Synchronous product automaton generation for controller optimization

Vince Molnár[1] and András Vörös[1]

Budapest University of Technology and Economics, Hungary

Abstract. Nowadays, embedded and other controllers gain an even increasing role in the operation of industrial processes. However, the optimization of controlled processes is a challenging task: the complexity of the optimization is increased by the theoretical complexity of 1) controller synthesis and 2) the computation of the state space of the controlled process. In this paper we take one step towards the efficient optimization of controlled processes by introducing a technique to compute the composite state space of the controller and the controlled process. The computation of the state space is even more difficult if the process to be controlled has concurrent behaviour, where the so-called state space explosion problem often prevents the exploration of the possible behaviours. Saturation is a state space exploration strategy which can combat the state space explosion problem efficiently. In this paper we utilize saturation based iteration to efficiently compute the composite state space i.e. possible behaviours. The algorithm expects the controller as a traditional Büchi automaton and computes the composite state space on-the-fly. The output of the algorithm than can be used for optimization or other purposes.

1 Introduction

Optimization is a complex task, especially regarding controller systems. For such systems it is not enough to only solve the optimization task, but additional computation is needed to traverse the possible behaviours of the controller and the process together (controlled process). This requires the computation of the composite state space i.e. synchronous product of the controller and the process. This paper focuses on the synchronous product generation: the approach uses the so-called saturation strategy for state space exploration and encodes the product symbolically.

Saturation [1] provides an efficient iteration strategy for exploring the possible states of concurrent systems. Since it operates directly on a symbolic representation of a system, it is able to handle large state spaces.

The optimization process on Fig. 1. starts with the definition of the controller specification (i.e. desired *property* of the process), usually using one of the various temporal logics available. Due to the intuitiveness of linear time specifications this research direction gets increasing attention. In this paper we

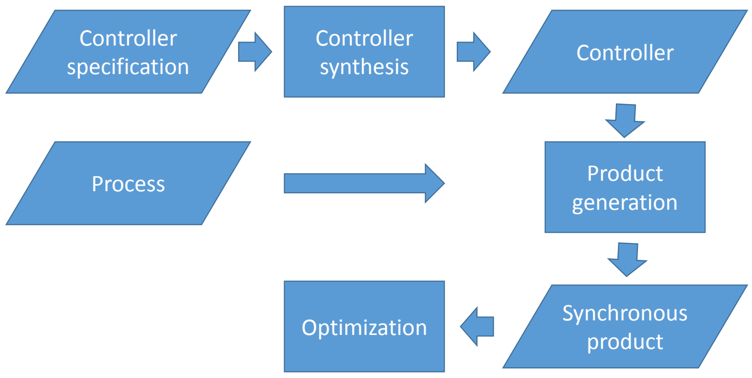


Fig. 1. Overview of the approach

follow this direction by introducing a new algorithm which combines the efficiency of saturation with the expressiveness of LTL specifications. We show how former algorithms can be used to compactly encode the synchronous product of the possible state space of the process and a Büchi automaton constructed from the LTL property. Our work focuses on the symbolic encoding of the synchronous product: the inputs are the Büchi automaton and the high level description of the process. The output is a special symbolic encoding which can then be used for synchronous product generation.

2 Background

In this section, we will briefly introduce the saturation iteration strategy and the constrained saturation algorithm, which served as a basis of our developments. At the end of the section the problem of synchronous product generation is discussed.

Saturation As mentioned before, saturation [1] is a *symbolic algorithm* for state space generation and model checking that is particularly efficient for concurrent, asynchronous systems. Saturation explores the reachable state space $\mathcal{S}_{rch}$ of a model $M = \langle \mathcal{S}, \mathcal{S}_{init}, \mathcal{E}, \mathcal{N} \rangle$ composed of K components (or subsystems), where:

- $\mathcal{S}$ is the possible set of *global* states. We define a *state variable* for each component denoted by $s_1, \ldots, s_K$ with possible *local* state spaces $\mathcal{S}_1, \ldots, \mathcal{S}_K$, so that the global state space can be defined as their Cartesian product: $\mathcal{S} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_K$. Each global state $\mathbf{s}$ is an K-tuple $\langle s_1, \ldots, s_K \rangle$, where each $s_k \in \mathcal{S}_k = \{0, 1, \ldots\}$ is the state of the kth component ($1 \leq k \leq K$). The variables are mapped into symbolic variables of the encoding decision diagrams;
- $\mathcal{S}_{init} \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{S}_{rch} \subseteq \mathcal{S}$ represents the set of states reachable from the initial states;

- $\mathcal{E}$ is the set of (asynchronous) events, usually transitions of a high-level model;
- $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{S}$ is the next state relation defined as the union of the separate next state relations of the events as following: $\mathcal{N} = \bigcup_{\varepsilon \in \mathcal{E}} \mathcal{N}_\varepsilon$, where $\mathcal{N}_\varepsilon$ is the next state relation of event ε. We often use $\mathcal{N}$ as a function, defining $\mathcal{N}(\mathbf{s}) = \{\mathbf{s}' | \langle \mathbf{s}, \mathbf{s}' \rangle \in \mathcal{N}\}$ as the states that are reachable from $\mathbf{s}$ in one step (and also $\mathcal{N}(S)$ as an extension to sets of states).

Saturation exploits the *locality* inherent in concurrent systems, where a single event usually affects only a small number of components (state variables). An event ε is *independent* from the component k, if *1)* its firing does not change the state of the component, and *2)* its enabling does not depend on the state of the component. If ε depends on component k, then we call it a supporting variable: $k \in \mathit{supp}(\varepsilon)$. Define $\mathit{Top}(\varepsilon)$ as a function that returns the largest index in $\mathit{supp}(\varepsilon)$. Then $\mathcal{E}_k$ is the set of events: $\{\varepsilon \in \mathcal{E} | \mathit{Top}(\varepsilon) = k\}$. For the sake of convenience we use $\mathcal{N}_k$ to represent the next state function of all the events $\varepsilon \in \mathcal{E}_k$, formally $\mathcal{N}_k = \bigcup_{\varepsilon \in \mathcal{E}_k} \mathcal{N}_\varepsilon$. Thus, the algorithm does not create a large, monolithic next state function representation. Instead it divides the global next state function $\mathcal{N}$ into smaller parts according to the set of events $\mathcal{E}$ in the high-level model.

Symbolic encoding of the next state functions of events $\varepsilon \in \mathcal{E}_k$ relies on the following observation: $\mathcal{N}_\varepsilon(\langle s_1, \ldots, s_K \rangle)$ and $\mathcal{N}_\varepsilon(\langle s_1, \ldots, s_k \rangle) \times \{\langle s_{k+1}, \ldots, s_K \rangle\}$ are equivalent. From this fact we can derive two important properties of saturation: *1)* in the encoding of $\mathcal{N}_\varepsilon$ it is only required to encode the state changes of state variables $s_1, \ldots, s_k$, where $k = \mathit{Top}(\varepsilon)$, as well as *2)* it is possible to apply the individual $\mathcal{N}_\varepsilon$ functions in a finer granularity: $\mathcal{N}_\varepsilon$ is applicable not only on the full state space representation, but also on the local state space representation composed of state variables $s_1, \ldots, s_k$.

Saturation uses a *special iteration strategy* driven by decision diagrams, which is highly efficient for asynchronous and concurrent systems [1]. The algorithm divides the global fixed-point computation into smaller parts, as it computes a local fixed-point with regard to a decision diagram node n_k. A node n_k is called *saturated*, if it represents a local state space computed as the fixed-point of the transitive closure of local next state relations: $\mathcal{S}(n_k) = \bigcup_{i:1 \leq i \leq k} \bigcup_{\varepsilon \in \mathcal{E}_i} \mathcal{N}_\varepsilon^*(\mathcal{S}(n_k))$, where $\mathcal{S}(n_k)$ is the set of states represented by node n_k [3].

In [3] the authors introduced an advanced saturation-based iteration strategy for structural model checking. The algorithm, called *constrained saturation*, computes the least fixed point of the reachability relation that satisfies a given constraint. Constrained saturation serves as the basis of our product computation algorithm.

Büchi automaton A Büchi automaton is similar to a finite state automaton, formally an automaton $\mathcal{A}$ is a tuple $\langle \Sigma, Q, \Delta, Q_0, F \rangle$, where Σ is a finite *alphabet*, Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*, $Q_0 \subseteq Q$ is a set of *initial states* and $F \subseteq Q$ is a set of *accepting states*. A run of an automaton over an input word is an infinite sequence of states starting with an initial state, where the transition relation holds between the consecutive states. An infinite run is *accepting* if it passes an accepting state in F infinitely often.

Synchronous Product. We define the synchronous product of the model $\mathcal{M}$ by interpreting the transition system generated by $\mathcal{M}$ as a Büchi automaton. We define a labeling function $L : \mathcal{S} \to 2^{AP}$ assigning a valuation of the atomic propositions of P to each state of $\mathcal{M}$. The alphabet of the automaton corresponding to $\mathcal{M}$ is the same as that of the property automaton: 2^{AP}. Inputs of its transitions are the valuations assigned to the target state by L. Synchronous composition with this automaton forces the property automaton $\mathcal{A}$ to read the valuations that appear on a state sequence of $\mathcal{M}$. In terms of the structures defined so far, the synchronous product can be defined as follows: $\mathcal{M} \times \mathcal{A} = \langle \Sigma, S \times Q, \Delta^{\times}, S_{init} \times Q_0, F \rangle$, where $\Delta^{\times} = \{ \langle \langle \mathbf{s}, q \rangle, \alpha, \langle \mathbf{s}', q' \rangle \rangle | \langle \mathbf{s}, \mathbf{s}' \rangle \in \mathcal{N}, \langle q, \alpha, q' \rangle \in \Delta, \alpha = L(\mathbf{s}') \}$.

3 Special Encoding Based On Constrained Saturation

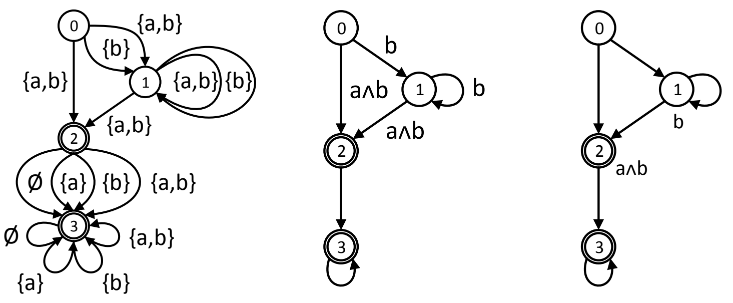
In this section, we will characterize a special form of Büchi automata on which our encoding relies. After that, we propose a new encoding that can be used as an input for the constrained saturation algorithm to compute the product state space.

3.1 Tableau Automata

The first step is the translation of the temporal expression into a Büchi automaton. Observing the output automaton of widely used tableau-based conversion algorithms (such as [2]) we identified a common structural property that can be exploited to efficiently encode and compute the product. We refer to these kinds of Büchi automata as *tableau automaton.*

Definition 1 (Tableau automaton). *A tableau automaton is a tuple $\langle AP, Q, \Delta, Q_0, F, L^+, L^- \rangle$, where*

- *AP is a set of atomic propositions, 2^{AP} being the alphabet of the automaton,*
- *Q is the set of states,*
- *$\Delta \subseteq Q \times Q$ is the transition relation,*
- *$Q_0 \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of accepting states, and*
- *$L^+ : Q \to 2^{AP}$ and $L^- : Q \to 2^{AP}$ are labeling functions, L^+ assigning propositions that must hold in the given state, while L^- assigning those that must not.*



(a) By definition (b) Compact representation (c) Tableau automaton

Fig. 2. Three forms of a Büchi automaton corresponding to the LTL property “$a \, \mathsf{R} \, b$”.

A run of a tableau automaton over an input word is also an infinite sequence of states $q_0 q_1 \ldots$ starting with an initial state $q_0 \in Q_0$, but unlike simple Büchi automata, there is an additional requirement beyond satisfying the transition relation. For every i, the input letter $\alpha_i \in 2^{AP}$ of the word representing a valuation of the atomic propositions must contain every proposition assigned to q_{i+1} by L^+ and must not contain any assigned by L^-, formally: $L^+(q_{i+1}) \subseteq \alpha_i$ and $L^-(q_{i+1}) \cap \alpha_i = \emptyset$. Accepting runs are defined the same way as for Büchi automata.

At this point, it is important to emphasize that tableau automata are only a special form of Büchi automata, with the same expressive power. An equivalent Büchi automaton has the same states (including initial and final states), the same alphabet, and a transition relation in the form of $\bigcup_{\langle q,q' \rangle \in \Delta} \{\langle q, \alpha, q' \rangle | L^+(q') \subseteq \alpha, L^-(q') \cap \alpha = \emptyset\}$. Because of this equivalence, we will often refer to a Büchi automaton directly corresponding to a tableau automaton to be in *tableau form*. Every Büchi automata can be transformed into this form.

Figure 2 shows different representations of the LTL expression $a\mathcal{R}b$. On Figure 2(a), the corresponding Büchi automaton is shown exactly as it is described by the definition: each transition in the transition relation gets an own arc. Labels of the arcs are the sets representing valuations of the atomic propositions a and b. Figure 2(b) shows the same automaton in a more compact form, merging arcs and characterizing their labels with a conjunctive expression. This automaton is in tableau form, since all of the arcs targeting the same state are labeled by the same conjunction. Moving these labels to the state itself results in a tableau automaton shown on Figure 2(c). Let ϕ_q denote the conjunction on the arcs targeting q. Then the labeling functions L^+ and L^- are defined such that every atomic proposition that is positive in ϕ_q is in $L^+(q)$, while those that are negative are returned by $L^-(q)$.

3.2 Encoding the Product Automaton

For now, we assume that the state space and the next state relations of the system are already computed. The specification is given as a Büchi automaton. Since our main goal is to exploit the power of saturation in product generation, we need to define the states and transitions in the way we did in Section 2. Formally we build a *product system* which is a tuple $\mathcal{M}^\times = \langle \mathcal{S}^\times, \mathcal{S}^\times_{init}, \mathcal{E}^\times, \mathcal{N}^\times \rangle$ collecting states in $\mathcal{S}^\times$, initial states in $\mathcal{S}^\times_{init}$ and transitions into $\mathcal{N}^\times$ preferably partitioned by events $\varepsilon \in \mathcal{E}^\times$ according to the events of the transition system.

Besides keeping the original state variables of the transition system, we need one or more additional variables to encode the states of the automaton representing the property.[1] Note that every event of the product system must affect the encoding variables of the automaton, since their steps are synchronized. For this reason, in order to keep the efficiency of the saturation iteration strategy, these variables need to be situated in the lower levels of the decision diagram encoding: inserting them immediately above the terminal level is an ideal choice. This way the encoding has no impact on the *Top* values of system events. Since the efficiency of saturation iteration strategy is *highly dependent on the Top values of system events*, not ruining the *Top* values produced by a good variable order is a sane requirement towards any algorithm.

The encoding of the transitions is a bit more challenging: as mentioned in Section 2 the steps of the system model and the property automaton need to read the same input letters, i.e. valuations of atomic propositions in the checked property. Thus it is insufficient to simply compute the Cartesian product of the next state relation of the model and the transition relation of the automaton.

In Section 2, we have already defined the product automaton of $\mathcal{M}$ and $\mathcal{A}$. To define the next state relation of the product system, we will drop the input labels of the transitions of the automaton: this can be done as saturation is not interested in what input the product automaton reads during the state space traversal as long as the system model and the property automaton both read the same word. Formally, the next state relation of the product system is $\mathcal{N}^\times = \{\langle\langle \mathbf{s}, q\rangle, \langle \mathbf{s}', q'\rangle\rangle | \exists \alpha \in 2^{AP}, \langle\langle \mathbf{s}, q\rangle, \alpha, \langle \mathbf{s}' q'\rangle\rangle \in \Delta^\times\}$. While this definition is mathematically correct and can even be realized as conjunctive-disjunctive decomposition suitable for saturation [3] (i.e. events are kept and the next state relation is the composition of next state relations of events), it fails to accomplish one of our main goals: preserving the *Top* values of events.

If the synchronization on the input word is encoded into the next state relation, *Top* values of every event are inevitably raised *to the same value* that can even be K, the highest level possible. This means that saturation's strategy to apply the next state relation in a finer granularity is spoiled, every event is processed on the same level and the optimizations of saturation targeting concurrency are lost. To understand the reason of this raise in the *Top* values, we define the subject level of atomic propositions.

[1] The property automaton is typically small enough to get encoded into a single variable, but a binary encoding (or anything in between) can also be used for a more compact representation.

We assume that the truth value of an atomic proposition is only dependent on a single state variable, we call this the *subject* of the atomic proposition. Let $Sub(p)$ denote the level on which this variable is encoded in the decision diagram. Due to the synchronization, each step of the system model results in a step of the property automaton. A step of the property automaton requires a full valuation of the atomic propositions, so every event $\varepsilon \in \mathcal{E}^{\times}$ of the product system now depends on all variables that are subjects of any $p \in AP$. By definition, this means that $supp(\varepsilon) \supseteq \{i|\exists p \in AP, Sub(p) = i\}$, i.e. the support of ε contains every level encoding variables that are subject to an atomic proposition in AP. It is easy to see that $Top(\varepsilon)$ is now at least $\max\{i|\exists p \in AP, Sub(p) = i\}$. For an example, imagine a property in which subjects of atomic propositions cover every state variable, so regardless of variable ordering, all *Top* values are raised to the maximum.

Since this is clearly not what we want to do, we devised a solution that preserves the *Top* values of the events by *decomposing the problem, separating the next state relation and the constraint of reading the same word.* This enables us to keep saturation's every advantage.

Our proposed solution exploits the way tableau automata work. Furthermore, we employ the main idea of *constrained saturation*: check and fire. Instead of intersecting relations, we 1) relax the next state relation of the product to $\mathcal{N} \times \Delta$ in order to ignore the input of the participating automata and then 2) only allow state transitions reaching *legal* states. A reached state $\langle \mathbf{s}, q\rangle$ is legal, if the valuation $L(\mathbf{s})$ determined by the system's state satisfies ϕ_q, i.e. all propositions in $L^+(q)$ are true and those in $L^-(q)$ are false. We use the characteristics of tableau automata to be able to validate *states*, not *steps* – just like constrained saturation does.

We have to constrain the steps of relaxed next state relation $\mathcal{R} = \mathcal{N} \times \Delta$ to traverse only legal states. As we utilize the constrained saturation algorithm for this purpose, we have to compute the input constraint for the algorithm. We have to recall now that constrained saturation computes the set of states $\mathcal{N}(S) \cap \mathcal{C}$ in each step during the state space traversal, where $\mathcal{C}$ is the constraint characterizing possible states. Following this idea, we define our constraint as the set of legal states: $\mathcal{C}^{\times} = \{\langle \mathbf{s}, q\rangle | s \in \mathcal{S}, q \in Q, L^+(q) \subseteq L(s), L^-(q) \cap L(\mathbf{s}) = \emptyset\}$.

Last but not least, the initial states of the product system can be obtained by pairing the appropriate initial states of $\mathcal{A}$ with initial states of M. The property automaton is typically interpreted such that the input of the first step from the initial state is the valuation implied by the initial state of M. This means that we initialize the property with the current (initial) state of the system, observing its behaviour starting from this point of time.

3.3 Abstracting the Constraint

The presented algorithm of Section 3.2 introduced an efficient encoding of the product system by decomposing the transition relation into an over-approximating next state relation and a constraint of legal states. We utilized constrained saturation to build the state space of the product system. We defined the legal state constraint as set $\mathcal{C}^{\times} = \{\langle s, q\rangle | s \in S_{rch}, q \in Q, L^{+}(q) \subseteq L(s), L^{-}(q) \cap L(s) = \emptyset\}$. With the previously defined next state relation $\mathcal{R}$ and this constraint, the constrained saturation algorithm explores the state space and builds a symbolic representation of the synchronous product.

We now introduce an abstraction layer providing the ability to build the symbolic product representation. This abstraction layer will "virtualize" the legal state constraint, letting us build the abstraction without precomputing the state space of the system model. The virtual constraint will encode the possible valuations and corresponding automaton states symbolically. Suppose that AP is a list of atomic propositions p_i ordered by $Sup(p_i)$, the level on which their subjects are encoded. Then the constraint is the set $\bigcup_{q \in Q}(p_1(q) \times \ldots \times p_n(q) \times \{q\})$ where $p_i(q)$ is a function assigning the possible valuations of p_i that satisfies the labeling of q, i.e. $p_i(q) = \{true\}$ if $p_i \in L^{+}(q)$, $p_i(q) = \{false\}$ if $p_i \in L^{-}(q)$ and $p_i(q) = \{true, false\}$ otherwise.

This approach has the advantage of using the constrained saturation algorithm with only a slight modification: the algorithm will only have to use the simple function described above to determine the next constraint node based on the valuations of the local states currently processed. This way the constraint only depends on the property automaton and can be built before starting the state space exploration.

4 Conclusion and Future Work

In this paper we introduced a basic algorithm which is suitable to serve as a basic building element of the optimization of complex controlled process systems. Our approach receives a special kind of Büchi automaton and the high level description (for example Petri net) description of the system and produces the synchronous product. The algorithm relies on the saturation iteration strategy, which is widely used for model checking concurrent systems. With some slight modifications we utilized it to be able to compute the synchronous product i.e. the possible behaviours of the controlled process. In the future we plan to integrate the introduced theory into practice and develop an on-the-fly algorithm which can exploit saturation for supporting optimization tasks.

References

1. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. Int. J. Softw. Tools Technol. Transf. 8(1), 4–25 (2006)
2. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E., Sistla, A. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 1855, pp. 248–263. Springer (2000)
3. Zhao, Y., Ciardo, G.: Symbolic CTL model checking of asynchronous systems using constrained saturation. In: Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 5799, pp. 368–381. Springer (2009)