Dealing with Incompleteness in Automata-Based Model Checking

Claudio Menghi[1(✉)], Paola Spoletini[2], and Carlo Ghezzi[1]

[1] DEIB, Politecnico di Milano, Milano, Italy
{claudio.menghi,carlo.ghezzi}@polimi.it
[2] Kennesaw State University, Marietta, USA
pspoleti@kennesaw.edu

Abstract. A software specification is often the result of an iterative process that transforms an initial incomplete model through refinement decisions. A model is incomplete because the implementation of certain functionalities is postponed to a later development step or is delegated to third parties. An unspecified functionality may be later replaced by alternative solutions, which may be evaluated to analyze tradeoffs. Model checking has been proposed as a technique to verify that a model of the system under development is compliant with a formal specification of its requirements. However, most classical model checking approaches assume that a complete model of the system is given: they do not support incompleteness. A verification-driven design process would instead benefit from the ability to apply formal verification at any stage, hence also to incomplete models. After any change, it is desirable that only the portion affected by the change, called replacement, is analyzed. To achieve this goal, this paper extends the classical automata-based model checking procedure to deal with incompleteness. The proposed model checking approach is able not only to evaluate whether a property definitely holds, possibly holds or does not hold in an incomplete model but, when the satisfaction of the specification depends on the incomplete parts, to compute the constraints that must be satisfied by their future replacements. Constraints are properties on the unspecified components that, if satisfied by the replacement, guarantee the satisfaction of the original specification in the refined model. Each constraint is verified in isolation on the corresponding replacement.

1 Introduction

The development process of any complex system can be viewed as a sequence of decisions that make the system *evolve* from an initial, high-level model into a fully detailed and verified implementation. Typically, this process is performed by iteratively decomposing the model of the system into smaller functionalities. At each stage, the model may be deliberately incomplete, either because development of certain functionalities is postponed or because the implementation will be provided by a third party, as in the case of a component-based or a service-based system. In the case of a postponed functionality, an implementation is usually provided at some later stage of the development process, possibly after exploring alternative solutions to evaluate their tradeoffs. There are also

J. Fitzgerald et al. (Eds.): FM 2016, LNCS 9995, pp. 531–550, 2016.
DOI: 10.1007/978-3-319-48989-6_32

cases in which the postponed functionality may become available at run time, as in the case of dynamically adaptive systems.

The verification community developed several techniques to check if a model of the system under development satisfies its requirements. In particular, model checking [3,10,11] has matured to a stage where practical use is now often possible. *Model checking* exhaustively analyzes the behavior of the system's model to ensure that all its executions satisfy the properties of interest. These techniques return *yes* if the model of the system satisfies its requirements, *no* and a *counterexample* in the opposite case. Mainstream model checking techniques assume that the model of the system and the properties against which it should be verified are completely defined when the verification takes place. This assumption is not always valid during the software development since, as we discussed earlier, models are often incomplete.

To support continuous verification, we should be able to *verify incomplete models* against given properties. This would allow even initial, incomplete, and high-level descriptions of the system to be verified against given properties, supporting early error detection. This is exactly the motivation of our work, which extends traditional *automata-based model checking* to verify if a model of the system $\mathcal{M}$ satisfies its properties, even when $\mathcal{M}$ is incomplete. The technique we develop assumes that $\mathcal{M}$ can contain one or more unspecified states, called *black box states*, that represent unspecified functionalities. To describe black box states, we introduce *Incomplete Büchi Automata* (IBAs), which extend the well known Büchi automata (BAs) and support the designer in the iterative, top-down refinement of a *sequential* system. Black box states can be (recursively) refined into other (I)BAs. Due to the presence of black box states, the model checking procedure is modified to produce three values: *yes* if the model of the system definitely satisfies its properties, *no* plus a counterexample if it does not, or *unknown* when the property is possibly satisfied, i.e., its satisfaction depends on the future refinement of the black box states. In this last case, a *constraint* per each black-box is synthesized, i.e., a property that must eventually be satisfied by the automata (replacement) that will replace the black box state in the refinement process. If, once refined, the replacements satisfy the synthesized constraints, then $\mathcal{M}$ fulfills its properties.

The paper is organized as follows. Section 2 introduces IBAs. Section 3 describes the advantages of using this new formalism on a small example. Section 4 shows how the classical verification procedure of BAs is modified to manage incompleteness in the model of the system. Section 5 describes how the constraint for the unspecified components is computed and used to verify the replacement of the corresponding black box state. Section 6 provides an experimental assessment of scalability of the approach. Section 7 presents related work and discusses its relation with our approach. Finally, Sect. 8 concludes the paper.

2 Modeling Formalisms

This section defines an extension of Büchi Automata [6] (BAs), called Incomplete Büchi Automata (IBAs), which support incomplete specifications, i.e., they

contain some parts left as black boxes that will be later defined. It also describes how IBAs are refined by replacing the unspecified components.

BAs are widely used models of computation that describe systems through a finite set of states and transitions. *States* are snapshots of the system configurations and *transitions* describe how the state of the system changes over time. They are labeled with *atomic propositions*, i.e., statements that are true when the transitions are performed. IBAs extend BAs by partitioning states into *regular* and *black box*. A black box state, in the following often abbreviated as (black) box, is a placeholder for a functionality that is currently left unspecified and will be later refined by another automaton.

Definition 1. *Given a set of propositions AP, an* incomplete Büchi automaton $\mathcal{M}$ *is a tuple* $\langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$*, where (a)* $\Sigma = 2^{AP}$ *is a finite alphabet; (b)* Q *is a finite set of states, partitioned into* R *(the set of* regular *states) and* B *(the set of* black box *states), such that* $Q = B \cup R$ *and* $B \cap R = \emptyset$*; (c)* $\Delta \subseteq Q \times \Sigma \times Q$ *is a transition relation; (d)* $Q^0 \subseteq Q$ *is a set of initial states; (e)* $F \subseteq Q$ *is a set of accepting states.*

Graphically, boxes are filled with black, initial states are marked by an incoming arrow, and accepting states are double circled.

As BAs, IBAs use their accepting states to recognize infinite words (also called *ω-words*), as formally defined in the following. A run of an IBA is defined as follows:

Definition 2. *Given an IBA defined over AP, a set of atomic propositions* AP'*, such that* $AP \subseteq AP'$*, and the alphabet* $\Sigma^\omega = 2^{AP'}$*, a* run $\rho : \{0, 1, 2, \ldots\} \to Q$ *over* $v \in \Sigma^\omega$ *is defined as follows: (a)* $\rho(0) \in Q^0$*; (b) for all* $i \geq 0$*,* $(\rho(i), v_i, \rho(i+1)) \in \Delta \vee ((\rho(i) \in B) \wedge (\rho(i) = \rho(i+1)))$*.*

Informally, a character v_i of the word v can be recognized by a transition of the IBA, changing the state of the automaton from $\rho(i)$ to $\rho(i+1)$, or can be recognized by a transition of the IBA that will replace the box $\rho(i) \in B$. In the latter case, the state $\rho(i+1)$ corresponds to $\rho(i)$, since the corresponding transition is fired inside the automaton that will replace $\rho(i)$. Note that characters in $AP' \setminus AP$, since they are not part of the already specified alphabet of the IBA, need to be recognized inside boxes.

Let $inf(\rho)$ be the states that appear infinitely often in the run ρ. A run ρ of an IBA $\mathcal{M}$ is *(a) definitely accepting* iff $(inf(\rho) \cap F \neq \emptyset) \wedge (\forall i \geq 0, \rho(i) \in R)$, i.e., some accepting states appear infinitely often in ρ and all of its states are regular; *(b) possibly accepting* iff $(inf(\rho) \cap F \neq \emptyset) \wedge (\exists i \geq 0 \mid \rho(i) \in B)$, i.e., some accepting states appear infinitely often in ρ and at least one of its states is a box; *(c) not accepting* otherwise.

An IBA $\mathcal{M}$ *definitely accepts* a word v iff there exists a definitely accepting run for v. Definitely accepted words describe behaviors the system is going to exhibit. $\mathcal{M}$ *possibly accepts* v iff it does not definitely accept v and there exists a possibly accepting run for v. Possibly accepted words describe possible behaviors. Finally, $\mathcal{M}$ *does not accept* v iff it does not contain any accepting or possibly

accepting run for v. A word is not accepted if it is neither a definitely accepted nor a possibly accepted behavior. The language $\mathcal{L}(\mathcal{M}) \in \Sigma^\omega$ ($\mathcal{L}_p(\mathcal{M}) \in \Sigma^\omega$) of $\mathcal{M}$, consists of all the words definitely accepted (possibly accepted) by $\mathcal{M}$. $\mathcal{L}(\mathcal{M})$ can be defined by considering the BA $\mathcal{M}_c$, called *completion*, obtained from $\mathcal{M}$ by removing its boxes and their incoming and outgoing transitions.

The refinement relation $\preceq$ allows the iterative elaboration of the model of the system by replacing boxes with other IBAs, called *replacements*. The idea behind the refinement relation is that every behavior of $\mathcal{M}$ *must be preserved* in its refinement $\mathcal{N}$, and every behavior of $\mathcal{N}$ must correspond to a behavior of $\mathcal{M}$. The final BA is obtained by substituting all the boxes with the corresponding replacements. A BA $\mathcal{N}$ is an *implementation* of an IBA $\mathcal{M}$ if and only if $\mathcal{M} \preceq \mathcal{N}$. The formal definition of refinement, further definitions, lemmas and theorems together with all the proofs of theorems and lemmas that support this work can be found in [25].

The refinement relation is both reflexive and transitive, and preserves the language containment, i.e., a possibly accepted word of $\mathcal{M}$ can be definitely accepted, possibly accepted or not accepted in the refinement $\mathcal{N}$, but every definitely accepted and not accepted word remains definitely accepted or not accepted in $\mathcal{N}$.

Consider an IBA $\mathcal{M}$. A refinement step consists of replacing a box with an (I)BA. Intuitively, a replacement defines an automaton $\mathcal{T}$ that refines the box b and the incoming Δ^{inR} and outgoing transitions Δ^{outR} which describe how $\mathcal{T}$ is connected with $\mathcal{M}$. Formally, it is defined as follows:

Definition 3. *Given an IBA* $\mathcal{M} = \langle \Sigma_\mathcal{M}, R_\mathcal{M}, B_\mathcal{M}, Q_\mathcal{M}, \Delta_\mathcal{M}, Q^0_\mathcal{M}, F_\mathcal{M} \rangle$, *a replacement* $\mathcal{R}$ *of a box* $b \in B_\mathcal{M}$ *is a triple* $\langle \mathcal{T}, \Delta^{inR}, \Delta^{outR} \rangle$. $\mathcal{T} = \langle \Sigma_\mathcal{T}, R_\mathcal{T}, B_\mathcal{T}, Q_\mathcal{T}, \Delta_\mathcal{T}, Q^0_\mathcal{T}, F_\mathcal{T} \rangle$ *is an IBA.* $\Delta^{inR} \subseteq \{(q', a, q) \mid (q', a, b) \in \Delta_\mathcal{M}$ *and* $q \in Q_\mathcal{T}\}$ *and* $\Delta^{outR} \subseteq \{(q, a, q') \mid (b, a, q') \in \Delta_\mathcal{M}$ *and* $q \in Q_\mathcal{T}\}$ *are its incoming and outgoing transitions, respectively.* $\mathcal{R}$ *must satisfy the following:*

1. *if* $b \notin Q^0_\mathcal{M}$ *then* $Q^0_\mathcal{T} = \emptyset$;
2. *if* $b \notin F_\mathcal{M}$ *then* $F_\mathcal{T} = \emptyset$;
3. *if* $(q', a, b) \in \Delta_\mathcal{M}$ *then, there exists* $(q', a, q) \in \Delta^{inR}$ *such that* $q \in Q_\mathcal{T}$;
4. *if* $(b, a, q') \in \Delta_\mathcal{M}$ *then, there exists* $(q, a, q') \in \Delta^{outR}$ *such that* $q \in Q_\mathcal{M}$;
5. *if* $(b, a, b) \in \Delta_\mathcal{M}$ *then, there exists* $(q', a, q) \in \Delta_\mathcal{T}$.

Condition 1 (2) forces the replacement of a non-initial (non-accepting) box to not contain initial (accepting) states. Condition 3 (4) forces each incoming (outgoing) transition of b to be associated with at least an incoming (outgoing) transition of the replacement. Finally, Condition 5 states that if there exist a self-loop over the box b labeled with a, there exist at least a transition labeled with a in the replacement $\mathcal{R}$. Note that this transition could be not reachable in the replacement.

Additional definitions, theorems and proofs can be found in [25].

3 Motivating Example

To describe how IBAs support iterative refinement, we consider a simple system in charge of sending messages, described in [2]. An initial, high level and incomplete model of the system $\mathcal{M}$ is shown in Fig. 1. When the system starts, it moves from q_1 to $send_1$. The state $send_1$ represents a function performing the first attempt to send the message. If the attempt succeeds, the success state q_3 is reached. Otherwise, the function $send_2$, which performs a second attempt, is activated. If the attempt succeeds, the success state q_3 is entered, otherwise the system enters the abort state q_2.

The model $\mathcal{M}$ is defined over the alphabet $\Sigma_{\mathcal{M}} = \{start, ok, fail, success, abort\}$. The ω-*word* $v = \{start\}.\{ok\}.\{success\}^{\omega}$ corresponds to two runs, ρ_1 and ρ_2. ρ_1 is a possibly accepting run such that $\rho_1(0) = q_1$, $\rho_1(1) = send_1$, and $\forall i \geq 2, \rho_1(i) = q_3$. ρ_2 is a not accepting run such that $\rho_2(0) = q_1$ and $\forall i \geq 1, \rho_2(i) = send_1$. Since there exists a possibly accepting run and no definitely accepting runs are present, $\mathcal{M}$ possibly accepts v. The word $v = \{start\}.\{success\}^{\omega}$ is instead not accepted since the only run associated with v is $\rho_3(0) = q_1$ and $\forall i \geq 1, \rho_3(i) = send_1$ which is neither definitely accepting nor possibly accepting. The language $\mathcal{L}(\mathcal{M})$ of behaviors associated with $\mathcal{M}$ is empty since there are no words accepted by $\mathcal{M}_c$. $\mathcal{M}_c$ is obtained by removing $send_1$ and $send_2$ and their incoming and outgoing transitions.

Figure 2 presents a BA $\mathcal{N}$ which is a refinement of the IBA $\mathcal{M}$. The boxes $send_1$ and $send_2$ are replaced with two instances of the same functionality $\mathcal{R}$, depicted inside two dashed-dotted frames. $\mathcal{R}$ *sends* a message and *waits* for an answer. If a *timeout* occurs, the sending procedure fails. If an *acknowledgement* is received, the procedure *succeeds* or *fails* depending on the type of

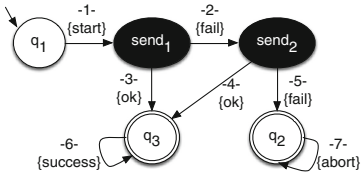


Fig. 1. The model $\mathcal{M}$.

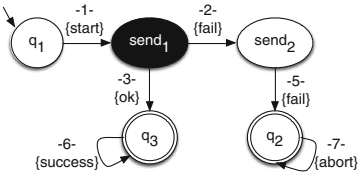


Fig. 3. A refinement $\mathcal{N}'$ of $\mathcal{M}$.

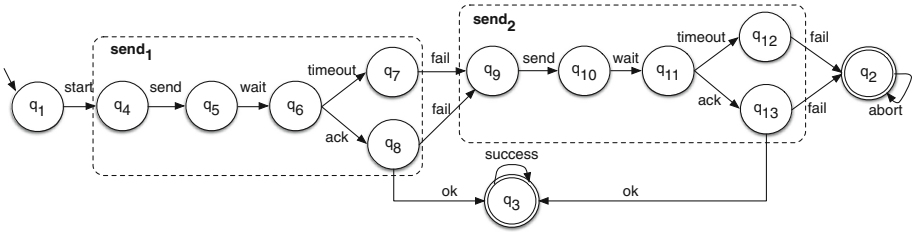


Fig. 2. A refinement $\mathcal{N}$ of $\mathcal{M}$.

acknowledgement. When the sending performed by $send_1$ fails, another attempt is performed by $send_2$, whose failure leads entering state q_2.

Figure 3 shows a hypothetical alternative refinement, which will be used later to explain our approach, where box $send_2$ is replaced by a component that always fails.

4 Automata-Based Checking

Given a model $\mathcal{M}$ and a property ϕ, model checking is used to verify whether $\mathcal{M}$ satisfies ϕ. When incomplete models are considered, the model checking algorithm may return three possible values depending on whether the property is *definitely satisfied* (T), *possibly satisfied* (?) or *not satisfied* (F).

The inductive (three-valued) semantic function $\|\mathcal{M}^\phi\|$ associates to $\mathcal{M}$ and ϕ one of the true values true (T), false (F) and unknown (?)

Definition 4. *Given an IBA $\mathcal{M}$ and an LTL formula ϕ:*

1. $\|\mathcal{M}^\phi\| = T \Leftrightarrow (\forall v \in (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}_p^\omega(\mathcal{M})),\ v \models \phi)$;
2. $\|\mathcal{M}^\phi\| = F \Leftrightarrow (\exists v \in \mathcal{L}^\omega(\mathcal{M}) \mid v \not\models \phi)$;
3. $\|\mathcal{M}^\phi\| = ? \Leftrightarrow ((\forall v \in \mathcal{L}^\omega(\mathcal{M}),\ v \models \phi) \wedge (\exists u \in \mathcal{L}_p^\omega(\mathcal{M}) \mid u \not\models \phi))$.

Case 1 specifies that ϕ is true in $\mathcal{M}$ iff every word v that is in the language definitely or possibly accepted by $\mathcal{M}$ satisfies ϕ. Case 2 specifies that ϕ is false in $\mathcal{M}$ iff a word v that is in the language definitely accepted by the IBA exists and v does not satisfy ϕ. Case 3 specifies that ϕ is possibly satisfied in $\mathcal{M}$ iff there exists a word u that is in the language possibly accepted by the IBA that does not satisfy ϕ and all the words v in the language definitely accepted by $\mathcal{M}$ satisfy ϕ. For example, the property $\phi = G(send \rightarrow F(success))$ is possibly satisfied by the model described in Fig. 1 since there exists a word $\{start\}.\{send\}.\{fail\}.\{fail\}.\{abort\}^\omega$ in the possibly accepted language which does not satisfy ϕ and there are no words in the definitely accepted language. It is possible to specify the satisfaction of an LTL formula ϕ with respect to $\mathcal{M}$ using the BA $\mathcal{A}_\phi$ equivalent to ϕ.

Definition 5. *Given an IBA $\mathcal{M}$ and a BA $\mathcal{A}_\phi$,*

1. $\|\mathcal{M}^{\mathcal{A}_\phi}\| = T \Leftrightarrow (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}_p^\omega(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{A}_\phi))$;
2. $\|\mathcal{M}^{\mathcal{A}_\phi}\| = F \Leftrightarrow (\mathcal{L}^\omega(\mathcal{M}) \not\subseteq \mathcal{L}^\omega(\mathcal{A}_\phi))$;
3. $\|\mathcal{M}^{\mathcal{A}_\phi}\| = ? \Leftrightarrow ((\mathcal{L}^\omega(\mathcal{M}) \subseteq \mathcal{L}^\omega(\mathcal{A}_\phi)) \wedge (\mathcal{L}_p^\omega(\mathcal{M}) \not\subseteq \mathcal{L}^\omega(\mathcal{A}_\phi)))$.

Based on Definition 5 the automata-based model checking procedure is composed by the following six steps.

(1) *Create the automaton $\mathcal{A}_{\neg\phi}$*: as in the classical approach, first we build the BA that contains the set of behaviors forbidden by property ϕ. The complexity of this step is $\mathcal{O}(2^{(|\neg\phi|)})$. The BA corresponding to $\neg\phi$ is presented in Fig. 4.

(2) *Extract the completion automaton* $\mathcal{M}_c$ which contains the definitely accepting behaviors of $\mathcal{M}$. Computing $\mathcal{M}_c$ has in the worst case complexity $\mathcal{O}(|Q_{\mathcal{M}}|+|\Delta_{\mathcal{M}}|)$ since it is sufficient to visit all the states of the automaton, and, for each box s, remove its incoming and outgoing transitions and the state s itself. In the example, the BA $\mathcal{M}_c$, associated with $\mathcal{M}$ contains the states q_1, q_2 and q_3 and the transitions 6 and 7.

(3) *Build the intersection automaton* $\mathcal{I}_c = \mathcal{M}_c \cap \mathcal{A}_{\neg\phi}$: $\mathcal{I}_c$ contains in the worst case $3 \cdot |R_{\mathcal{M}}| \cdot |Q_{\mathcal{A}_{\neg\phi}}|$ states and describes the behaviors of $\mathcal{M}_c$ that violate the property. The intersection between $\mathcal{M}_c$ associated with the model $\mathcal{M}$ described in Fig. 1 and the automaton $\mathcal{A}_{\neg\phi}$ described in Fig. 4 contains all the behaviors of the sending message system that violate the property.

(4) *Check the emptiness of the intersection automaton* $\mathcal{I}_c$: if $\mathcal{I}_c$ is not empty, the condition $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\mathcal{A}_{\neg\phi}) \neq \emptyset$ holds, i.e., the property is not satisfied and every infinite word in the intersection automaton is a counterexample. If, instead, $\mathcal{I}_c$ is empty, $\mathcal{M}$ possibly satisfies or definitely satisfies ϕ depending on the results of the next steps of the algorithm. The intersection automaton $\mathcal{I}_c$ of the motivating example is empty. Indeed, both q_2 and q_3, accepting states of $\mathcal{M}_c$, are never reachable from q_1. Thus, $\mathcal{M}$ definitely satisfies or possibly satisfies ϕ depending on the next steps of the algorithm.

(5) *Compute the intersection* $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_{\neg\phi}$ *of the incomplete model* $\mathcal{M}$ and the automaton $\mathcal{A}_{\neg\phi}$ associated with the property ϕ: to check whether $\mathcal{M}$ definitely satisfies or possibly satisfies ϕ, it is necessary to verify whether $(\mathcal{L}(\mathcal{M}) \cup \mathcal{L}_p(\mathcal{M})) \cap \mathcal{L}(\mathcal{A}_{\neg\phi}) = \emptyset$. We propose a new algorithm to compute $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_{\neg\phi}$ when $\mathcal{M}$ is incomplete. The intersection automaton $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_{\neg\phi}$ between an IBA $\mathcal{M}$ and a BA $\mathcal{A}_{\neg\phi}$ is a BA $\langle \Sigma_{\mathcal{I}}, Q_{\mathcal{I}}, \Delta_{\mathcal{I}}, Q^0_{\mathcal{I}}, F_{\mathcal{I}} \rangle$ defined as follows:

- $\Sigma_{\mathcal{I}} = \Sigma_{\mathcal{M}} \cup \Sigma_{\mathcal{A}_{\neg\phi}}$ is the alphabet of $\mathcal{I}$;
- $Q_{\mathcal{I}} = ((R_{\mathcal{M}} \times R_{\mathcal{A}_{\neg\phi}}) \cup (B_{\mathcal{M}} \times R_{\mathcal{A}_{\neg\phi}})) \times \{0,\ 1,\ 2\}$ is the set of states;
- $\Delta_{\mathcal{I}} = \Delta^c_{\mathcal{I}} \cup \Delta^p_{\mathcal{I}}$. $\Delta^c_{\mathcal{I}}$ is the set of transitions $(\langle q_i, q'_j, x\rangle, a,\ \langle q_m, q'_n, y\rangle)$ where $(q_i, a, q_m) \in \Delta_{\mathcal{M}}$ and $(q'_j, a, q'_n) \in \Delta_{\mathcal{A}_{\neg\phi}}$. $\Delta^p_{\mathcal{I}}$ corresponds to the set of transitions $(\langle q_i, q'_j, x\rangle, a,\ \langle q_m, q'_n, y\rangle)$ where $q_i = q_m$ and $q_i \in B_{\mathcal{M}}$ and $(q'_j, a, q'_n) \in \Delta_{\mathcal{A}_{\neg\phi}}$. Moreover, each transition in $\Delta_{\mathcal{I}}$ must satisfy the following conditions:
  - if $x = 0$ and $q_m \in F_{\mathcal{M}}$, then $y = 1$.
  - if $x = 1$ and $q'_n \in F_{\mathcal{A}_{\neg\phi}}$, then $y = 2$.
  - if $x = 2$ then $y = 0$.
  - otherwise, $y = x$;
- $Q^0_{\mathcal{I}} = Q^0_{\mathcal{M}} \times Q^0_{\mathcal{A}_{\neg\phi}} \times \{0\}$ is the set of initial states;
- $F_{\mathcal{I}} = F_{\mathcal{M}} \times F_{\mathcal{A}_{\neg\phi}} \times \{2\}$ is the set of accepting states.

The intersection $\mathcal{I}$ between the model $\mathcal{M}$ depicted in Fig. 1 and the BA $\mathcal{A}_{\neg\phi}$ of Fig. 4 that corresponds to the negation of the property is the BA described in Fig. 5. The set of states $Q_{\mathcal{I}}$ is composed by states obtained by combining states of the automaton associated with the negation of the property $\mathcal{A}_{\neg\phi}$ with regular states and boxes of the model $\mathcal{M}$. We define $M_{\mathcal{I}} = B_{\mathcal{M}} \times R_{\mathcal{A}_{\neg\phi}} \times \{0,\ 1,\ 2\}$ as the set of *mixed* states (indicated in Fig. 5 with

a stipple border). The portion of the state space that contains mixed states associated with the states of the model $send_1$ and $send_2$ are surrounded by a dashed-dotted frame. $PR_{\mathcal{I}} = R_{\mathcal{M}} \times R_{\mathcal{A}_{\neg\phi}} \times \{0,\ 1,\ 2\}$ is the set of *purely regular* states. For example, state ① is obtained by combining states q_1 of $\mathcal{M}$ and p_1 of $\mathcal{A}_{\neg\phi}$. This state is initial and purely regular since both q_1 and p_1 are initial and regular. State ② is mixed since it is obtained by combining the box $send_1$ of $\mathcal{M}$ and the state p_1 of $\mathcal{A}_{\neg\phi}$. As for classical intersection of BAs [11], labels 0, 1 and 2 indicate that no accepting state is entered, at least one accepting state of $\mathcal{M}$ is entered, and at least one accepting state of $\mathcal{M}$ and one accepting state of $\mathcal{A}_{\neg\phi}$ are entered, respectively.

The transitions in $\Delta^c_{\mathcal{I}}$ are obtained by the synchronous execution of transitions of $\mathcal{M}$ and $\mathcal{A}_{\neg\phi}$. For example, the transition from ② to ③ is obtained combining the transitions -2- of $\mathcal{M}$ and -1- of $\mathcal{A}_{\neg\phi}$. The transitions in $\Delta^p_{\mathcal{I}}$, graphically indicated through dashed lines in Fig. 5, are obtained when a transition of $\mathcal{A}_{\neg\phi}$ synchronizes with a transition in the replacement of a box of $\mathcal{M}$. For example, the transition from ② to ⑥ is generated when $\mathcal{A}_{\neg\phi}$ and $\mathcal{M}$ perform the transition -2- and a transition inside the box $send_1$, respectively.

The intersection $\mathcal{I}$ contains in the worst case $3 \cdot |Q_{\mathcal{M}}| \cdot |Q_{\mathcal{A}_{\neg\phi}}|$ states.

(6) *Check the emptiness of the intersection automaton* $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_{\neg\phi}$: by checking the emptiness of the automaton $\mathcal{I}$ we verify whether the property ϕ is definitely satisfied or possibly satisfied by $\mathcal{M}$. Since we have already checked that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{A}_{\neg\phi})$, two cases are possible: if $\mathcal{I}$ is empty, $\mathcal{L}_p(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{A}_{\neg\phi})$ and the property is definitely satisfied whatever refinement is proposed for the boxes of $\mathcal{M}$, otherwise, $\mathcal{L}_p(\mathcal{M}) \not\subseteq \mathcal{L}(\mathcal{A}_{\neg\phi})$, meaning that there exists some refinement of $\mathcal{M}$ that violates the property. For example, the run $start.(send) \wedge (!success).fail.fail.abort^{\omega}$, which is a possible run of $\mathcal{M}$, violates ϕ since there exists a path where a *send* is not followed by a *success*. This behavior can be generated by replacing boxes $send_1$ and $send_2$ with a component that allows paths where a message is *sent* and no *success* is obtained, and an empty component that neither tries to *send* the message again nor waits for a *success*, respectively.

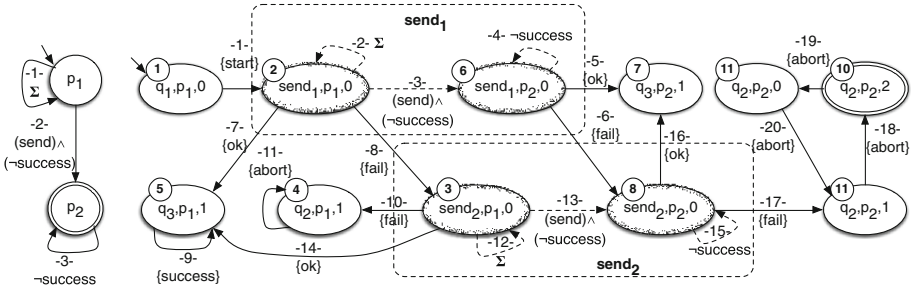


Fig. 4. $\mathcal{A}_{\neg\phi}$.

Fig. 5. $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_{\neg\phi}$.

5 Constraint Computation and Replacement Checking

When a property ϕ is possibly satisfied, each word v that is recognized by the intersection automaton $\mathcal{I}$ corresponds to a behavior $\mathcal{B}$ the system may exhibit that violates ϕ. To satisfy ϕ, the developer must design the replacements of the boxes to forbid $\mathcal{B}$ from occurring. This section shows how to decompose the global information described by $\mathcal{B}$ into local constraints for boxes that become proof obligations for their replacements. For clarity and reasons of space, the section will give an informal, but precise, description of the process; all the formal details can be found in the report published at [25].

A local constraint C for a box b is a pair $\langle S, S_p \rangle$, where S and S_p are two *sub-properties* encoding the replacements of b that make ϕ not satisfied or possibly satisfied, respectively. Hereafter we will first assume that a box constraint only includes a sub-property S that specifies all behaviors that would lead to violating ϕ. This happens, for example, in the case where $\mathcal{M}$ contains *only one box*. Intuitively the reason is that when there are more boxes, the violating behavior may be caused by the "collaboration" of multiple boxes of the model. We will briefly describe the case with multiple boxes at the end of the section.

To compute the constraint C for the box b we need to identify the behaviors described by $\mathcal{I}$ that are recognized by the automaton and traverse the box. To do that, we first perform a cleaning step that eliminates all states from which an accepting state that can be entered infinitely many often is not reachable. This is done with a procedure similar to [7,19]. The resulting automaton will be $\mathcal{I}_{cl}$.

The sub-property S can be computed by first extracting from $\mathcal{I}_{cl}$ the fragment (called $\mathcal{P}$) where states are in the form $\langle b, -, - \rangle$ and then decorating the fragment with additional information, namely:

1. the set Δ^{inS} of incoming transitions that lead to $\mathcal{P}$ from the other states of $\mathcal{I}_{cl}$;
2. the set Δ^{outS} of outgoing transitions that lead from $\mathcal{P}$ to the other states of $\mathcal{I}_{cl}$;
3. the subset G of the source states of Δ^{inS} that are reachable in $\mathcal{I}_{cl}$ from one of its initial states without traversing any other state in the form $\langle b, -, - \rangle$;
4. the subset R of target states of transitions in Δ^{outS} such that an accepting state of $\mathcal{I}_{cl}$ can be reached without traversing any other state in the form $\langle b, -, - \rangle$;
5. a relation K between the target states of Δ^{outS} and the source states of Δ^{inS}. A pair $\langle s_1, s_2 \rangle$ exists in K iff state s_2 is reachable from s_1 by a path of $\mathcal{I}_{cl}$ that does not traverse any state in the form $\langle b, -, - \rangle$.

For example, the sub-property described in Fig. 7 is derived from the clean intersection automaton of Fig. 6 resulting from the model in Fig. 3 and the property in Fig. 4. The automaton $\mathcal{P}$ in Fig. 7 contains the states ② and ⑥ and the transitions -2-, -3- and -4. The transition -1- (-6-) is the incoming (outgoing) transition of $\mathcal{S}$ and its source (target) state is also contained in the set G (R). Because the source and destination states of the incoming and outgoing

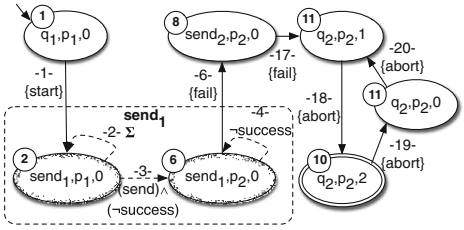


Fig. 6. The intersection $\mathcal{I}_{cl}$.

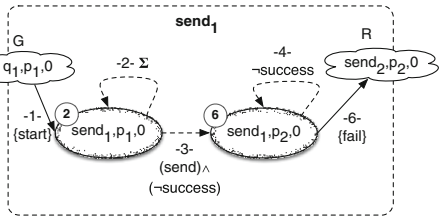


Fig. 7. The sub-property S.

transitions are states of $\mathcal{I}_{cl}$ that do not belong to $\mathcal{P}$, we graphically represent them with a cloud shape on the outer frame that encloses $\mathcal{P}$ (see Fig. 7, where the source state of -1- is indicated with the triple $\langle q_1, p_1, 0\rangle$). Cloud states are also marked G or R if they belong to G or R, respectively. Cloud states on the frame that encloses a replacement are also used to indicate how the replacement is connected to the states of the embedding IBA (see Fig. 8). Intuitively, every path of that connects a G-marked cloud state with a R-marked cloud state of the sub-property is a behavior the replacement of the box $send_1$ should not exhibit. Such behavior would enable a violation of the original property. In our example K is empty, since there is no path from ⑧ to ①.

Suppose now that the designer proceeds in the top-down decomposition producing a replacement $\mathcal{R}$ for a box b and wants to check if ϕ is satisfied by the new design, e.g., she/he proposes the replacement of Fig. 8. One possible option would be *refinement checking*. The refinement checking procedure generates the refinement $\mathcal{N}$ of $\mathcal{M}$ by replacing b with $\mathcal{R}$ and checks $\mathcal{N}$ against ϕ. By following this approach, the entire model would be verified at each refinement round. This also implies that the verification needs to be performed by a party which knows the whole system and cannot be delegated to a third party which a partial view of the system. The reason for computing a constraint C for box b has instead the goal of enabling *replacement checking*, in which the verification is applied only on the replacement $\mathcal{R}$ (a small fragment of the model) by checking it against the previously generated constraint C.

To check whether a replacement *violates* a constraint, we build the intersection $\mathcal{U}$ of the automaton $\mathcal{T}$ (of the replacement) and $\mathcal{P}$ (of the sub-property). In doing so we ignore incoming and outgoing transitions. Because we are computing violating behaviors, boxes and their incoming and outgoing transitions are removed from $\mathcal{T}$. If the replacement contains boxes, the same procedure we describe is repeated without removing boxes to find possibly violating behaviors. The intersection $\mathcal{U}$ is shown in Fig. 10.

An additional initial state g and an additional accepting state r, with a self loop, are added to $\mathcal{U}$. States g and r are connected to states of $\mathcal{U}$ as follows. If there is an incoming (outgoing) transition $\langle q, l, q'\rangle$ ($\langle q'', l, q'''\rangle$) to (of) the replacement and an incoming (outgoing) transition $\langle\langle q, p, -\rangle, l, \langle b, p', -\rangle\rangle$ ($\langle\langle b, p'', -\rangle, l, \langle q''', p''', -\rangle\rangle$) to (of) the sub-property originating in a state in G (R), then a transition labeled l is added to $\mathcal{U}$ to connect state g

$(\langle q'', \langle b, p'', -\rangle, -\rangle)$ to each of the states labeled $\langle q', \langle b, p', -\rangle, 0\rangle$ (r). In our example, transition -10- is added to $\mathcal{U}$ since incoming transitions -1- and -7- are both labeled *start*, the source of -1- is in G and it is obtained from the state q_1. Transition -13- is added to $\mathcal{U}$ since both outgoing transitions of the replacement and of the sub-property are labeled $fail$, the destination of the transition -6- is in R and it is obtained from the state $send_2$ of the model.

In general, $\mathcal{U}$ must be further enriched by relation K to include all the paths in the intersection automaton which allow reaching the source of an incoming transition of the constraint from the destination of an outgoing transition. This does not happen for our example, because the relation is empty. If a tuple exists in K, a violating run recognized by $\mathcal{I}_{cl}$ exits a target state of one of its outgoing transitions and enters the source state of one of its incoming transitions.

Once the automaton $\mathcal{U}$ has been built, we can check whether the replacement violates the constraint by checking emptiness. If $\mathcal{U}$ is not empty, the constraint, and therefore also the property of interest, is violated. If the constraint is not violated, the same procedure can check possibly violating behaviors by considering $\mathcal{T}$ without removing boxes. If neither violating nor possibly violating behaviors are present, the property is definitely satisfied.

When an IBA $\mathcal{M}$ has to satisfy the property ϕ and contains multiple boxes, the constraint C computed for the box b may have both components: S and S_p. S specifies the behaviors rendered by $\mathcal{R}$ that would violate ϕ, and S_p specifies the possibly violating behaviors. S is computed as explained above, but, since there are multiple boxes, in the computation of G, R and K it is necessary to not traverse *any* state in the form $\langle b_i, -, -\rangle$, where b_i is a box of $\mathcal{M}$. In fact, when *violating* behaviors are considered, only states of the intersection automata which are not obtained from boxes can be traversed. For example, when the model of Fig. 1 and the property ϕ of Fig. 4 are evaluated, the sub-property S associated with $send_1$ (extracted from the intersection $\mathcal{I}$ presented in Fig. 5) does not contain any outgoing transition in R. Furthermore, the automaton $\mathcal{P}$ does not contain any accepting state. For this reason, no violating behaviors can be exhibited by the replacement. Thus, the replacement either definitely satisfies or possibly satisfies the constraint.

S_p is obtained similarly to S. However, possibly violating behaviors may also traverse states in the form $\langle b_i, -, -\rangle$, where b_i is a box of $\mathcal{M}$ different from b. Figure 9 shows the sub-property S_p for $send_1$ associated with the model of Fig. 1 –which contains multiple boxes– and the property ϕ of Fig. 4. S_p is extracted from the intersection automaton $\mathcal{I}$ presented in Fig. 5. Two types of possibly accepting runs are present: the ones that cross the component and leave S_p through the outgoing transition -8-, and behaviors in which a sending activity is performed and no *success* is obtained and the component is left by firing -6-. In the first case, there is no assurance the replacement of $send_2$ will guarantee a success after any send. In the second case, since the component is left after the execution of a sending activity, $send_2$ must wait for a success.

The replacement checker detects possibly violating behaviors by considering the replacement $\mathcal{R}$ against the sub-property S_p. In computing the intersection

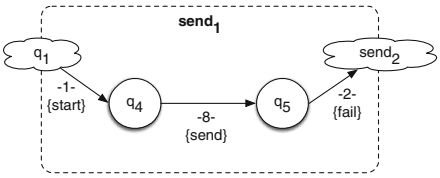


Fig. 8. Replacement $\mathcal{R}$.

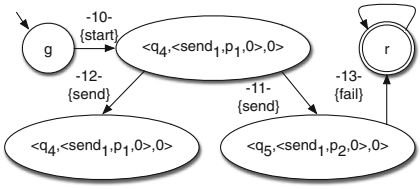


Fig. 10. Intersection $\mathcal{U}$.

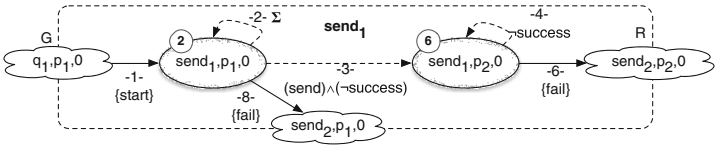


Fig. 9. Sub-property S_p.

automaton, the boxes of the automaton $\mathcal{T}$ of $\mathcal{R}$ are also considered (if present). For example, the intersection between the sub-property S_p of Fig. 9 and the replacement shown in Fig. 8, is presented in Fig. 10. Since the language accepted by the automaton is not empty, the replacement possibly satisfies the property. Note that, when multiple boxes are present, a possibly violating behavior may exist over which the box b has no control (this situation can be verified during the constraint computation). In this case, even if the intersection between the replacement and the property is empty, ϕ is possibly violated. If the sub-property S_p has no incoming transitions in G and there exists a possibly violating behavior of the system over which the box b has no control, the replacement checking procedure returns that the property is possibly satisfied in constant time.

Theorem 1. *The constraint computation procedure has a* $\mathcal{O}(|Q_{\mathcal{I}}|^3|)$ *temporal complexity. The replacement checking complexity is* $\mathcal{O}(|Q_{\mathcal{T}}| \cdot |Q_{\mathcal{P}}| + |\Delta_{\mathcal{T}}| \cdot |\Delta_{\mathcal{P}}| + |\Delta^{inR}| \cdot |\Delta^{inS}| + |\Delta^{outR}| \cdot |\Delta^{outS}| + (|\Delta^{outS}| \cdot |\Delta^{inS}|) \cdot (|\Delta^{outR}| \cdot |\Delta^{inR}|))$

The time complexity of computing constraints is due to the complexity of calculating the relation K, which requires computing the reachability relation between all the pairs of states in the automaton. In the replacement checking complexity the first part concerns the computation of the intersection of the replacement $\mathcal{T}$ and $\mathcal{P}$ (associated with the sub-property), the term $|\Delta^{inR}| \cdot |\Delta^{inS}| + |\Delta^{outR}| \cdot |\Delta^{outS}|$ concerns the computation of the transitions in the intersection automaton generated analyzing the incoming and outgoing transitions of the replacement and the sub-property, and the last part is due to the use of K in the replacement checking.

6 Evaluation

The proposed approach is evaluated through an empirical study that aims to assess its feasibility and scalability, by answering the following questions:

RQ1: How **feasible** is reasoning with IBAs with respect to BAs?
RQ2: How **effective** is replacement checking with respect to refinement checking?

To answer RQ1 and RQ2, we set up experiments based on random model generation, as done in [16,33,34,38]. We considered four tasks:

T1: we check fully refined, complete BAs $\mathcal{N}$ against selected LTL properties;
T2: for each BA $\mathcal{N}$ of task T1, we generate an IBA $\mathcal{M}$ of which it is a refinement. IBAs are verified against the same LTL properties;
T3: for all IBAs and LTL properties, we consider the IBAs that possibly satisfy one of the properties. For each IBA $\mathcal{M}$, we consider the replacement $\mathcal{R}$ of the box b that was abstracted into the box b by task T2. We refine $\mathcal{M}$ by expanding box b with the replacement $\mathcal{R}$ and verify the resulting IBA against ϕ;
T4: we evaluate replacements against their (previously computed) constraints.

To answer RQ1, we conducted two experiments: **(E1)** compares performance and results of T1 and T2; **(E2)** given a $\mathcal{M}$ obtained from $\mathcal{N}$ that possibly satisfies ϕ, considers performance and results of T2 and T3. To answer RQ2, we conducted one experiment: **(E3)** given a $\mathcal{M}$ that possibly satisfies ϕ and the replacement $\mathcal{R}$ of its box b, compares performance and results of T3 and T4.

The evaluation is based on the `CHIA` (CHecker for Incomplete Automata) prototype tool[1], a Java 7 stand-alone application. CHIA has been developed as a proof of concept and does not aim at competing with state of the art model checking tools.

Experimental inputs. The random generation of IBAs is based on the procedure presented in [34] (also used in [33,38]). BAs are generated over an alphabet of two propositions. For each proposition, a directed graph with a single initial state and k transitions is randomly created. The "hardness" of the problem is changed by controlling: 1. the number of states; ($|Q|$); 2. the density of the transitions, i.e., the ratio between the number of transitions per proposition p and the number of states; 3. the density of accepting states, i.e., the ratio between the accepting and total state number. To avoid trivial automata, the initial state is associated with an outgoing transition for each proposition of the alphabet. The BAs that are generated as explained above are also used to further generate the IBAs and replacements needed by tasks T2, T3 and T4. This is done by randomly abstracting fragments of the BAs into boxes. The automata fragments and the corresponding incoming and outgoing transitions are used as replacements associated with the boxes. The box density is used to compute the number of boxes to be injected in the IBA. The replacement density specifies the number of states of the BA to be encapsulated into these boxes. Table 1 presents the values of the parameters used in the scalability assessment.

The reported experiments used three properties randomly selected from the Büchi Store [36]: $\phi_1 = F(a \rightarrow Fb)$, $\phi_2 = G(a \rightarrow F(a \wedge Fb))$ and $\phi_3 = F(a \wedge X(a \wedge Xa)) \wedge F(b \wedge X(b \wedge Xb))$ which have one, three and six temporal operators and correspond to a BA of size 10, 28 and 41, respectively.

[1] https://github.com/claudiomenghi/CHIA.

Table 1. Parameters values.

Parameter	N of states	Transition density	Accepting density	Box density	Replacement density
Initial	100	1.0	0.1	0.1	0.1
Increment	100	1.0	0.2	0.2	0.2
Final	1000	3.0	0.5	0.5	0.5

Table 2. E1 and E2 verification results.

	E1					E2			
	T1	T2	ϕ_1	ϕ_2	ϕ_3	T3	ϕ_1	ϕ_2	ϕ_3
C1	T	T	42,7	42,8	48,0	T	0	1,8	1,9
C2	T	?	57,3	57,2	52,0	?	100	98,2	98,1
C3	F	F	55,5	55,9	55,6	F	2	1,8	1,8
C4	F	?	44,5	44,1	44,4	?	98	98,2	98,2

Table 3. $E3$ and $E3$b verification result.

	$E3$					$E3$b			
	T3	T4	ϕ_1	ϕ_2	ϕ_3	ϕ_1	ϕ_2	ϕ_3	
C1	T	T	0,1	0	0,1	1,7	1,9	1,8	
C2	?	?	96,3	96,3	96,3	-	-	-	
C3	F	F	3,6	3,7	3,6	98,3	98,1	98,2	

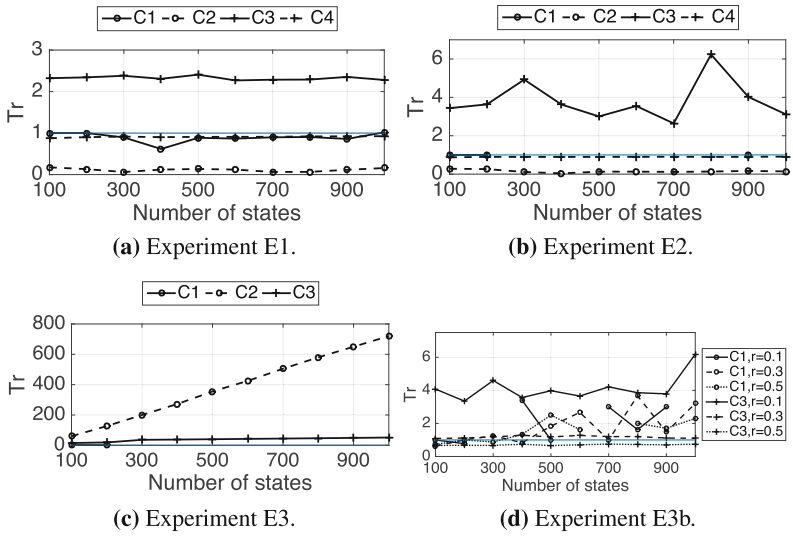
Each experiment is composed by a set of tests, which randomly generate a model, select a property, and perform the corresponding verification. The tests are performed 20 times to reduce biases due to the random generation.

Results. *E1.* Table 2 reports the results. Lines 1 and 2 (3 and 4) report data in which T1 returns T (F). The first line shows that both BAs (Column $T1$) and IBAs (Columns $T2$) return T 42,7 %, 42,8 % and 48,0 % of the cases for properties ϕ_1, ϕ_2 and ϕ_3, respectively. Likewise, line 3 shows that both BAs and IBAs return a value F in 55,5 %, 55,9 % and 55,6 % of the cases for ϕ_1, ϕ_2 and ϕ_3, respectively. Thus, in almost half of the cases the developer does not need to wait until the end of the development process to know if the property is definitely satisfied or violated.

Figure 11a shows the average ratio T_r between the verification time of BAs (computed by task $T1$) and the verification time of the corresponding IBAs (computed by task $T2$) with respect to the size of the automaton. As expected, IBA verification performs better when properties are violated both by the BA and the IBA, and worse in the case that the property is verified by the BA, while the result is unknown for the corresponding IBA. Performance results are similar for the other cases.

E2. Table 2 reports the results. Given that $T1$ has returned T, $T3$ returns T in 0 %, 1,8 % and 1,9 % of the cases for ϕ_1, ϕ_2 and ϕ_3, respectively. In this case the developer does not need to proceed with the refinement to infer that the property is satisfied. Likewise, when verification for the BA returns F. Task T3 returns F in 2 %, 1,8 % and 1,8 % of the cases for the properties ϕ_1, ϕ_2 and ϕ_3, respectively. In these cases, the developer has to fix her/his current design before proceeding in the refinement.

Figure 11b shows the average ratio T_r between the verification time of the BAs (computed by task $T1$) and the verification time of IBAs obtained by task $T3$ with respect to the size of the automaton. As expected, verification is faster



(a) Experiment E1. **(b)** Experiment E2.

(c) Experiment E3. **(d)** Experiment E3b.

Fig. 11. Experiments results

for IBAs in case C3, and it is slower in case C2. Performance results are similar otherwise.

E3. As showed in Table 3, in all the considered cases, T3 and T4 return the same values, confirming correctness of the approach. The percentage of cases in which the verification result is unknown is high because the replacement is only applied to one box out of many others that may still be present in the refinement.

Figure 11c shows the average ratio T_r between the verification of the IBA after replacement and verification of the replacement against the constraint with respect to the number of the states of the IBA. In case $C2$ the ratio grows linearly, thus indicating that checking the replacement against the constraint is better that checking the refined IBA. Indeed, as the number of boxes increases, there is a high chance that the box b whose replacement $\mathcal{R}$ is considered, is only reachable from the initial state by traversing another box, and there exists another possibly accepting run in the intersection not involving b. Thus, replacement verification is performed in constant time: whatever replacement the developer proposes for b the property remains possibly satisfied. However, this is a favorable situation. For this reason, we refine the experiment ($E3$b) to consider the case in which the starting IBA contains only one box and therefore the replacement refines it to a complete BA.

E3b. In this setting, after a replacement is plugged into the box properties are either satisfied or not, as shown in Table 3, which summarizes our results. Figure 11d compares the performance of T3 and T4 with respect to the states of $\mathcal{M}$. We plot the data for three different values of the replacement density r. As the number of states in the replacement gets smaller with respect to the

total number of states, verifying the replacement against the constraint performs better than verifying the refinement.

Conclusions. *E1* and *E2* demonstrate the advantages of reasoning with IBAs. There are no remarkable performance differences between the verification of IBAs and BAs. Furthermore, even if in some cases reasoning with IBAs implies an overhead, the proposed model checking procedure provides considerable benefits, allowing an earlier detection of design flaws. *E3* and *E3b* show that the speedup of checking replacements against checking the flattened refinement is generally considerable. This is in particular true when several boxes remain in the refinement and when the replacement that makes the refinement complete is small in size with respect to the complete automaton.

Threats to Validity. The most important threat to validity concerns the random generation of experimental inputs (IBAs and replacements). There is no guarantee that either the random generation procedure or the values of the parameters chosen in the generation reflect real-life examples. Furthermore, there is no assurance about the significance of the formulae, randomly selected from the Büchi Store [36], over the generated automata. To compensate for these threats, we have complemented the assessment with an analysis of two real world applications reported in [24].

7 Related Work

Modeling incompleteness. Many modeling formalisms, such as MTSs [23], PKSs [4], $\mathcal{X}$KSs [9] and LTS$^{\uparrow}$ [19], support the specification of incompleteness. These formalisms can be used in a top-down, hierarchical development process, but they have not been explicitly proposed with this purpose. MTSs represent incompleteness/uncertainty using maybe transitions, i.e., transitions that can be present or not in the final design of the system, while LTS$^{\uparrow}$ express the behavior of systems that are executed in an unknown environment. Differently, IBAs represents unspecified parts by means of black box states, i.e., states that can be refined in other state machines. In this sense IBAs are similar to HSMs [2] which have been proposed to model sequential processes when a top-down development process is used. However, HSMs can only be analyzed at the end of the development process when a fully specified model of the system is produced. Other formalisms, such as [16–18], support uncertainty i.e., they associate incomplete parts with a set of possible replacements. Finally, models, such as Featured Transition Systems (FSTs) [13], used in variability modeling, are also related. In variability modeling the goal is to represent a large family of different systems efficiently. Differently from IBAs in variability models the replacements (or variants) are known upfront.

Checking incomplete models. The verification procedure discussed in Sect. 4 is similar to others proposed in literature in which properties are expressed as LTL formulae or automata. The procedure was designed considering the three value

inductive semantics of LTL. In the three valued semantics, when the property is possibly satisfied, there is no assurance on the existence of two refinements such that the first satisfies and the second violates the formula. This differentiate our work from others that consider the thorough semantics, e.g. [5]. The three valued semantics has been considered in the context of PKSs [4], MTSs [20,21, 23,37] and $\mathcal{X}$KSs [9]. Differently from these works, our procedure supports the verification of automata when the behavior of the system inside a set of states (black boxes) is currently unknown. [19] describes how to verify LTSs (LTS$^{\uparrow}$) when they are executed in an unknown environment. [2] proposes a technique to check HSMs. However, the proposed technique can only be executed when the whole behavior of the system is specified. Verification of variability-intensive systems aims at checking whether all the products of a family satisfy a property of interest. It has been proposed for example in [12,35], where MTSs and FSTs are considered, respectively.

Constraint computation. The constraint computation problem is similar to other problems, such as synthesis and supervisory control. In program synthesis [15,28] the developer usually computes a model of the system that satisfies the properties of interest. Differently, our goal is to compute sub-properties for the unspecified parts. In this sense, the addressed problem is more similar to assumption generation [19]. In [19], the authors, given a model of the system $\mathcal{M}$ which contains a set of controllable actions, compute an assumption for its environment. If the environment satisfies the assumption, when it is executed in parallel with $\mathcal{M}$ guarantees the satisfaction of the property of interest. Differently, our approach tries to compute assumptions for the unknown components of the system. The constraint computation can be interpreted also as a supervisory control problem [7,26,29,30] in which each box is associated with a set of controllable actions. The problem is to synthesize a strategy the controller can employ to modify the behavior of the incomplete model $\mathcal{M}$ in its boxes to satisfy the properties of interest. In this sense, supervisory control is more similar to the assumption generation problem [19]. Finally, in [37], the authors propose a synthesis technique that constructs MTSs from a combination of safety properties and scenarios but without considering the problem of constraining unspecified parts.

Replacement checking. The replacement checking goal is to verify, after a change, the portion of the state space affected by the change. Thus, problems such as compositional reasoning, component substitutability and hierarchical model checking are related to our work. Compositional reasoning [14] reduces the verification effort by verifying properties on individual components and inferring the properties that hold in the global system without its explicit creation. For example, in the assume-guarantee paradigm [1,22,27], if M guarantees ϕ and M' guarantees ψ when it is located in an environment that satisfies ϕ, then, when M and M' are executed in parallel, they satisfy ψ. In this framework, our constraint can be interpreted as a post-condition that a component has to guarantee. The replacement checking can be considered as a procedure used to verify whether the component ensures its post-condition. Component

substitutability [8,32] considers the verification of a system when a component is removed from the system and replaced by a new one. The checker verifies whether the new component preserves the behaviors provided by the old one. A constraint can be interpreted as the "most general" component that ensures the properties of interest. Incremental verification [31] is a technique to efficiently verify code by focusing on the differences between the current and the previous version. Differently from replacement checking, it requires the whole system to run and, hence, is not tailored for distributed design.

8 Conclusion and Future Work

This paper presented an automata-based model checking algorithm that verifies whether an incomplete model of the system definitely satisfies, possibly satisfies or does not satisfy its requirements. If the specification is possibly satisfied, a constraint on the unspecified parts is computed. Whenever an unspecified part is refined, i.e., a replacement is proposed, it is verified in isolation against the previously computed constraint. We provided the theoretical background behind our framework and evaluated its feasibility and effectiveness using a set of random generated models. The presented approach is a step toward the integration of formal verification in modern development processes.

Evaluating the approach on a realistic case study is one of our future goals. Moreover, we also plan to realize a solid, efficient tool designed upon existing symbolic model checkers, and integrate it in commonly used IDEs. Finally, we also aim to analyze the benefits of applying our approach in a distribute development environment, where the refinement of some boxes is delegated to third parties.

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst. Des. **15**(1), 7–48 (1999)
2. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3), 273–303 (2001)
3. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999). doi:10.1007/3-540-48683-6_25
5. Bruns, G., Godefroid, P.: Generalized model checking: reasoning about partial state spaces. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 168–182. Springer, Heidelberg (2000). doi:10.1007/3-540-44618-4_14
6. Büchi, J.R.: Symposium on decision problems: On a decision method in restricted second order arithmetic. Stud. Logic Found. Math. **44**, 1–11 (1966)
7. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems, 2nd edn. Springer, New York (2008)
8. Chaki, S., Sharygina, N., Sinha, N.: Verification of evolving software. In: Specification and Verification of Component-Based Systems, SAVCBS (2004)

9. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. Trans. Softw. Eng. Methodol. (TOSEM) **12**(4), 371–408 (2003)
10. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. Trans. Program. Lang. Syst. (TOPLAS) **8**(2), 244–263 (1986)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Softw. Eng. **39**(8), 1069–1089 (2013)
13. Classen, A., Heymans, P., Schobbens, P., Legay, A., Raskin, J.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: International Conference on Software Engineering, pp. 335–344. ACM (2010)
14. de Roever, W.-P., de Boer, F., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J., Verification, C.: Introduction to Compositional and Non-compositional Methods. Cambridge University Press, Cambridge (2012)
15. D'ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesizing non-anomalous event-based controllers for liveness goals. Trans. Softw. Eng. Method. (TOSEM) **22**(1), 9:1–9:36 (2013). Article no. 9
16. Famelis, M., Salay, R., Chechik, M.: Partial models: towards modeling and reasoning with uncertainty. In: International Conference on Software Engineering, ICSE, pp. 573–583. IEEE Computer Society (2012)
17. Famelis, M., Salay, R., Chechik, M.: The semantics of partial model transformations. In: International Workshop on Modeling in Software Engineering, pp. 64–69. IEEE Computer Society (2012)
18. Famelis, M., Salay, R., Sandro, A., Chechik, M.: Transformation of models containing uncertainty. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) MODELS 2013. LNCS, vol. 8107, pp. 673–689. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41533-3_41
19. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: International Conference on Automated Software Engineering, ASE, pp. 3–12. IEEE Computer Society (2002)
20. Huth, M.: Model checking modal transition systems using Kripke structures. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 302–316. Springer, Heidelberg (2002). doi:10.1007/3-540-47813-2_21
21. Huth, M., Jagadeesan, R., Schmidt, D.: Modal transition systems: a foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001). doi:10.1007/3-540-45309-1_11
22. Jones, C.B.: Tentative steps toward a development method for interfering programs. Trans. Program. Lang. Syst. (TOPLAS) **5**(4), 596–619 (1983)
23. Larsen, K.G., Thomsen, B.: A modal process logic. In: Third Annual Symposium on Logic in Computer Science, LICS, pp. 203–210. IEEE Computer Society (1988)
24. Menghi, C.: Dealing with incompleteness in automata based model checking. Ph.D. thesis, Politecnico di Milano (2015). https://www.politesi.polimi.ithandle/10589/114509
25. Menghi, C., Spoletini, P., Ghezzi, C.: Modeling, refining and analyzing Incomplete Büchi Automata. ArXiv e-prints (2016). http://arxiv.org/abs/1609.00610
26. Miremadi, S., Lennartson, B., Åkesson, K.: BDD-based supervisory control on extended finite automata. In: Conference on Automation Science and Engineering, CASE, pp. 25–31. IEEE (2011)

27. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI Series, vol. 13, pp. 123–144. Springer, Heidelberg (1985)
28. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Symposium on Principles of Programming Languages, pp. 179–190. ACM Press (1989)
29. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
30. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**(1), 81–98 (1989)
31. Sery, O., Fedyukovich, G., Sharygina, N.: FunFrog: bounded model checking with interpolation-based function summarization. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 203–207. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33386-6_17
32. Sharygina, N., Chaki, S., Clarke, E., Sinha, N.: Dynamic component substitutability analysis. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 512–528. Springer, Heidelberg (2005). doi:10.1007/11526841_34
33. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005). doi:10.1007/11591191_28
34. Tabakov, D., Vardi, M.Y.: Model checking Büchi specifications. In: International Conference on Language and Automata Theory and Applications, LATA, pp. 565–576. Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona (2007)
35. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: model checking of modal transition systems with variability constraints. J. Log. Algebr. Math. Program. **85**(2), 287–315 (2016)
36. Tsay, Y.-K., Tsai, M.-H., Chang, J.-S., Chang, Y.-W.: Büchi store: an open repository of Büchi automata. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 262–266. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19835-9_23
37. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of partial behavior models from properties and scenarios. IEEE Trans. Softw. Eng. **35**(3), 384–406 (2009)
38. Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: a new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006). doi:10.1007/11817963_5