties (environment assumption). Each property is first translated to a deterministic parity automaton [18]. In special cases—like specifications in general reactive(1) form [19]—our algorithms take full advantage of this restricted form of specification. However, we do not impose restrictions on the input specification beyond ω-regularity.

Our approach extends the one of [21] in several ways. We synthesize safety properties by first reducing them to relation constraints and then manipulating the constraints in a fully symbolic form. Unlike previous techniques, this approach does not tend to embed the structures of the property automata in the implementation and produces designs with fewer state bits, better state encoding, and simpler combinational logic. It can also be considerably faster than techniques that work on explicit representations of the automata, especially when many safety properties are combined. The result of the process is a parameterized system suitable for incremental synthesis. This is useful when the specification has more than just safety or obligation properties.

We extend the symbolic approach based on transition constraint to the synthesis from obligation properties [14], which include implications between safety properties. For such implications, in particular, we show how to reduce the synthesis game to two safety games based on transition constraints.

The typical approach of automatic synthesis from the specification derives deterministic automata for each safety property; these automata operate in parallel to constrain the transitions of the system. The transition function of the composition of these automata is then inherited by the implementation. In this paper, we propose a novel way to extract the transition function of the implementation. When all the safety properties describe a language that can be generated by a relation so that the problem of sequential synthesis is converted to a problem of combination synthesis. Otherwise, we add just enough memory so that the conversion is correct. We then solve the problem of combinational synthesis by solving Boolean equations. The general solutions capture all the possible ways in which the system can satisfy the safety properties in the specification. The parameterized representation of the general solutions allows us to take advantage of the incremental synthesis framework of [21]. An additional advantage of our approach is that it is symbolic and thus adept at manipulating a large set of safety properties.

The size of the synthesized implementation depends on the transition function of the game and on the system's winning strategy. The authors in [1] provide a heuristic to select a winning strategy that attempts to minimize the amount of combinational logic in the implementation. On the other hand, the authors in [6] provide a heuristic to select a winning strategy that attempts to minimize the amount of sequential logic. However, improving the symbolic representation of the game obtained from the specification is not the focus of these works. The importance of efficient game representation has also been observed in [13]. The author remarks that current techniques are unable to extract an efficient transition structure of the implementation and proposes a tree-based approach to reduce the dependency of the implementation on the syntax of the specification. No experiments are reported.

The paper is organized as follows: Sec. II covers background and introduces notation. Sec. III, IV and V show that safety properties can be generated by a relation. Sec. VI and VII

This work was supported in part by SRC contract 2012-TJ-2271.

ISBN 978-0-9835678-3-7/13.

discuss the synthesis from safety properties from such a relation. Sec. VIII extends the relation-based approach to obligation properties. Experimental results are described in Sec. IX and conclusions are drawn in Sec. X.

II. Preliminaries

A. Linear-Time Properties

A finite automaton on ω-words $\langle \Sigma, Q, q_{\text{in}}, \delta, \alpha \rangle$ (an ω-automaton) is defined by a finite alphabet Σ, a finite set of states Q, an initial state $q_{\text{in}} \in Q$, a transition function $\delta : Q \times \Sigma \to 2^Q$ that maps a state and an input letter to a set of possible successors, and an acceptance condition α that describes a subset of Q^ω, that is, a set of infinite sequences of states. A *deterministic* automaton is such that $\delta(q, \sigma)$ is empty or a singleton for all states $q \in Q$ and all letters $\sigma \in \Sigma$. (In the case $\delta(q, \sigma)$ is a singleton, we write $\delta(q, \sigma) = q'$ for $\delta(q, \sigma) = \{q'\}$.) A run of automaton A on ω-word $w = w_0 w_1 \ldots$ is a sequence $q_0, q_1, \ldots$ such that $q_0 = q_{\text{in}}$, and for $i \geq 0$, $q_{i+1} \in \delta(q_i, w_i)$. A run is accepting iff (if and only if) it belongs to the set described by α, and a word is accepted iff it has an accepting run in A. The subset of Σ^ω accepted by A is the language of A, written $L(A)$.

Several types of acceptance conditions α are in use. We are concerned with parity conditions [16], [8], which concern the set of states $\mathsf{inf}(\rho)$ that occur infinitely often in a run ρ. A parity condition assigns a *priority* to each state: a condition of index k is a function $\pi : Q \to \{i \mid 0 \leq i \leq k\}$. A run ρ is accepting iff $\max\{\pi(q) \mid q \in \mathsf{inf}(\rho)\}$ is odd; that is, iff the highest recurring priority is odd [5], [9]. A deterministic parity word automaton (DPW) is a deterministic ω-automaton equipped with a parity condition; it is of minimum index if there is no other DPW for the same language with an acceptance condition of lower index. In the sequel, parity automata are deterministic and of minimum index [5]. A conjunctive parity condition is a set of parity conditions; and a run is accepting iff it is accepting by every parity condition. A disjunctive parity condition is a set of parity conditions; and a run is accepting iff it is accepting by some parity condition. Automata are also assumed to be *reduced*: all states are reachable from the initial state and the language accepted from them is nonempty.

We fix a finite set of atomic propositions X and consider the alphabet $\Sigma = 2^X$. An ω-regular linear-time property is a subset of Σ^ω that is accepted by a DPW. A linear-time *safety* property is a closed set of the product topology of Σ^ω. Safety properties are accepted by DPWs of index 2 such that there is no path from priority 0 states to priority 1 states. Non-safety properties are *progress* properties [15]. A safety automaton is *irredundant* if no two states accept the same language.

Linear-time temporal logic (LTL) is a specification mechanism that can express a subset of the ω-regular properties. Formulae of LTL are built from the atomic propositions in X by applying Boolean connectives and temporal operators U (until), R (releases), and X (next). Convenient abbreviations include G (globally), F (eventually), and W (weak until). An LTL formula is in negation normal form if negation is restricted to atomic propositions. The language described by the LTL formula ϕ is denoted by $L(\phi)$.

B. Realizability, Synthesis, and Games

An ω-regular property W is *satisfiable* if it has a model. For a given partition (X_e, X_s) of the atomic propositions, W is *realizable* if there exists a winning strategy for the player controlling X_s (the system) in the following game against the player controlling X_e (the environment): at each turn the environment and the system choose subsets of the propositions they control, jointly selecting an element of Σ. The elements chosen at successive turns form an infinite sequence $\rho \in \Sigma^\omega$. If ρ is in W, then the system wins; otherwise the environment wins. If the system has a winning strategy, then W is realizable, in which case a program or circuit satisfying W can be extracted from the strategy.

A full specification of the game requires detailing what each player knows of the opponent's choices when making its own choices. Variants of realizability result from different assumptions: If the system is fully apprised of the environment's choice for the same turn, *Mealy* realizability is obtained. In the opposite case, *Moore* realizability follows. We adopt a formulation that encompasses both Mealy and Moore realizability as special cases.

An ω-regular property ϕ over X specifying a reactive system is accepted by a DPW $A_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$. To check the realizability of ϕ, A_ϕ is interpreted as an input-based parity game $G_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$, where $X = X_{ed} \cup X_s \cup X_{ep}$, and Σ is the Cartesian product of a disclosed environment alphabet $\Sigma_{ed} = 2^{X_{ed}}$, a system alphabet $\Sigma_s = 2^{X_s}$, and a private environment alphabet $\Sigma_{ep} = 2^{X_{ep}}$. When the token is in state $q \in Q$, the environment chooses a letter σ_{ed} and discloses it to the system; then the system chooses a letter σ_s and discloses it to the environment; finally the environment selects a letter σ_{ep} and the token moves to state $q' = \delta(q, (\sigma_{ed}, \sigma_s, \sigma_{ep}))$. If the system has a strategy $\tau_s : S \times Q \times \Sigma_{ed} \to \Sigma_s \times S$ to win this game from q_{in} then ϕ is realizable[1]. The set S is the system's memory.

III. $\mathcal{R}$-Generability

This section is concerned with the safety languages that can be generated by a relation on the alphabet Σ. It characterizes the $\mathcal{R}$-generable languages in terms of the automata that accept them and establishes the correspondence between the automata-based view and the linguistic view of [7]. The automata-based approach provides efficient membership tests that are used in subsequent sections to devise an efficient synthesis procedure for properties that are $\mathcal{R}$-generable.

Definition 1. A set of infinite words $W \subseteq \Sigma^\omega$ is *$\mathcal{R}$-generable* if there exists a binary relation R on Σ such that a sequence $w_0 w_1 w_2 \ldots$ is in W iff $\forall i \geq 0$, (w_i, w_{i+1}) is in R.

The subset of Σ^ω generated by $R \subseteq \Sigma \times \Sigma$ is denoted by $L(R)$. It has been shown in [7] that a set of infinite words $W \subseteq \Sigma^\omega$ is $\mathcal{R}$-generable iff it is suffix-closed, fusion-closed, and limit-closed[2]. These concepts are defined as follows:

[1] If $|\Sigma_{ed}| = 1$ then system's winning strategy has a *Moore* implementation.

[2] In the context of synthesis, it is convenient to drop the requirement that the relation be total. As part of the realizability check of a specification, a subset of the alphabet is computed over which the relation is indeed total.

Definition 2. The language $W \subseteq \Sigma^\omega$ is *suffix-closed* if for every word $w_0w_1w_2\ldots \in W$ then the suffix $w_1w_2\ldots$ is in W. The language $W \subseteq \Sigma^\omega$ is *fusion-closed* if the words xvy and avb are in W, then $xvb \in W$ (and $avy \in W$). The language $W \subseteq \Sigma^\omega$ is *limit-closed* if whenever the words w_0a, w_0w_1b, $w_0w_1w_2c$, ... belong to W, then the limit of the prefixes $w_0, w_0w_1, w_0w_1w_2, \ldots$, which is the infinite word $w_0w_1w_2\ldots$ is also in W.

Limit-closed ω-regular languages are accepted by safety automata [11]. The structure of the ω-automata that recognize suffix-closed and fusion-closed languages are now examined. For lack of space, most proofs are omitted, except those that provide constructions used in the algorithms.

Definition 3. An automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi\rangle$ is *initially free* iff $\forall\sigma \in \Sigma\,.\,\delta(q_{\text{in}}, \sigma) = \{q' \mid \exists q \in Q\,.\,\delta(q,\sigma) = q'\}$.

Lemma 1. *An ω-regular language $W \subseteq \Sigma^\omega$ is suffix-closed iff it is accepted by an initially-free automaton over Σ.*

To check whether an ω-automaton A accepts a suffix-closed language, one constructs an initially-free automaton A' as described in the proof of Theorem 1 below. If $L(A') \subseteq L(A)$ then $L(A)$ is suffix-closed. When A is a deterministic safety automaton, if its initial state simulates every other state then $L(A)$ is suffix-closed.

Definition 4. An ω-automaton A is *1-definite* if the current state of A is determined by the most recent letter read.

The following result is a special case of the test for definiteness [17], [10]:

Lemma 2. *An automaton is 1-definite iff for every input letter $\sigma \in \Sigma$, there exists a state $q \in Q$ such that for every state $q' \in Q$, $\delta(q', \sigma)$ is either $\emptyset$ or q.*

The notion of definiteness is relaxed to characterize fusion-closed languages in terms of automata.

Definition 5. An automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi\rangle$ is *half definite* iff for every letter $\sigma \in \Sigma$ the states in $\{q' \mid \exists q \in Q\,.\,q' \in \delta(q,\sigma)\}$ are language equivalent.

Lemma 3. *If an ω-regular language $W \subseteq \Sigma^\omega$ is fusion-closed, then all deterministic automata that accept it are half-definite. If an ω-regular language $W \subseteq \Sigma^\omega$ is accepted by a half-definite deterministic automaton, then it is fusion-closed.*

Corollary 1. *An ω-regular language $W \subseteq \Sigma^\omega$ is fusion-closed and limit-closed iff it is accepted by a 1-definite safety automaton.*

The following theorem characterizes the ω-regular languages that are $\mathcal{R}$-generable in terms of the structure of their accepting automata. This provides an efficient membership test for safety languages that can be generated by relations.

Theorem 1. *A language $W \subseteq \Sigma^\omega$ is $\mathcal{R}$-generable iff it is accepted by an initially-free, 1-definite safety automaton.*

Proof: If a set $W \subseteq \Sigma^\omega$ is generated by a relation R, an initially-free, 1-definite safety automaton $A = \langle \Sigma, Q, q_{\text{in}}, \delta, Q\rangle$ can be built as follows. For each letter $\sigma \in \Sigma$ that appears in some pair of R, a state q_σ is added to Q, distinct from q_{in}. Let $\delta(q_\sigma, \sigma') = q_{\sigma'}$ for each pair $(\sigma, \sigma') \in R$. Moreover, let $\delta(q_{\text{in}}, \sigma) = q_\sigma$ for every letter σ that appears in first position in some pair of R. This guarantees that A is initially-free because q_{in} is connected to every state in $Q \setminus q_{\text{in}}$ that has at least one outgoing transition. Then, A accepts W.

If $A = \langle \Sigma, Q, q_{\text{in}}, \delta, Q\rangle$ is an initially-free, 1-definite safety automaton accepting W, a relation R is built as follows. The pair of letters (σ, σ') is added to R when $\delta(q, \sigma') = q'$ and σ is a letter that labels the transitions into q. (No pair is added to R for a state with no incoming transitions.) Then, R generates W. ∎

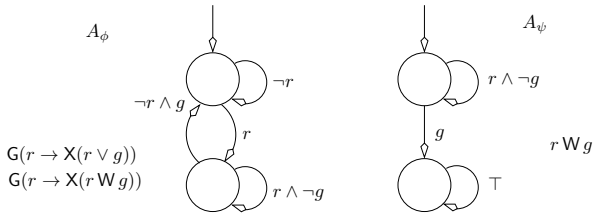

Fig. 1. Irredundant automata for three LTL formulae. All states have priority 1. A_ϕ is 1-definite and shows that both formulae represent transition constraints (transition constraints are defined in Sec. IV). A_ψ is not 1-definite because the input word $(r \wedge \neg g)^\omega$ cannot distinguish the target state.

The check of Theorem 1 can be simplified when the safety automaton is known to be deterministic and irredundant.

Lemma 4. *If a deterministic safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi\rangle$ is initially-free then it is also 1-definite.*

Example 1. Consider A_ϕ shown in Figure 1. It is a reduced, deterministic, initially-free, and 1-definite automaton. The language $L(A_\phi)$ is $\mathcal{R}$-generable and the relation R is given by the Boolean formula $(\neg r \wedge \neg r') \vee (\neg r \vee r') \vee (r \wedge r') \vee (r \wedge \neg r' \wedge g')$, which can be simplified to $\neg r \vee r' \vee g'$.

If a language $W \subseteq \Sigma^\omega$ is fusion-closed and limit-closed then it is a subset of some $\mathcal{R}$-generable language $W' \subseteq \Sigma^\omega$. For $R_{\text{in}} \subseteq \Sigma$, let $L(R_{\text{in}})$ be $R_{\text{in}}\Sigma^\omega$.

Theorem 2. *Given a fusion-closed and limit-closed language $W \subseteq \Sigma^\omega$, let $R_{in} = \{\sigma \mid \exists \sigma w \in W\}$. Then there exists an $\mathcal{R}$-generable language W' such that $W = W' \cap L(R_{in})$.*

An acceptor for the language W' in Theorem 2 is obtained through the construction in the proof of Theorem 1.

The following result is already foreshadowed in [17]. This theorem provides us with a method to detect ω-regular properties which are $\mathcal{R}$-generable.

Theorem 3. *If an irredundant safety automaton $A = \langle \Sigma, Q, q_{in}, \delta, \pi\rangle$ that accepts the ω-regular safety property φ is not 1-definite, then no other automaton that accepts φ is 1-definite.*

IV. LTL AND $\mathcal{R}$-GENERABILITY

This section provides a syntactic characterization of a subset of the LTL formulae that describe $\mathcal{R}$-generable languages. When a formula is syntactically $\mathcal{R}$-generable, the automata-based procedure of Sec. III can be skipped.

Definition 6. An LTL formula ϕ is a *transition-constraint* if it belongs to the class defined by the following grammar, in which x is a proposition:

$$P ::= \mathsf{G}(f), \qquad f ::= p \mid n \mid f \wedge f \mid f \vee f,$$
$$p ::= x \mid \neg x, \qquad n ::= \mathsf{X}\, p \ .$$

The grammar defines LTL formulae in negation normal form. Only the X operator is permissible inside the G operator and its nesting is not allowed. The set of safety properties described by Definition 6 is closed under conjunction.

Given a formula ϕ produced by the grammar in Definition 6, the relation R that generates the language of ϕ is obtained by replacing each subformula $\mathsf{X}\, x$ by x' and each subformula $\mathsf{X} \neg x$ by $\neg x'$. Finally, the G operator is discarded to obtain the propositional formula that is the representation of R. Conversely, given a relation $R \subseteq \Sigma \times \Sigma'$, an LTL safety formula ϕ_R in the form described in Definition 6 can be obtained by replacing each x' by $\mathsf{X}\, x$ and $\neg x'$ by $\neg \mathsf{X}\, x$, finally applying the G operator. (The relation $R \subseteq \Sigma \times \Sigma$ generated from a transition constraint is a Boolean formula, a minterm that satisfies this formula describes a pair $(\sigma, \sigma') \in R$ such that the cube of non-primed variables extracted from the minterm encodes σ and the cube of primed variables extracted from the minterm encodes σ'.)

Example 2. Some LTL formulae describe transition constraints even though they are do not satisfy Definition 6. Simple rewriting suffices for something like $\phi = \mathsf{G}(r \rightarrow \mathsf{X}(r \vee g))$, while the construction of an irredundant safety automaton is used to show that $\psi = \mathsf{G}(r \rightarrow \mathsf{X}(r\, \mathsf{W}\, g))$ describes the transition constraint ϕ. The formula $r\, \mathsf{W}\, g$ does not describe a transition constraint. The automaton for this property is shown on the right in Figure 1; this automaton is not 1-definite, but it is deterministic and irredundant. By Theorem 3, it does not accept an $\mathcal{R}$-generable language.

V. General Safety Properties

Safety properties like $r\, \mathsf{W}\, g$ are neither suffix-closed nor fusion-closed. The objective of this section is to find an $\mathcal{R}$-generable language $\hat{W}$ that embeds an arbitrary safety language W. It is shown that a fusion-closed and limit-closed language $\hat{W}$ over an augmented alphabet exists such that it is in one-to-one correspondence with W. Theorem 2 can be invoked to decompose $\hat{W}$ into the intersection of an $\mathcal{R}$-generable language and one that constrains the initial letter.

Given a safety language L that is not $\mathcal{R}$-generable, the problem of augmenting the alphabet Σ to $\hat{\Sigma} = \Sigma \times K$ is solved through the irredundant automaton A that accepts L.

Let $A_\phi = \langle \Sigma, Q, q_{\text{in}}, \delta, \pi \rangle$ be an irredundant deterministic safety automaton that accepts property ϕ. If A_ϕ is not 1-definite then there exists $\sigma \in \Sigma$ such that the automaton A_ϕ can be in two or more different states after reading the letter σ. This ambiguity of the irredundant automaton A_ϕ after reading one letter defines an incompatibility graph. The vertices of the graph are the states of the automaton, and there is an edge between two distinct vertices v_1 and v_2 iff there is a letter $\sigma \in \Sigma$ such that, for some states t_1 and t_2, $\delta(t_1, \sigma) = v_1$ and $\delta(t_2, \sigma) = v_2$. The chromatic number of this graph gives the minimum cardinality of the K required to turn the irredundant automaton into a 1-definite automaton. Each element of K corresponds to one of the colors and $\gamma : Q \rightarrow K$ maps states to colors. One can obtain another safety automaton $\hat{A}_\phi = \langle \hat{\Sigma}, Q, q_{\text{in}}, \hat{\delta}, \pi \rangle$, where $\hat{\Sigma} = \Sigma \times K$ and $\hat{\delta}(q, (\sigma, \gamma(\delta(q, \sigma)))) = \delta(q, \sigma)$. The label of each transition is augmented with the color of the target state; this guarantees that $\hat{A}_\phi$ is a 1-definite safety automaton. (If A_ϕ can be in several different states after reading a letter $\sigma \in \Sigma$ then all the states are colored differently in the incompatibility graph.) Since $\hat{A}_\phi$ is 1-definite, the transition function $\hat{\delta}$ can be replaced by a new transition function $\tilde{\delta} : \hat{\Sigma} \rightarrow Q$ where $\tilde{\delta}((\sigma, k)) = \{q' \mid \exists q \in Q \,.\, q' = \hat{\delta}(q, \sigma) \wedge k = \gamma(q')\}$. The state coloring function γ can also be replaced by an edge coloring function $\tilde{\gamma} : Q \times \hat{\Sigma} \rightarrow K$, where $\tilde{\gamma}(q, \sigma) = \gamma(\delta(q, \sigma))$. (The color of an initial state that does not have any incoming transitions is not important.) It will be seen in Sec. VI that the map $\tilde{\gamma}$ is convenient when checking realizability of the property ϕ through its transition constraint.

Example 3. The irredundant automaton A_{ϕ_1} for the property $\phi_1 = r\, \mathsf{W}\, g$ is shown in Figure 2. This is not a fusion-closed language, therefore no 1-definite automaton exists. Because of the input $r \wedge \neg g$ the two states are incompatible. The chromatic number of the incompatibility graph derived from A_{ϕ_1} is 2.

The automaton $\hat{A}_\phi$ derived from an irredundant A_ϕ through the coloring procedure described earlier accepts a fusion-closed and limit-closed language over the alphabet $\hat{\Sigma}$. The language of $\hat{A}_\phi$ can be represented by a relation R_ϕ and an initial predicate R_{in}. Let $\zeta : \hat{\Sigma} \rightarrow \Sigma$ be the projection function that maps letter $(\sigma, k) \in \hat{\Sigma}$ to σ; let $\zeta(w)$ and $\zeta(W)$ denote the point-wise extensions of ζ to a word $w \in \hat{\Sigma}^\omega$ and to a language $W \subseteq \hat{\Sigma}^\omega$. Then the language of $\hat{A}_\phi$ embeds the language described by ϕ so that $\zeta(L(\hat{A}_\phi)) = L(\phi)$.

The following lemma shows that the safety language accepted by A_ϕ is embedded in the language accepted by $\hat{A}_\phi$. It proves that every word in $L(A_\phi)$ has a corresponding word in $L(\hat{A}_\phi)$ through the runs of the automata A_ϕ and $\hat{A}_\phi$.

Lemma 5. *Given a safety property $W \subseteq \Sigma^\omega$, there exists an augmented alphabet $\hat{\Sigma}$ and an $\mathcal{R}$-generable language $\hat{W} \subseteq \hat{\Sigma}^\omega$ such that $\zeta : \hat{\Sigma}^\omega \rightarrow \Sigma^\omega$ is a bijection from $\hat{W}$ to W.*

Proof: Let A_ϕ be an irredundant safety automaton accepting W; let $\hat{A}_\phi$ be the 1-definite automaton obtained through the procedure described above. Let $\hat{W}$ be the language of $\hat{A}_\phi$. The automata A_ϕ and $\hat{A}_\phi$ are isomorphic and every edge (q, σ) of A_ϕ has a unique corresponding edge $(q, (\sigma, \tilde{\gamma}(q, \sigma)))$ in $\hat{A}_\phi$.

Therefore, for every word $w \in W$, there is a unique word $\hat{w} \in \hat{\Sigma}^\omega$ such that $\zeta(\hat{w}) = w$ and $\hat{w}$ has a run in $\hat{A}_\phi$. This run $\hat{\rho}$ is identical to the run ρ of w in A_ϕ. Hence, $\hat{w} \in \hat{W}$. Since, there is an injection from $\hat{W}$ to $\hat{\Sigma}$, the restriction of the function ζ to $\hat{W} \subseteq \hat{\Sigma}$ is a bijection from $\hat{W}$ to W. ■

Example 4. Continuing Example 3. The safety property $\phi_1 = r\, \mathsf{W}\, g$ is defined over the alphabet $\Sigma = 2^{\{r,g\}}$. Let $K = \{\neg x, x\}$ as $|K| = 2$. The irredundant 1-definite automaton $\hat{A}_{\phi_1}$ is shown in Figure 2. The relation R_{ϕ_1} is

$$((r \wedge \neg g \wedge \neg x) \wedge ((r' \wedge \neg g' \wedge \neg x') \vee (g' \wedge x'))) \vee (x')$$

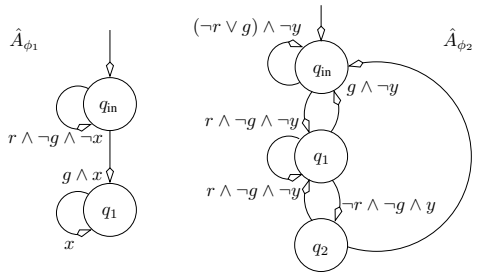

Fig. 2. Automata for $\phi_1 = r \,\mathsf{W}\, g$ and $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g)))$

and R_{in} is $((r \wedge \neg g \wedge \neg x) \vee (g \wedge x))$.

Example 5. The LTL formula $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g))$ over $\Sigma = 2^{\{r,g\}}$ does not describe a fusion-closed language. States q_{in} and q_2 are incompatible with each other. Let $K = \{\neg y, y\}$ as $|K| = 2$. The automaton $\hat{A}_{\phi_2}$ is shown in Figure 2. The predicate R_{in} is $\neg y$ and the relation R_{ϕ_2} is

$$((\neg r \wedge \neg g \wedge y \wedge ((r' \vee g') \wedge \neg y')) \vee ((\neg r \vee g) \wedge \neg y \wedge \neg y')) \\ \vee (r \wedge \neg g \wedge \neg y \wedge (((r' \vee g') \wedge \neg y') \vee (\neg r' \wedge \neg g' \wedge y'))) \ .$$

There exists another approach that can derive a transition constraint from an arbitrary safety property. A transition constraint can be derived from an LTL safety property by putting it in *separated normal form* [2]. For instance, $\mathsf{G}(r \rightarrow \mathsf{X}(r \,\mathsf{W}\, g))$ can be written as $\mathsf{G}((r \rightarrow \mathsf{X}\, x_1) \wedge (x_1 \rightarrow g \vee (r \wedge \mathsf{X}\, x_1)))$. This rewriting, however, may use more auxiliary variables than the approach based on the incompatibility graph.

Of course, given a conjunction of safety properties, obtaining transition constraints from each of them in turn and then conjoining all the transition constraints does not guarantee optimality. This is because the product automaton obtained from composing the irredundant automata for the corresponding safety properties may be neither reduced nor irredundant. In fact, the conjunction of two languages that are not fusion-closed and limit-closed may result in a fusion-closed and limit-closed (and maybe even suffix-closed) language.

VI. Realizability of Transition Constraints

This section describes how to check the realizability of a safety property ϕ embedded in a fusion-closed, limit-closed language $\hat{W}$. The language $\hat{W}$ is described by an initial predicate R_{in} and a relation R_ϕ. One can obtain an input-based game G_ϕ from the automaton A_ϕ that recognizes the language described by ϕ. One can also derive an input-based game $\hat{G}_\phi$ from the automaton that accepts $\hat{W}$. Finally a game $\hat{G}_\phi^{\mathcal{R}}$ can be derived from R_ϕ and R_{in}. It can be shown that one can obtain system's or environment's winning strategy for G_ϕ by playing $\hat{G}_\phi$ or vice-versa. Moreover, one can obtain system's or environment's winning strategy for $\hat{G}_\phi$ by playing $\hat{G}_\phi^{\mathcal{R}}$ or vice-versa. Therefore, one can obtain system's or environment's winning strategy for G_ϕ by playing $\hat{G}_\phi^{\mathcal{R}}$.

For lack of space, the details of the constructions of strategies for one game from those of the other are omitted. (These constructions are rather lengthy and tedious and are not used in the synthesis process: they are only used to prove its correctness.) However, it must be mentioned that in $\hat{G}_\phi^{\mathcal{R}}$ the input letter includes the "color" added to A_ϕ to obtain $\hat{A}_\phi$. The choice of the color is given to the system. Since there is only one way to choose the right color, and the system needs to make the right choice to win, this additional responsibility does not affect the outcome of the game.

Now it is shown how to check for the existence of winning strategies in $\hat{G}_\phi^{\mathcal{R}}$ symbolically; that is, by an algorithm that manipulates the characteristic functions of sets and relations over $\hat{\Sigma}$. The game $\hat{G}_\phi^{\mathcal{R}}$ is played in two stages; the first stage checks the realizability of R_ϕ and the second stage checks the realizability of $R_{\text{in}} \wedge R_\phi$. Given a set of target letters $T(X')$, that is, a set expressed in terms of next-state variables, the pre-image operator[3] MX is defined as follows:

$$\mathsf{MX}_\phi\, T = \forall X'_{ed} \,.\, \exists X'_s \,.\, \forall X'_{ep} \,.\, \exists X'_K \,.\, R_\phi(X, X') \wedge T(X') \ .$$

The greatest fixpoint operator MG_ϕ is defined as $\mathsf{MG}_\phi\, p = \nu Z \,.\, p \wedge \mathsf{MX}_\phi\, Z$. The realizability of R_ϕ is checked by computing the *realizable subset* $\hat{\Sigma}_r^\phi$ of $\hat{\Sigma}$ such that $\hat{\Sigma}_r^\phi = \mathsf{MG}_\phi\, \top$. The greatest fixpoint computation removes the *terminal letters* from the alphabet $\hat{\Sigma}$. The terminal letters of the alphabet are defined inductively as letters after which there does not exist a strategy to pick a next letter such that R_ϕ is satisfied or letters after which only terminal letters can be selected to satisfy R_ϕ. Finally, the realizability of $R_{\text{in}} \wedge R_\phi$ is checked. The system wins the game $\hat{G}_\phi^{\mathcal{R}}$ iff $\mathsf{MX}_{\text{in}}\, \mathsf{MG}_\phi\, \top = \top$, where

$$\mathsf{MX}_{\text{in}}\, T = \forall X'_{ed} \,.\, \exists X'_s \,.\, \forall X'_{ep} \,.\, \exists X'_K \,.\, R_{\text{in}}(X') \wedge T(X') \ .$$

For every letter in $\hat{\Sigma}_r^\phi$, the system can always pick the next letter from the same set so that R_ϕ is satisfied. The operator MX_{in} establishes the system's ability to start a word from a letter in $\hat{\Sigma}_r^\phi$ such that R_{in} is satisfied. Therefore, the system wins the game $\hat{G}_\phi^{\mathcal{R}}$ iff $\mathsf{MX}_{\text{in}}\, \mathsf{MG}_\phi\, \top = \top$, which means that the system can force the selection of a letter from $\hat{\Sigma}_r^\phi$ ($\hat{\Sigma}_r^\phi = \mathsf{MG}_\phi\, \top$) such that the predicate R_{in} is also satisfied.

Example 6. Examples 4 and 5 are continued here. Consider the property $\phi = \phi_1 \wedge \phi_2$ where $\phi_1 = r \,\mathsf{W}\, g$ and $\phi_2 = \mathsf{G}(r \wedge \neg g \rightarrow \mathsf{X}(r \vee g \vee \mathsf{X}(r \vee g)))$. Let $X_{ed} = \emptyset$, $X_s = \{r\}$, $X_{ep} = \{g\}$, and $X_K = \{x, y\}$. Initially r is asserted until g is asserted; after that, whenever r is asserted then it must be reasserted at least every other step until g is asserted. The iterates of the $\mathsf{MG}_\phi\, \top$ computation are $Z_0 = \top$, $Z_1 = (x \wedge \neg y) \vee (x \wedge \neg r \wedge \neg g) \vee (\neg y \wedge r \wedge \neg g)$, $\hat{\Sigma}_r^\phi = Z_2 = Z_1$. Since $\mathsf{MX}_{\text{in}}\, Z_2 = \top$, property ϕ is realizable.

VII. Synthesis from Transition Constraints

This section reviews Boolean equations [4] and their relation to safety games. In particular, the connection between the solution of Boolean equations and the winning strategy of the safety game is established. The synthesis approach discussed in this section scales well when the specification contains a large percentage of safety properties.

[3] In Sec. VIII the environment has to check the realizability of an assumption ϕ. In that case the pre-image operator is $\mathsf{MX}_\phi\, T = \exists X'_{ed} \,.\, \forall X'_s \,.\, \exists X'_{ep} \,.\, \exists X'_K \,.\, R_\phi(X, X') \wedge T(X')$.

A. Boolean Equations

Let $x_1, \ldots, x_m$ and $y_i, \ldots, y_n$ be two sets of variables ranging over a Boolean algebra B. ($\neg$, $\vee$, and $\wedge$ denote complementation, join, and meet in B, respectively.) A Boolean equation in independent variables $x_1, \ldots, x_m$ and unknowns $y_1, \ldots, y_n$ is a formula of the form

$$\forall x_1.., x_m.\exists y_1.., y_n.F_0(x_1.., x_m){=}F(y_1.., y_n, x_1.., x_m), \quad (1)$$

where $F_0 = \exists y_1, \ldots, y_n \,.\, F$ is the *consistency condition* of F. When no confusion arises, we write F to signify (1).

A *particular solution* of (1) is a set of Boolean functions $f_i(x_1, \ldots, x_m)$, for $1 \leq i \leq n$, such that

$$\forall x_1.., x_m \,.\, F_0(x_1.., x_m) = F(f_1.., f_n, x_1.., x_m) \ .$$

A *general solution in parametric form* of (1) is a set of Boolean functions $g_i(x_1, \ldots, x_m,\, p_1, \ldots, p_i)$, for $1 \leq i \leq n$, where each p_j is a Boolean function of $x_1, \ldots, x_m$, such that

$$\forall p_1.., p_n \,.\, \forall x_1.., x_m \,.\, F_0(x_1.., x_m) = F(g_1.., g_n, x_1.., x_m) \ ,$$

and for every particular solution $\{f_1, \ldots, f_n\}$ of (1) there is a choice of p_j's that produces a particular solution $\{f'_1, \ldots, f'_n\}$ such that, for $1 \leq i \leq n$,

$$\forall x_1.., x_m \,.\, F_0(x_1.., x_m) \leq f_i(x_1.., x_m) \leftrightarrow f'_i(x_1.., x_m) \ .$$

A general solution to (1) can be computed by the method of *successive eliminations* [4], which, given F, returns F_0 and the solution functions g_i. Letting $F_n = F$ and $F_{i-1} = \exists y_i \,.\, F_i$ for $1 \leq i \leq n$, it produces

$$\begin{aligned} g_i = \neg F_i(g_1.., g_{i-1}, \bot, x_1.., x_m) \vee \\ (p_i \wedge F_i(g_1.., g_{i-1}, \top, x_1.., x_m)) \ . \end{aligned} \quad (2)$$

Example 7. Consider $F_2 = (\neg x_1 \wedge y_1) \vee (x_2 \wedge y_2)$. Then

$$\begin{aligned} F_1{=}\exists y_2.F_2{=}(\neg x_1 \wedge y_1) \vee x_2 \quad F_0{=}\exists y_1.F_1{=}\neg x_1 \vee x_2 \\ g_1(x_1, x_2, p_1) = \neg x_2 \vee (p_1 \wedge \neg x_1) \vee (p_1 \wedge x_2) \\ g_2(x_1, x_2, p_1, p_2) = x_1 \vee (x_2 \wedge \neg p_1) \vee p_2 \ . \end{aligned}$$

One can verify that $\forall p_1, p_2, x_1, x_2.\neg x_1 \vee x_2{=}F(g_1, g_2, x_1, x_2)$.

Setting $p_1 = p_2 = \bot$ in g_1 and g_2, one obtains the particular solution $f_1 = \neg x_2$, $f_2 = x_1 \vee x_2$. The same solution is obtained for $p_1 = \neg x_2$ and $p_2 = x_1 \vee x_2$. The particular solution $f'_1 = \neg x_2$, $f'_2 = x_2$ cannot be obtained from g_1 and g_2, but, for $i \in \{1, 2\}$, $\forall x_1, x_2 \,.\, \neg x_1 \vee x_2 \leq f_i \leftrightarrow f'_i$. Note that when the consistency condition is identically satisfied,

$$\neg F_i(g_1.., g_{i-1}, \bot, x_1.., x_m) \leq F_i(g_1.., g_{i-1}, \top, x_1.., x_m)) \ .$$

Therefore, if p_i is taken in the interval defined by the two bounds, the particular solution obtained for y_i is p_i itself.

Solving a Boolean equation can be interpreted as finding winning strategies for a two-player game. One player selects a value $(\hat{x}_1, \ldots, \hat{x}_m) \in (\{\bot, \top\})^m$ for the independent variables, while the other must choose a value $(\hat{y}_1, \ldots, \hat{y}_n) \in (\{\bot, \top\})^n$ for the unknowns such that $F_0(\hat{x}_1, \ldots, \hat{x}_m) = F(\hat{y}_1, \ldots, \hat{y}_n,\, \hat{x}_1, \ldots, \hat{x}_m)$. A particular solution to the equation gives one winning strategy for the second player, while a general solution describes all winning strategies (that differ over the consistency condition).

B. Parameterized Solutions and Transition Constraints

Once the realizability of an ω-regular safety property ϕ is established, an implementation that satisfies ϕ can be generated from a system's winning strategy in the game $\hat{G}^{\mathcal{R}}_{\phi}$. This section describes the procedure to obtain an implementation that satisfies ϕ from the initial predicate R_{in} and relation R_ϕ. The solution of equations derived from R_{in} defines the initial condition, while the solution of equations derived from R_ϕ defines the steady state behavior.

The parameterized reactive system M_ϕ that implements ϕ consists of the solutions for the initial values and steady state values for variables in $X'_s \cup X'_K$ and a state variable $\mathcal{I}$ which is initially $\bot$ and then is $\top$ forever. For each element $u' \in (X'_s \cup X'_K)$, let u_{in} be the initial solution and u_∞ the steady-state solution. The initial value of u is $\bot$ and its update is given by $u' = (\neg\mathcal{I} \wedge u'_{\text{in}}) \vee (\mathcal{I} \wedge u'_\infty)$. The initialization bit distinguishes between the initial and steady state solutions.

We now describe how the solutions for initial values and steady state values are computed. The relation R_ϕ is defined over the variables $\hat{X}$ and $\hat{X}'$, where $\hat{X} = X_{ed} \cup X_s \cup X_{ep} \cup X_K$, while $\hat{X}' = X'_{ed} \cup X'_s \cup X'_{ep} \cup X'_K$. Given $\hat{\Sigma}^\phi_r = \mathsf{MG}_\phi \top$ the following four relations are used to synthesize an implementation for the property ϕ:

$$\begin{aligned} F &= R_\phi(\hat{X}, \hat{X}') \wedge \hat{\Sigma}^\phi_r(\hat{X}') & F_s &= \forall X'_{ep} \,.\, \exists X'_K \,.\, F \\ I &= R_{\text{in}}(\hat{X}') \wedge \Sigma^\phi_r(\hat{X}') & I_s &= \forall X'_{ep} \,.\, \exists X'_K \,.\, I \ . \end{aligned}$$

The existence of solutions of these Boolean equations has been established by checking the realizability of ϕ through R_{in} and R_ϕ. The steady state solution for the variables in X'_s is computed from F_s. The steady state solution for the variables in X'_K is computed from F. The solution for the initial values for variables in X'_s is computed from I_s. The solution for the initial values for variables in X'_K is computed from I. If a variable X'_s appears in the steady-state (initial) solution of X'_K then it is substituted by its steady-state (initial) solution.

Example 8. Continuing Example 6, a system M_ϕ is obtained through the synthesis of x', y' and r'. The steady state Boolean equation for the unknown variables $\{x', y'\}$ is $F_4 = R_\phi(\hat{X}, \hat{X}') \wedge \hat{\Sigma}^\phi_r(\hat{X}')$. This equation can be computed from the values of R_ϕ and Σ^ϕ_r described in Example 6. Let $\{r_i, x_i, y_i\}$ be the set of parameters, then the steady state solution of variables $\{x', y'\}$ is given by:

$$\begin{aligned} y'_\infty &= \neg F_3^{\neg y'} \vee (y_i \wedge F_3^{y'}) = \ r \wedge \neg g \wedge \neg p_1 \wedge x \wedge \neg g' \ , \\ x'_\infty &= \neg F_4^{\neg x'} \vee (x_i \wedge F_4^{x'}) = \ \neg r \vee g \vee x \vee g' \ . \end{aligned}$$

If the variable y' had appeared in x'_∞ then y' would be substituted by its function y'_∞. The steady state solution of the variables in $\{r'\}$ is computed from the Boolean equation $F_1 = \forall g' \,.\, \exists\{x', y'\} \,.\, F_4$, where

$$r'_\infty = \neg F_1^{\neg r'} \vee (r_i \wedge F_1^{r'}) = \ p_1 \vee \neg x \vee y \ ,$$

Likewise the initial values for $\{x', y'\}$ are synthesized from $I = ((r' \wedge \neg g' \wedge \neg x') \vee (g' \wedge x')) \wedge \neg y'$, where

$$y'_{\text{in}}{=}\neg I_3(0) \vee \big(y_i \wedge I_3(1)\big){=}\bot, x'_{\text{in}}{=}\neg I(0) \vee \big(x_i \wedge I(1)\big){=}g'.$$

The initial value of r' is computed from $I_1 = \forall X'_{ep} \,.\, \exists X'_K \,.\, I$, where $r'_{\text{in}} = \neg I_1(0) \vee \big((r_i \wedge I_1(1)\big) = \top$.

Each variable $v \in \hat{X}$ represents a latch (register) which stores the current value of the corresponding value of $v' \in \hat{X}'$. Each variable $o' \in X'_s$ represents the output of the sequential machine and is labeled as the corresponding variable o. Each variable $o' \in X'_K$ is stored in the latch represented by the variables in X_K, these are treated as internal signals.

The solution is kept in parameterized form so that winning strategies for the progress properties can be found. This is done by computing the appropriate values of the parameters (which may be functions requiring some finite memory to satisfy the progress properties). If the specification does not contain any progress properties then a simplified M_ϕ can be obtained by assigning any values to the parameters.

VIII. Obligation Properties

If a game with an ω-regular winning condition has a graph with more than one strongly connected component (SCC) then the winning and losing states can be computed inductively starting from the terminal SCCs. At each non-terminal SCC, one computes the states that each player can control to its winning states outside of the SCC (which are already known). The game is then played on the remaining states. This approach is discussed in [12]. In this section this idea is applied to the obligation properties defined by the implication of two safety properties (e.g., environment assumption and system guarantee). Every obligation property results in a DPW of minimum index 1 with more than one SCC.

To check realizability of an implication between two safety properties such as $\psi \rightarrow \phi$, one converts ψ and ϕ to parity games G_ψ and G_ϕ with safety conditions π_ψ and π_ϕ. The SCCs of their product can be partitioned in three ordered sets; the bottom set S_B contains the states in which the antecedent has been violated. The middle set S_M contains the states in which only the consequent has been violated, and the top set S_T contains the states where both properties hold. The states in $S_T \cup S_B$ have priority 1; those in S_M have priority 0.

This game does not need to be built explicitly, though. Given an implication $\Phi = \psi \rightarrow \phi$, where both ψ and ϕ are safety properties, one can obtain the relations R_ψ and R_ϕ as described in Sec. V. These relations are used to check the realizability of Φ. The pre-image operator MX_ϕ defined in Sec. VI cannot be used for checking Φ because it computes the states that can be forced by the protagonist to stay within the SCC, while in a game obtained from Φ, the protagonist may be able to win the game by staying within S_T or by forcing a move out of S_T to states from which it can force the play to S_B. Therefore, a modified pre-image operator needs to be defined that takes into account the protagonist's option to escape the SCC. For lack of space we only outline its use.

The solution of the game obtained from Φ follows three steps. In the first step one plays the game $\mathsf{MG}_\psi \top$ to compute the letters from which the environment can satisfy R_ψ. In this game the environment is the protagonist and the system is the antagonist. One may need to augment the alphabet Σ to $\tilde{\Sigma} = \Sigma \times K_e$ as described in Sec. V; the control of coloring variables X_{K_e} is assigned to the environment. The system is eventually able to force a violation of R_ψ from the letters in $\tilde{\Sigma} \backslash \Sigma_r^\psi$. From the letters Σ_r^ψ, the system can only satisfy R_Φ by satisfying R_ϕ. The new pre-image operator is used to compute $\mathsf{MG}_\Phi \top$; the objective of the system is to keep satisfying R_ϕ while the environment cannot use strategies that will give the system the option to choose the letter from $\tilde{\Sigma} \backslash \Sigma_r^\psi$. Once again, one may need to augment the alphabet $\tilde{\Sigma}$ to $\hat{\Sigma} = \tilde{\Sigma} \times K_s$. The system is able to satisfy R_Φ from the letters in $\Sigma_r^\Phi = \mathsf{MG}_\Phi \top \cup (\hat{\Sigma} \setminus (\Sigma_r^\psi \times K_s))$. Finally, the system wins the game obtained from Φ when the constraint $R_{\text{in}}^\psi \rightarrow R_{\text{in}}^\phi$ allows the system to select a letter from Σ_r^Φ.

As discussed in Sec. V, the control of coloring variables is assigned to the player who is trying to satisfy the property. This is why the variables in X_{K_e} (X_{K_s}) are controlled by the environment (system) when it is trying to satisfy ψ (ϕ). This distinction is at work in the following example.

Example 9. Consider the LTL formula $\Phi = \psi \rightarrow \phi$, where $\psi = \mathsf{G}(r \wedge \mathsf{X}\, r \rightarrow \mathsf{XX}(r \rightarrow l))$ and $\phi = \mathsf{G}((r \wedge \neg l \rightarrow \neg g) \wedge (r \wedge \mathsf{X}\, r \rightarrow \mathsf{XX}(r \rightarrow g)))$. Then R_{in}^ψ is $\neg x$ and R_ψ is

$$(\neg r \rightarrow \neg x') \vee (r \wedge \neg x \rightarrow (r' \leftrightarrow x')) \vee (r \wedge x \rightarrow ((r' \wedge l' \wedge x') \vee (\neg r' \wedge \neg x'))) \ ,$$

while R_{in}^ϕ is $(\neg r \vee l \vee \neg g) \wedge \neg y$ and R_ϕ is

$$(r \wedge \neg l \rightarrow \neg g) \wedge \big((\neg r \rightarrow \neg y') \vee (r \wedge \neg y \rightarrow (r' \leftrightarrow y')) \vee (r \wedge y \rightarrow ((r' \wedge g' \wedge y') \vee (\neg r' \wedge \neg y')))\big) \ .$$

The alphabets $\Sigma_\Phi = 2^{\{r,l,g\}}$ and $\hat{\Sigma}_\Phi = 2^{\{r,l,g,x,y\}}$, where

$$X_{ed}{=}\{r,l\},\ X_s{=}\{g\},\ X_{ep}{=}\emptyset,\ X_{K_e}{=}\{x\},\ X_{K_s}{=}\{y\}.$$

The system loses both games $G_{\neg\psi}$ and G_ϕ, but it can win the game G_Φ. In the game G_ϕ the environment can force the system to violate ϕ at any time by playing the sequence $r \wedge \neg l, r \wedge \neg l, r \wedge \neg l$. On the other hand, in the game G_Φ, this sequence forces the environment to violate ψ and if the environment never plays this sequence then system can always satisfy ϕ. Thus G_Φ is won by the system.

IX. Experimental Results

The approach described here has been implemented in Vis [3] as an extension of the Safety-First approach described in [21]. We report on preliminary experiments conducted on a parameterized example coming from [1] (Amba Bus). The performance and quality of the implementation is compared in Table I to Anzu [1] and our Safety-First approach.

The specifications used for Anzu are different from those used by the other two tools because Anzu requires the safety components to be pre-synthesized into transition constraints. The salient feature of our approach is the significantly smaller sizes of the implementation. We only report latch count in [21], but starting from a more abstract specification than Anzu was paid with higher latch and gate counts. The new approach, however, keeps the simpler and more abstract specification, and still manages to achieve the most efficient designs.

The specification of the Amba bus controller for n clients contains $n-1$ properties that are $\mathcal{R}$-generable, but are not

TABLE I. Experimental Results

Model	Safety		Parity		Properties	Time(s)			latches			Gates	
	E	S	E	S	ANZU	ANZU	SF	SF+TC	ANZU	SF	SF+TC	ANZU	SF+TC
AMBA2	3	17	2	3	56	2.39	6.87	0.44	24	37	15	4409	281
AMBA3	4	22	2	4	68	44.67	14.2	1.25	30	42	18	20686	586
AMBA4	5	26	2	5	80	35.30	109.9	3.47	34	48	20	17501	860
AMBA5	6	31	2	6	93	224.06	139.7	5.23	39	56	22	48154	1747
AMBA6	7	34	2	7	105	1011.7	301.1	10.18	43	55	23	74948	1792
AMBA7	8	38	2	8	117	1758.5	965.6	17.93	48	61	24	88808	1714
AMBA8	9	41	2	9	129	2034.9	875.3	76.72	52	67	26	222598	3621
AMBA9	10	44	2	10	141	7861.2	1439.6	193.90	57	77	28	175298	4597
AMBA10	11	48	2	11	153	28319.8	3727.6	224.21	61	81	29	172195	4941
AMBA11	12	51	2	12	165	8403.3	3154.0	410.44	65	87	30	179291	6240
AMBA12	13	55	2	13	177	49138.7	6641.2	878.63	69	92	31	224266	7223
AMBA13	14	60	2	14	189	13163.4	32562.4	1335.04	73	98	32	239494	9361
AMBA14	15	64	2	15	200	17104.9	12202.2	1865.70	77	105	33	284027	8202
AMBA15	16	69	2	16	212	TO	TO	2611.00	-	-	34	-	12385

produced by the grammar in Definition 6. The specification also contains two safety properties that are not $\mathcal{R}$-generable irrespective of the number of clients.

We also implemented a limited retiming step (not applicable in the SAFETY-FIRST approach), applied after the safety properties of the system have been synthesized. Consider the function $f = (a_L \wedge b_L) \vee f_i$, where a_L and b_L are the latched versions of the signals a and b. One can implement this function with one latch $c = a \wedge b$ then $f = c_L \vee f_i$. This step may reduce the number of memory elements in the parameterized representation of the transition function. Both the combinational and sequential logic is reduced by this step.

In the case of AMBA bus controller, retiming has significant impact because this controller can be implemented as a Moore machine. When a Moore implementation is not possible, the effectiveness of retiming may be less noticeable. The runtime of the algorithm is affected by retiming because the parameterized representation is also simplified: with fewer BDD variables finding suitable variable orders becomes easier.

The results of Table I do not make use of conversion of general safety properties to transition constraints. Rather the automata for these properties are used directly. In the case when general safety properties are converted to transition constraints, the extraction of an optimal parameterized transition relation incurs significant penalty and is being investigated.

X. CONCLUSION

We have presented a technique that obtains a significantly simpler representation of the synthesis game. This results in significant improvement in solving the game and produces implementations that are an order of magnitude smaller than previous techniques. The results being reported here include the logic that keeps track of the environment's assumptions. However, this logic is often not required after the game has been played. We are investigating techniques that will allow the extraction of an implementation that only depends on this logic when absolutely necessary.

REFERENCES

[1] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: A case study. In *In Proceedings of the Design, Automation and Test in Europe*, pages 1188–1193, 2007.

[2] A. Bolotov and M. Fisher. A resolution method for CTL branching time temporal logic. In *Fourth International Workshop on Temporal Representation and Reasoning (TIME)*. IEEE Press, 1997.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[4] F. M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer, Boston, 1990.

[5] O. Carton and R. Maceiras. Computing the Rabin index of a parity automaton. *Theoretical Informatics and Applications*, 33:495–505, 1999.

[6] R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *Proceedings of the 12th Conference on Formal Methods in Computer-Aided Design (FMCAD 2012)*, pages 91–100, 2012.

[7] E. A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983.

[8] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, Oct. 1991.

[9] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006*, pages 117–123, Miami, FL, Jan. 2006.

[10] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, second edition, 1978.

[11] L. H. Landweber. Decision problems for ω-automata. *Mathematical Systems Theory*, 3(4):376–384, 1969.

[12] M. Lange and O. Friedmann. The pgsolver collection of parity game solvers. Technical report, Ludwig-Maximilians-Universität - München, 2009.

[13] P. Madhusudan. Synthesizing reactive programs. In *CSL*, pages 428–442, 2011.

[14] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Annual ACM Symposium on Principles of Distributed Computing*, pages 377–410, Quebec City, Quebec, Canada, Aug. 1990.

[15] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[16] A. W. Mostowski. Regular expressions for infinite trees and a standard form of automata. In A. Skowron, editor, *Computation Theory*, pages 157–168. Springer-Verlag, 1984. LNCS 208.

[17] M. Perles, M. O. Rabin, and E. Shamir. The theory of definite automata. *IEEE Transactions on Electronic Computers*, pages 233–243, 1963.

[18] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science*, pages 255–264, Seattle, WA, Aug. 2006.

[19] N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *7th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 364–380. Springer, 2006. LNCS 3855.

[20] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[21] S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for the synthesis of reactive systems. *Software Tools for Technology Transfer (Online First)*, pages 1–22, 2012.