A Counterexample-Based Incremental and Modular Verification Approach

Étienne André, Kais Klai, Hanen Ochi, and Laure Petrucci

LIPN, CNRS UMR 7030
Université Paris 13, Sorbonne Paris Cité
99 avenue Jean-Baptiste Clément
93430 Villetaneuse, France
{first.last}@lipn.univ-paris13.fr

Abstract. Model checking is a powerful and widespread technique for the verification of finite state concurrent systems. However, the main hindrance for wider application of this technique is the well-known state explosion problem. In [16], we proposed an incremental and compositional verification approach where the system model is partitioned according to the actions occurring in the property to be verified and where the environment of a component is taken into account. But the verification at each increment might be costly. On the other hand, Symbolic Observation Graphs provide a compact analysis means for LTL\X properties. We have shown a purely modular construction of these in [15]. Therefore, in this paper, we combine both techniques to benefit from their pros. Also, we propose a novel approach for incrementally checking the validity of the counter-example.

1 Introduction

Model checking is a powerful and widespread technique for the verification of finite state concurrent systems. However, the main hindrance for wider application of this technique is the well-known state explosion problem. Modular and compositional approaches to verification are promising to tackle this problem. They are based on the "divide and conquer" principle and aim at deducing the properties of the system from those of its components analysed in isolation.

In [16], we proposed an incremental and compositional verification approach where the system model is partitioned according to the actions occurring in the property to be verified and where the environment of a component is taken into account using the linear place invariants of the system. The first component contains only the actions occurring in the formula, and each newly added component is obtained based on the neighbourhood of those already analysed.

However, the verification at each increment might be costly. On the other hand, Symbolic Observation Graphs (SOGs) [9,17] provide an abstraction-based approach leading to a compact representation of the system's state space graph, and allowing for the analysis of properties expressed using LTL\X (Linear Time

Logic [20] from which the "next operator" has been removed). We have shown a purely modular construction of these in [15].

Therefore, in this paper, we combine both techniques to benefit from their advantages. Since, it has been empirically shown that breaking up a system is a difficult task (see for instance [5]), we assume here that the system is already given as a set of components sharing global actions. In order to use an approach derived from [16], either all actions of the formula belong to a single component, or we compose all those containing such actions, to start with. Note that [16] considered Petri nets models, whereas the technique is here generalised to Labelled Transitions Systems (LTSs). In general, the LTS and a counterexample can be derived on-the-fly as long as an initial state and a transition relation are provided.

Related Work. During the last 20 years, many researchers have worked on the use of abstraction and/or modularity to tackle the explosion problem of model-checking properties on concurrent systems. On one hand, modularity refers to a wide range of techniques that make use of the fact that components have some intrinsic behavior of their own. Each component (subpart) of the global system is verified separately and the behavior of the main system is deduced from the behaviors of its components (see, e.g. [22,4]). Among modular techniques, authors of [18] present algorithms to exploit the modular analysis in the determination of reachable states with specified partial markings, to determine possible deadlocks, both global and local, and also liveness. The idea there was to start from a system designed in a modular way and construct the state space of the complete system in a similar way: one local state space per module and a synchronization graph showing their interactions. The technique was applied to a problem of controller design, where some of the actions could be controlled and others not. The approach advocated was also to lift these actions to the global (i.e. synchronization) level, so that both synchronized and controllable actions are visible in the synchronization graph and only there. Another related paradigm is compositional state space verification [26]. In this paradigm, systems are specified as a parallel composition of subcomponents, and the state space of the full system is computed from the state spaces of the subcomponents. Moreover, the state spaces of subcomponents can be replaced by smaller and behaviourally equivalent state spaces before constructing the state space of the full system. Authors use methods and models considering actions in the context of synchonous communicating systems.

On the other hand, abstraction-based techniques aim to build an abstract model of the system by getting rid of some of its irrelevant parts so that the analysis can be achieved on the abstract model instead of the original system. Depending on the property to be checked, the abstract model can either compeletely characterize the system, or represents a super set of its possible behaviors. In the first case, the abstraction satisfies the formula if and only if the original system does (e.g. [25,21,9,17,7]). In the second case, only a sufficient condition exists (i.e. if the abstract model is error-free, then so is the original system). Thus, when the abstract model does not satisfy the property, one can not decide about

the verification result on the original system (e.g. [13]). Counterexample-driven abstraction refinement techniques (e.g. [1,3,6,24,10]) come with an iterative approach to face this weakness: when the abstract model does not satisfy the property, an abstract counterexample is automatically supplied and we check whether it corresponds to a concrete counterexample in the system. If this is the case, we conclude that the system does not satisfy the property. Otherwise, we start over using a new abstract model. In [10] as in the approach presented in this paper, the abstract model used in one pass is obtained using the one computed in the previous pass, while in [3,24,1,6] the abstract model is constructed from scratch and the new one is model-checked.

The approach we present in this paper has the advantage to combine modularity and counter-example abstraction refinement for the verification of temporal properties (generic properties, e.g. deadlock freeness, can also be considered in a similar way).

Benefits and originality of the approach. The approach presented in this paper enjoys several advantages. Firstly, SOGs are computed locally. This favours reuse of modules since once the SOG is computed, it can be used in another environment without need of calculating it again. Moreover, for confidentiality issues, a SOG showing only global actions can be provided instead of the module itself, thus hiding the details of the internal functioning to external users, and favouring the use of "black box" (or "gray box") modules. The verification process is incremental at all stages: not only the formula verification but also checking the counterexample. Thus, the whole LTS does not always require a complete analysis, and the satisfaction of the property can be decided on-the-fly.

Even though the combination of both techniques from [15] and [16] is quite easy, it also leads to improvements. The definition of aggregates contains a more elaborate structure for detecting internal deadlocks, making things way easier at the composition stage. Moreover, the validation of counterexamples is also incremental, sticking to the spirit of the overall approach.

Outline. The paper is structured as follows: after preliminary definitions and notations in Section 2, the approach is introduced in Section 3. It defines the different steps as well as the associated model checking algorithm. Section 4 presents the application of our approach to case studies. Finally, Section 5 concludes and gives perspectives for future work.

2 Preliminaries

The technique presented in this paper applies to different kinds of process models that can map to labelled transition systems, e.g. Petri nets. The techniques addressed here are of particular interest for the analysis of workflow Petri nets (WF-nets) as shown in [14]. For the sake of simplicity and generality, we chose to present it for labelled transition systems, since this formalism is well adapted to event-based approaches.

2.1 Labelled Transition Systems

Definition 1 (Labelled Transition Systems).
A labelled transition system *(LTS for short) is a* 4*-tuple* $\langle \Gamma, Act, \rightarrow, I \rangle$ *where:*

- Γ *is a finite set of* states*;*
- Act *is a finite set of* actions*;*
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *is a* transition relation*;*
- $I \subseteq \Gamma$ *is a set of* initial states*.*

In this paper, we restrict the set of states Γ to those that are reachable from an initial state in I. We distinguish observed actions, denoted by a set Obs, from unobserved actions, denoted by $UnObs$ (with $Obs \cup UnObs = Act$ and $Obs \cap UnObs = \emptyset$). Observed actions include the set of actions occurring in an LTL formula to be verified and interface actions allowing for the synchronisation of two LTSs. Unobserved actions are the remaining ones. Therefore, unobserved actions can be seen as silent τ actions.

In the sequel, we use the following notations:

- For $s, s' \in \Gamma$ and $a \in Act$, we denote by $s \xrightarrow{a} s'$ that $(s, a, s') \in \rightarrow$.
- If $\sigma = a_1 a_2 \cdots a_n$ is a sequence of actions, $\overline{\sigma}$ denotes the set of actions occurring in σ, while $|\sigma|$ denotes the length of σ, and $s \xrightarrow{\sigma} s'$ denotes that $\exists s_1, s_2, \cdots s_{n-1} \in \Gamma$: $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots s_{n-1} \xrightarrow{a_n} s'$.
- The set *Enable(s)* denotes the set of actions a such that $s \xrightarrow{a} s'$ for some state s'. For a set of states S, *Enable(S)* denotes $\bigcup_{s \in S} Enable(s)$.
- $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$ denotes a path of an LTS.
- $s \not\rightarrow$, for $s \in \Gamma$, denotes that s is a dead state, i.e., $Enable(s) = \emptyset$.
- $Reach_{UnObs}(s) = \{s' \mid s \xrightarrow{\sigma} s' \wedge \overline{\sigma} \subseteq UnObs\}$ is the set of states that are reachable from a state s by unobserved actions only. For $S \subseteq \Gamma$, $Reach_{UnObs}(S) = \bigcup_{s \in S} Reach_{UnObs}(s)$.
- $s \not\Rightarrow$, for $s \in \Gamma$, denotes that no state of $Reach_{UnObs}(s)$ enables an observed action, i.e., $Enable(Reach_{UnObs}(s)) \cap Obs = \emptyset$. Conversely, $s \Rightarrow$ denotes $\neg(s \not\Rightarrow)$, i.e. there is a state in $Reach_{UnObs}(s)$ enabling an observed action.
- A finite path $C = s_1 \xrightarrow{\sigma} s_n$ is said to be a *circuit* if $s_n = s_1$ and $|\sigma| \geq 1$. If $\overline{\sigma} \subseteq UnObs$ then C is said to be a *livelock*. If, in addition, $s_1 \not\Rightarrow$ then C is called a *strong livelock*. Otherwise it is called a *weak livelock*.

If $s \not\Rightarrow$ for $s \in \Gamma$, only a dead state or a *strong livelock* are reachable from s. In this paper we assume that a strong livelock behaviour is equivalent to a deadlock. These two behaviours are not distinguished and both are called deadlock. The reason for this is that if unobserved actions are local to a module, the system will somehow be stuck in this module, whatever the others' behaviour.

2.2 Model Checking LTL Formulae

Checking LTL formulae on an LTS is reduced to analyse its *maximal paths*. A *maximal path* is either a finite path (leading to a terminal state) or an infinite one.

Since we observe a subset of the LTS's actions, we distinguish the infinite paths where observed actions occur infinitely often from those where from some point, only unobserved actions occur infinitely often (called divergent paths). It is well known that preserving maximal paths suffices to preserve properties expressed using LTL\X. This corresponds to the CFFD semantics [12], which is exactly the weakest equivalence preserving LTL\X. The usual solution in automata theoretic approaches to check LTL formulae on an LTS is to convert each of its finite paths (leading to a terminal state) to an infinite one by adding a loop onto its last state.

Definition 2 (Maximal paths). *Let $\mathcal{T}$ be an LTS and $\pi = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_n$ a path of $\mathcal{T}$. Then, π is said to be* a maximal path *if one of the following two properties holds:*

- $s_n \not\rightarrow$,
- $\pi = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots s_m \xrightarrow{a_m} \cdots \xrightarrow{a_{n-1}} s_n$ *and* $s_m \xrightarrow{a_m} \cdots \xrightarrow{a_{n-1}} s_n$ *is a circuit.*

Observed Behaviour In the following, we define a particular mapping (called *observed behaviour*) applied to states of an LTS $\mathcal{T}$. It will be established that it is the necessary and sufficient local information to be retained so that LTL\X properties can be checked on the composition of two processes.

Definition 3 (Observed behaviour mapping). *Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ be an LTS. The observed behaviour is progressively defined by :*

1. $\lambda_{\mathcal{T}} : \Gamma \rightarrow 2^{Obs}$
   $\lambda_{\mathcal{T}}(s) = Enable(Reach_{UnObs}(s)) \cap Obs$
2. $\lambda_{\mathcal{T}} : 2^{\Gamma} \rightarrow 2^{Obs}$
   $\lambda_{\mathcal{T}}(S) = \{\lambda_{\mathcal{T}}(s) \mid s \in S\}$
3. $\lambda^{min}_{\mathcal{T}}(S) = \{X \in \lambda_{\mathcal{T}}(S) \mid \nexists Y \in \lambda_{\mathcal{T}}(S) : \ Y \subset X\}$.

Informally, the observed behaviour of a state s, $\lambda_{\mathcal{T}}(s)$, represents the set of observed actions which can be executed from s, possibly via a sequence of unobserved actions. The observed behaviour is then extended to sets of states: the observed behaviour $\lambda_{\mathcal{T}}$ of a set of states S is a set of sets of observed actions. This set contains the observed behaviour of the states of S. Finally, $\lambda^{min}_{\mathcal{T}}(S)$ is the minimal set of subsets (w.r.t. the set inclusion relation) of $\lambda_{\mathcal{T}}(S)$.

The following proposition establishes that deadlock-freeness of an LTS can be deduced from computing the observed behaviour associated with its states.

Proposition 1. *Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ be an LTS. $\mathcal{T}$ is deadlock-free if and only if $\forall S \subseteq \Gamma : \emptyset \notin \lambda^{min}_{\mathcal{T}}(S)$.*

Note that it is actually sufficient to check that, for all individual states s, $\lambda_{\mathcal{T}}(s) \neq \emptyset$. In Section 3, we will need to consider sets of states (instead of states), and this is the reason why Proposition 1 is needed in this form.

2.3 Synchronisation of LTSs

In the following, we define the synchronised product of two LTSs. The synchronised product of n LTSs (for $n > 2$) can be built by iterative multiplication.

Definition 4 (LTS synchronised product). *Let* $\mathcal{T}_i = \langle \Gamma_i, Act_i, \rightarrow_i, I_i \rangle, i = 1, 2$ *be two LTSs. The* synchronised product *of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *is the minimal LTS* $\mathcal{T}_1 \times \mathcal{T}_2 = \langle \Gamma, Act, \rightarrow, I \rangle$ *given by:*

1. $\Gamma \subseteq \Gamma_1 \times \Gamma_2$;
2. $Act = Act_1 \cup Act_2$;
3. $\rightarrow$ *is the transition relation, defined by:*
   $\forall (s_1, s_2) \in \Gamma : (s_1, s_2) \xrightarrow{a} (s'_1, s'_2) \Leftrightarrow$
   $\begin{cases} s_1 \xrightarrow{a}_1 s'_1 \wedge s_2 \xrightarrow{a}_2 s'_2 & \text{if } a \in Act_1 \cap Act_2 \\ s_1 \xrightarrow{a}_1 s'_1 \wedge s_2 = s'_2 & \text{if } a \in Act_1 \setminus Act_2 \\ s_1 = s'_1 \wedge s_2 \xrightarrow{a}_2 s'_2 & \text{if } a \in Act_2 \setminus Act_1 \end{cases}$
4. *The set of states* Γ *contains all (and by minimality only) reachable states:*
   $\Gamma = \{(s_1, s_2) \in \Gamma_1 \times \Gamma_2 \mid \exists (i_1, i_2) \in I_1 \times I_2, \exists \sigma \in Act^* : (i_1, i_2) \xrightarrow{\sigma} (s_1, s_2)\}$;
5. $I = I_1 \times I_2$;

Note that the parallel operator for synchronisation is similar to Hoare's classical alphabetised parallel operator for CSP [11], with the exception that τ actions are synchronised in our settings.

Every state of the synchronised product is a pair of states; the first component indicates the corresponding state of the first LTS; the second component indicates the one of the second LTS. Each LTS can still perform its own activities autonomously, i.e. only one component of the pair representing a state of the composed LTS is changed by such an action. For common activities, both components of the state are changed synchronously.

Consider the two examples of LTSs in Figure 1 (unobserved actions are denoted by τ). The synchronised product of these two LTSs is an LTS containing 24 reachable states, depicted in Figure 2.

Recall that even if two LTSs are deadlock free, their synchronised product is not necessarily. Both LTSs in Figure 1 are deadlock free; however, in the synchronised product in Figure 2, the path $(s_0, s'_0) \xrightarrow{\tau} (s_0, s'_2) \xrightarrow{\tau} (s_1, s'_2)$ leads to the deadlock state (s_1, s'_2).

Notations. Given n LTSs $\mathcal{T}_i$, for $i = 1 \ldots n$, we denote by $\mathcal{T}_{\langle i,\ldots,k \rangle}$, for $1 \leq i < k \leq n$, the LTS representing the synchronised product of the LTSs $\mathcal{T}_i$, $\mathcal{T}_{i+1}$, $\ldots$, $\mathcal{T}_k$. When $i = k$, $\mathcal{T}_{\langle i,\ldots,k \rangle}$ is denoted by $\mathcal{T}_{\langle i \rangle}$.

3 Approach

In this section we describe our incremental and modular approach for model checking LTL\X properties. In order to counter the state space explosion problem we propose to abstract each LTS involved in the whole system by a SOG.

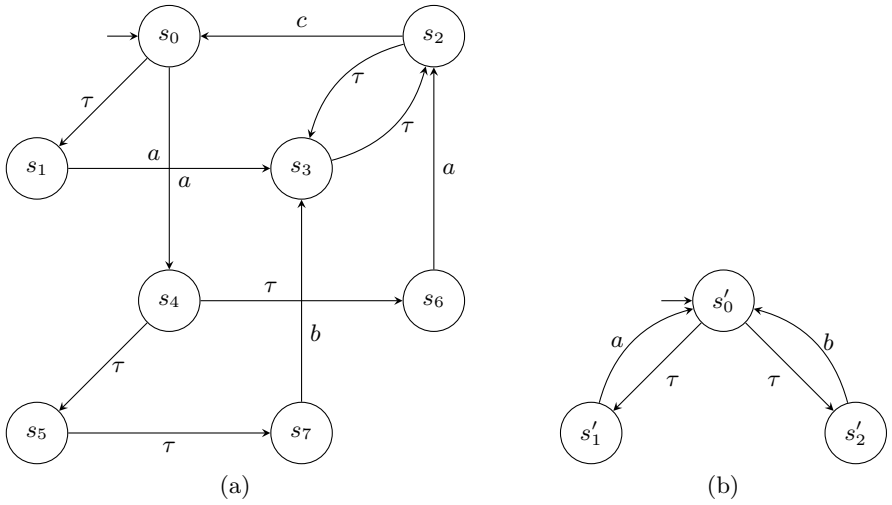


Fig. 1. Two LTSs

This allows for not considering local states, i.e. states (s_1, s_2) that permit the execution of neither interface actions nor actions occurring in the formula to be checked. We recall the notion of Symbolic Observation Graph in Section 3.1, and the preservation of LTL\X properties in Section 3.2. Then we present our approach on top of these notions in Section 3.3.

3.1 The Symbolic Observation Graph

The construction of the SOG corresponding to an LTS is guided by the set of actions occurring in an LTL\X formula expressing a property to be checked. Such actions are said to be observed while the other actions of the system are unobserved. Previous results [9,17] show that such a formula is satisfied by the LTS if and only if it is satisfied by the respective SOG. The SOG is defined as a graph where each node is a set of states linked by unobserved actions and each arc is labelled by an observed action. Nodes of the SOG are called *aggregates* and may be represented and managed efficiently using decision diagram techniques (BDDs, see e.g. [2]). In practice, due to the small number of actions in a typical formula, the SOG has a very moderate size and thus the time complexity of the verification process is negligible in comparison to the building time of the SOG (see [9,17,15] for experimental results). SOGs are used to abstract LTSs so that all internal behaviour is hidden. Additional information is attached to the aggregates so that the preservation of LTL\X formulae still holds by composition. The observed actions are of two kinds: the actions occurring in the LTL formula to be checked, and the interface actions.

Definition 5 (Aggregate). *Let $\mathcal{T} = \langle \Gamma, Act, \rightarrow, I \rangle$ be an LTS with $Act = Obs \cup UnObs$. An* aggregate *is a tuple $a = \langle S, \lambda, l \rangle$ defined as follows:*

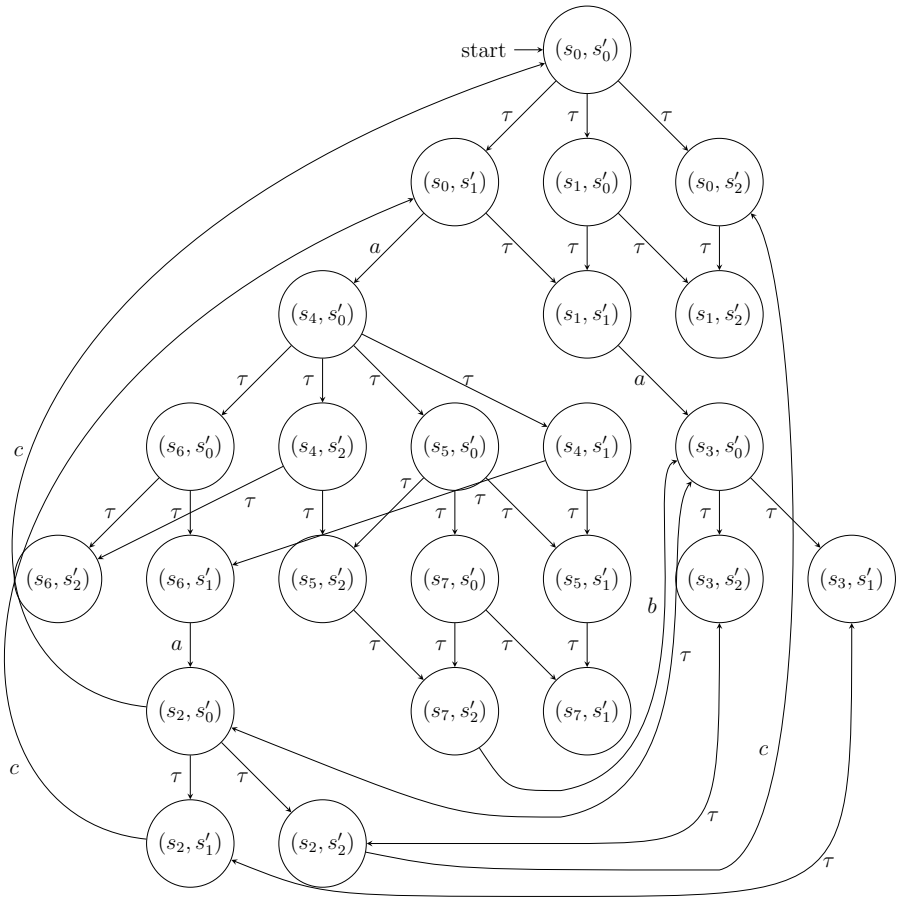


Fig. 2. Product of the 2 LTSs in Figure 1

1. *S is a nonempty subset of Γ satisfying: $Reach_{UnObs}(S) = S$;*
2. *$\lambda = \lambda_{\mathcal{T}}^{min}(S)$*
3. *$l \in \{true, false\}$; $l = true$ iff S contains a weak livelock.*

From now on, $a.S$, $a.\lambda$ and $a.l$ denote the corresponding attributes of a given aggregate a.

In the following definition, we inductively define a SOG associated with an LTS.

Definition 6 (Symbolic Observation Graph). *A* symbolic observation graph *(SOG for short) associated with an LTS $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ is a 4-tuple $\langle \mathcal{A}, Act', \rightarrow', I' \rangle$ where:*

1. *$\mathcal{A}$ is a finite set of aggregates satisfying:*
   - *there is an aggregate $a_0 \in \mathcal{A}$ with $a_0.S = Reach_{UnObs}(I)$, and*

- *if, for some* $a \in \mathcal{A}$ *and* $o \in Obs$*, the set* $Ext(a,o) := \{s' \notin a.S \mid \exists s \in a.S, s \xrightarrow{o} s'\}$ *is not empty, then it is a pairwise-disjoint union of non-empty sets* $S_1 \ldots S_k$*, and for* $i = 1 \ldots k$*, there is an aggregate* $a_i \in \mathcal{A}$ *with* $a_i.S = Reach_{UnObs}(S_i)$*;*

2. $Act' = Obs$*;*
3. $\rightarrow' \subseteq \mathcal{A} \times Act' \times \mathcal{A}$ *is the transition relation satisfying:*
   - *if* $a \neq a'$ *then* $(a,o,a') \in \rightarrow'$ *iff* $a'.S = Reach_{UnObs}(S')$ *for some* $S' \subseteq Ext(a,o)$*, and*
   - $(a,o,a) \in \rightarrow'$ *iff* $Reach_{UnObs}(\{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{o} s'\}) = a.S$*;*
4. $I' = \{a_0\}$ *(where* $a_0.S = Reach_{UnObs}(I)$*).*

Note that Definition 6 does not guarantee the uniqueness of a SOG for a given LTS. In fact, it offers some flexibility for its implementation. In particular, the SOG can be nondeterministic even if the original LTS is not. It is clear that the canonical minimal SOG is obtained when the SOG is deterministic. Actually, one can take advantage of such nondeterminism to obtain smaller aggregates. Even if the SOG obtained in this way has more aggregates than a deterministic one, its construction might consume less time and memory.

This is different from, e.g. determinisation of a process or specification with unobserved actions hidden used in some model checkers.

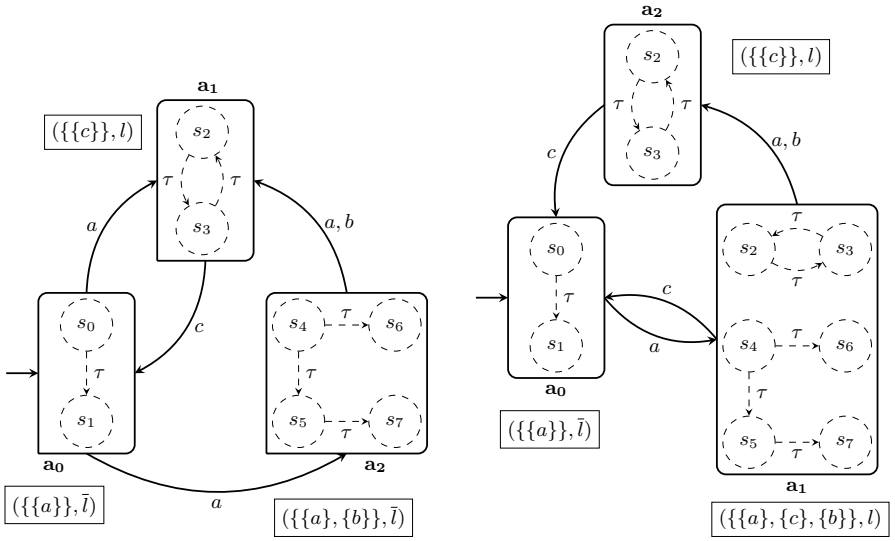


Fig. 3. Two possible SOGs for the LTSs in Figure 1(a)

The two SOGs (a) and (b) of Figure 3 correspond to two possible SOGs associated with the LTS of Figure 1(a) page 7, while the SOG of Figure 4 is a

SOG of the LTS of Figure 1(b). Let us explain the first two SOGs. The set of observed actions is $\{a, b, c\}$ and the unobserved actions are represented by the mute action τ. Each aggregate a is indexed with a pair $(a.\lambda, a.l)$. The left part λ is the observed behaviour associated with a, and indicates whether a contains a deadlock state (viz. $\emptyset \in a.\lambda$). The symbol l (resp. $\bar{l}$) is used when a contains (resp. does not contain) a livelock. The first SOG (Figure 3(a)) is nondeterministic and the sets of states of the aggregates represent a partition of the LTS's states. In this SOG, one can regroup a_1 and a_2 leading to the deterministic SOG (Figure 3(b)) where s_2 and s_3 belong to two different aggregates.

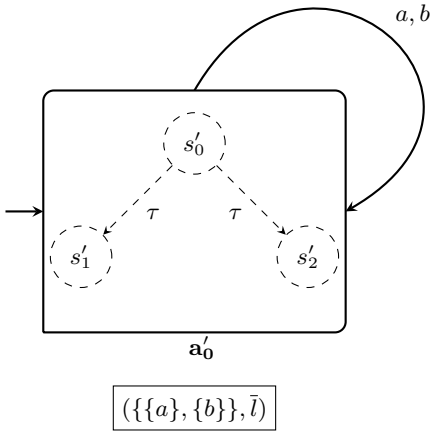


Fig. 4. SOG for the LTSs in Figure 1(b)

Preservation of LTL\X Properties. The equivalence between checking a given LTL\X property on the observation graph and checking it on the original LTS is ensured by the preservation of maximal paths. Thus, the symbolic observation graph preserves the validity of formulae written in classical LTL from which the "next operator" has been removed (because of the abstraction of the immediate successors) (see for instance [23,8]).

The following theorem establishes that checking an LTL\X formula on an LTS can be reduced to check it on a corresponding SOG. It is easily proven by combining our definition of a SOG and results of [9].

Theorem 1. *Let $\mathcal{G}$ be a SOG over a set of observed actions Obs, corresponding to an LTS $\mathcal{T}$. Let φ be a formula from LTL\X on a subset of Obs.*

Then $\mathcal{T} \models \varphi$ iff $\mathcal{G} \models \varphi$.

3.2 Composition of SOGs

Let us consider several LTSs which communicate synchronously. This section shows how to compose the SOGs of the individual LTSs so that the result is isomorphic to some SOG of the composition of the original LTSs. Thus, the composition of SOGs is correct (with respect to LTL\X formulae) if and only if the composition of the original LTSs is correct. However, it is well known that deadlock-freeness is not preserved by composition (e.g. the two LTSs of Figure 1 are deadlock-free but their synchronised product in Figure 2 is not).

The computation of the observed behaviour associated with an aggregate a can be done using symbolic operations exclusively (BDD operations). Moreover, it is not necessary to explore all the states of the aggregate but only analyse the observed transitions and the states that enable these states (immediately).

From now on, an aggregate a is identified by two attributes $a.l$ and $a.\lambda$. Also, the set of states $a.S$ of an aggregate a does not have to be stored explicitly within the aggregate. Once the SOG is built, it will not play any role in the composition process.

When composing several modules, a SOG corresponding to each module is computed locally and once and the obtained SOGs are then composed, leading to a new SOG. The observed behaviour and the livelock attributes of each aggregate of this SOG are deduced from those of the composed aggregates, as follows.

Definition 7 (Product aggregate). *Let $\mathcal{T}_i = \langle \Gamma_i, Obs_i \cup UnObs_i, \rightarrow_i, I_i \rangle$, for $i = 1, 2$, be two LTSs. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Let a_i be an aggregate of $\mathcal{G}_i$. The* product aggregate *$a = a_1 \times a_2$ is defined by:*

1. $a.l = a_1.l \vee a_2.l$
2. $a.\lambda = \{(x \cap y) \cup (x \cap (Obs_1 \setminus Obs_2)) \cup (y \cap (Obs_2 \setminus Obs_1)) \mid x \in a_1.\lambda,\ y \in a_2.\lambda\}$

Deducing the weak livelock attribute of the product aggregate from the involved aggregates is rather trivial: there exists a livelock in the product aggregate $a = a_1 \times a_2$ if and only if there exists a livelock in a_1 or there exists a livelock in a_2. Computing the observed behaviour $a.\lambda$ requires some explanation. First note that the sets of observed actions Obs_1 and Obs_2 are not necessarily identical. When we compose a_1 and a_2, if a_1 can progress in $\mathcal{G}_1$ by using local observed actions (i.e. actions that are observed in $\mathcal{G}_1$ but not shared by $\mathcal{G}_2$), the product aggregate a should be able to do the same. If this is not the case, then a has to have the same behaviour as a_1 and a_2 conjunctively. In this way, the observed behaviour associated with a product aggregate is helpful to deduce whether the involved set of (pairs of) states contains a deadlock.

Proposition 2. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs. Let $\mathcal{T} = \langle \Gamma, Obs \cup UnObs, \rightarrow, I \rangle$ be their synchronised product. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Let a_1 and a_2 be two aggregates of $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively, such that $a = a_1 \times a_2$. Then $\exists s \in (a_1.S \times a_2.S) \cap \Gamma : s \not\Rightarrow$ if and only if $\emptyset \in a.\lambda$.*

Given two SOGs $\mathcal{G}_1$ and $\mathcal{G}_2$, their synchronised product is a SOG $\mathcal{G}$. The synchronised product of two SOGs can be defined similarly to the synchronised product of two LTSs (Definition 4). The only difference is that we deal with aggregates (carrying additional information) instead of states. Definition 7 allows for deducing the attributes of a product attribute $a = a_1 \times a_2$ from the attributes of a_1 and a_2. In particular, the observed behaviour computation allows to detect new deadlock situations, i.e. deadlocks due to the composition process.

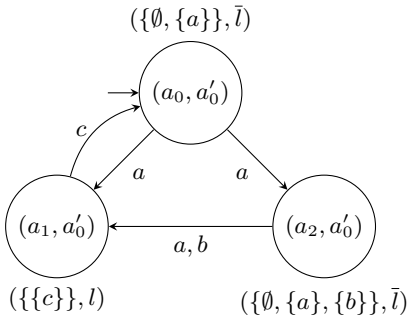


Fig. 5. Product of the SOGs in Figures 3(a) and 4

For instance, the synchronised product between the SOGs of Figures 3(a) and 4 is a SOG (presented in Figure 5) containing three aggregates (a_0, a'_0), (a_2, a'_0) and (a_1, a'_0) where the first two contain a deadlock. Indeed, by composing their observed behaviour we obtain the empty set as a member of the observed behaviour of the product aggregate.

The following theorem will be a basis for our approach. We give an informal illustration of this theorem in Figure 6.

Theorem 2. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs with synchronised product $\mathcal{T}$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$ with respect to observed actions Obs_1 and Obs_2 respectively. Let $\mathcal{G}$ be the synchronised product of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then, $\mathcal{G}$ is a SOG of $\mathcal{T}$ with respect to the observed actions $Obs_1 \cup Obs_2$.*

Corollary 1. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two LTSs with synchronised product $\mathcal{T}$. Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be SOGs corresponding to $\mathcal{T}_1$ and $\mathcal{T}_2$ with respect to observed actions Obs_1 and Obs_2 respectively. Let $\mathcal{G}$ be the synchronised product of $\mathcal{G}_1$ and $\mathcal{G}_2$. Then $\mathcal{T} \models \varphi$ iff $\mathcal{G} \models \varphi$.*

3.3 Verification Algorithm

We suppose that a decomposition of the system $\mathcal{T}$ into n LTSs $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$, and an LTL\X formula φ are given. We also suppose that this decomposition is such

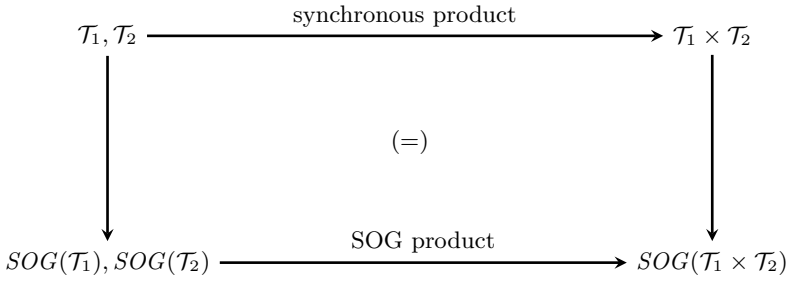


Fig. 6. Illustration of Theorem 2

that all actions appearing within φ appear only in $\mathcal{T}_1$. If this is not the case, we compose all components containing such actions, so that such actions appear only in $\mathcal{T}_1$.

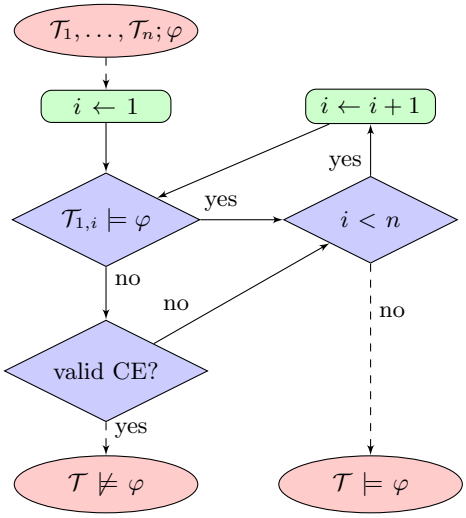


Fig. 7. Our approach (general scheme)

We give the general scheme of our approach in Figure 7. The main principle is that we will check φ on the synchronisation of the SOGs corresponding to an iteratively growing number of LTSs. Starting with $i = 1$, we first check whether $\mathcal{T}_1 \models \varphi$, viz. whether the first subsystem satisfies φ (test "$\mathcal{T}_{\langle 1,\ldots,i\rangle} \models \varphi$" in Figure 7, with $i = 1$). If not, we then check the validity of the counterexample exhibited (test "valid ce" which will be explained below); if the counterexample

is indeed valid, the global system $\mathcal{T}$ does not satisfy the property ("$\mathcal{T} \not\models \varphi$"). If the counterexample is not valid, or if the first subsystem satisfies φ, we go one step further ("$i \leftarrow i+1$") by considering the system obtained by composition of the first and the second subsystems. Note that the satisfiability test (test "$\mathcal{T}_{\langle 1,\ldots,i\rangle} \models \varphi$" with $i = 2$) is performed on the synchronised SOGs, and not on the LTSs, which is much more efficient (see [16]). Also recall that this is equivalent, by Corollary 1. This scheme is performed again iteratively until all subsystems have already been considered; in that case, if the composition of the n SOGs corresponding to the n subsystems satisfies the formula, then the whole system $\mathcal{T}$ also satisfies the formula ("$\mathcal{T} \models \varphi$").

Checking Validity of Counterexamples. Suppose that $\mathcal{T}_{\langle 1,\ldots,i\rangle}$ does not satisfy φ and a counterexample σ has been found. Checking that σ is an actual counterexample (test "valid ce") is performed by analysing the environment part of the system, i.e. $\mathcal{T}_{\langle i+1,\ldots,n\rangle}$. This can be achieved in an incremental way as well, as depicted in Figure 8. Let σ_k be the projection of σ on the actions shared by $\mathcal{T}_{\langle 1,\ldots,i\rangle}$ and $\mathcal{T}_{\langle i+1\rangle}$. If σ_k is not an accepted run of $\mathcal{T}_{\langle i+1\rangle}$, then the counterexample is not valid. Otherwise, we iteratively check the validity of σ on the LTS $\mathcal{T}_{\langle i+1,\ldots,k\rangle}$, for $k = (i+2)\ldots n$. If all iterations show that the projection of σ (on the appropriate sets of actions) is an accepted run, then σ is a valid counterexample.

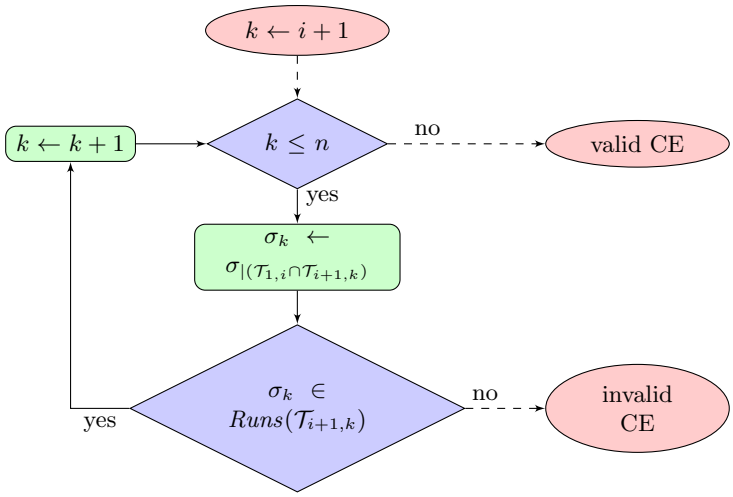


Fig. 8. Approach for checking validity of counterexamples

Advantages. The main interest of our scheme relies on the iterative composition of SOGs instead of LTSs (by Corollary 1). Furthermore, such SOGs are computed

locally: one SOG corresponds to one LTS, independently of any information concerning neighbouring systems except their shared actions. As a consequence, one can reuse SOGs; even better, one can provide a SOG instead of an LTS, and thus allow for confidentiality (the original system is not provided, only its abstraction with respect to its neighbouring systems is). Similarly, refinement of one subsystem is possible without reverifying the whole system, as long as the SOG of the refined subsystem is the same as the original one.

Also note that our scheme is more general than the one of [16] in the sense that we do not give any assumption on the decomposition: we suppose for the sake of simplicity that it is given *a priori*.

4 Case study: the Clients/Servers Example

4.1 Description of the Model

Our approach based on an incremental and modular verification is illustrated on the well known Clients/Servers problem. This is a distributed application which partitions tasks between the providers of a service (called servers), and the requesters of this service (called clients). Clients and servers communicate by sending and receiving messages. This system can be modelled by a composition of clients and servers LTSs depicted in Figure 9. Each client can issue service requests by sending messages to any of the servers, and each server can provide the service to the requesting clients, sending it an answer message.

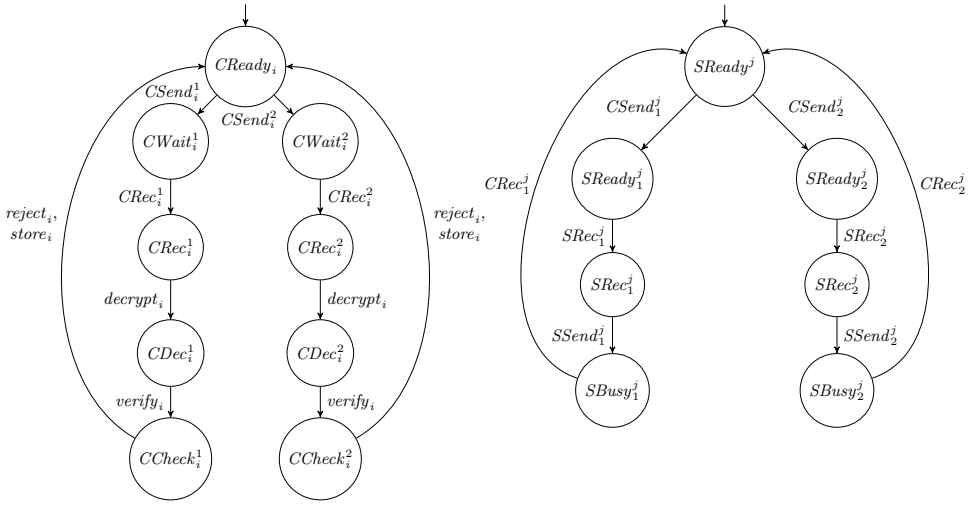
When a client i is ready (state $CReady_i$), it sends a message (action $CSend_i^j$) to a server j which is also in a ready state, $SReady^j$ (for sake of clarity, we use subscript for clients and superscript for servers). The client is then in a pending state ($CWait_i$) waiting for a response from the server to move to the ready state again by enabling $CRec_i^j$. Until then, the server is in a busy state ($SBusy_i^j$) to proceed the received message, and then sends a response message to the corresponding client before returning to its initial state.

Each client has an internal behaviour: After receiving message, the client decrypts and verifies it according to its own rules (actions $decrypt_i$ and $verify_i$). If the message is valid, the client stores it in a local database (action $store_i$); otherwise, the client rejects it (action $reject_i$).

4.2 First Property

We are interested in checking whether the first client receives a response from each server to whom it sends a message. This can be expressed by the LTL formula $\varphi_1 = \Box(CSend_1^j \Rightarrow \Diamond CRec_1^j)$, where $\Box$ reads "always" and $\Diamond$ reads "eventually".

We consider the case where the first client sends a message to server 1, and receives a response from server 1; other cases can be obtained similarly. As mentioned in the previous section, we propose to compose all components such that all actions of the formula appear only in the first LTS (first client $\mathcal{T}_{C1}$): the



(a) LTS $\mathcal{T}_{Ci}$ for client i (b) LTS $\mathcal{T}_{Sj}$ for server j

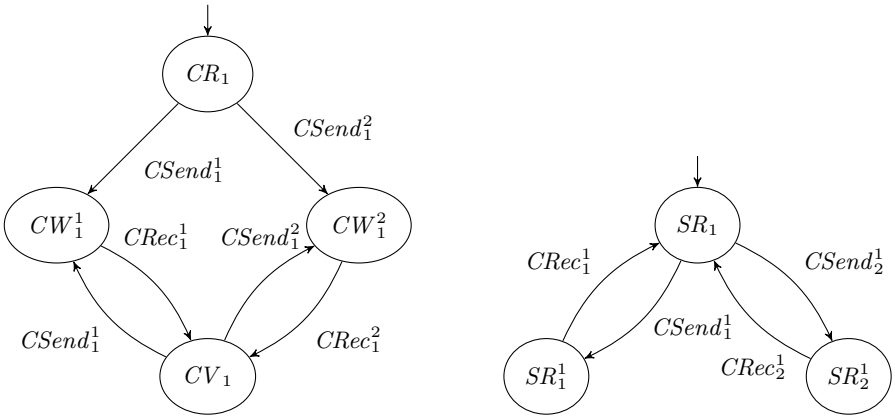
Fig. 9. LTSs composing the Clients/Servers model

second client and the servers are denoted respectively $\mathcal{T}_{C2}$, $\mathcal{T}_{S1}$ and $\mathcal{T}_{S2}$ so that $\mathcal{T} = (\mathcal{T}_{C1}, \mathcal{T}_{S1}, \mathcal{T}_{C2}, \mathcal{T}_{S2}) = \mathcal{T}_{<1,2,3,4>}$. This case can easily be generalized to an arbitrary number of servers and clients.

Step 1 In order to check the LTL formula φ_1 using our approach, we start with the first subsystem $\mathcal{T}_{C1}$. As we can see in Figure 9(a), we obviously have $\mathcal{T}_{C1} \models \varphi_1$ since once a message is sent to the first server (action $CSend_i^1$), this client eventually receives an answer from that server (action $CRec_i^1$). Let us verify this on the corresponding SOG, that we give in Figure 10(a). Observe that only the actions of φ_1 (viz., $CSend_1^1$, $CRec_1^1$, $CSend_1^2$ and $CRec_1^2$) are observable. It is straightforward to verify that φ_1 holds for this SOG.

Step 2 Following our approach in Figure 7, the second step is to synchronise SOGs associated with $\mathcal{T}_{C1}$ and the next subsystem (the first server component, viz. $\mathcal{T}_{S1}$). We give in Figure 10 the SOG of $\mathcal{T}_{S1}$, where only the actions of φ_1 (viz., $CSend_1^1$ and $CRec_1^1$) and the interface actions (viz. $CSend_2^1$ and $CRec_2^1$) are observable.

The obtained synchronised product of SOGs, denoted by $(\mathcal{T}_{C1}, \mathcal{T}_{S1})$, is represented in Figure 11. Note that, for the sake of clarity, we abbreviated some state names; for example, $CReady_1$ is abbreviated with CR_1, and $SRec_1^2$ is ab-



(a) SOG of Client 1 (b) SOG of Server 1

Fig. 10. SOGs of Client 1 and Server 1 for φ_1

breviated with SV_1^2 (V is used for *ReceiVe*, other letters are straightforward). For this subsystem, the formula holds, viz. $(\mathcal{T}_{C1}, \mathcal{T}_{S1}) \models \varphi_1$.

Step 3 We give the rest of the analysis with less details. The verification process is applied to the synchronised product by composing one more subsystem. We get the synchronised product of SOGs $(\mathcal{T}_{C1}, \mathcal{T}_{S1}, \mathcal{T}_{C2})$, that we do not represent here. It can be shown that the formula φ_1 is satisfied by this product.

Step 4 Then, we go one step further, and we perform the synchronised product of SOGs $(\mathcal{T}_{C1}, \mathcal{T}_{S1}, \mathcal{T}_{C2}, \mathcal{T}_{S2})$ (again, which is not given here). The formula is also satisfied by this product. Hence, the formula is satisfied by the whole system $\mathcal{T}$.

With the earlier approach from [16], the whole system had to be analysed to prove that the property holds. In this case, the contribution of our modular and incremental approach using a counterexample is not visible. Nevertheless, the gain is obtained when checking the property on the synchronised product of subsystems' abstractions (SOGs), so that the graph is smaller and the verification process is faster. Considering the LTSs instead of the SOGs would result in a much larger product. Even for the simple (non-synchronised) LTSs in Figure 9, their corresponding SOG (in Figure 10) is much smaller: compare a maximum of 9×7 states (in the worst case) for the LTSs, with a maximum of 4×3 for the SOG (actually 7, see Figure 11. This example confirms the first advantage of our approach which reduces the complexity of model checking.

4.3 Second Property

Let us now consider another property to verify: the first client has to alternate between the two servers at least once when sending messages. This property can

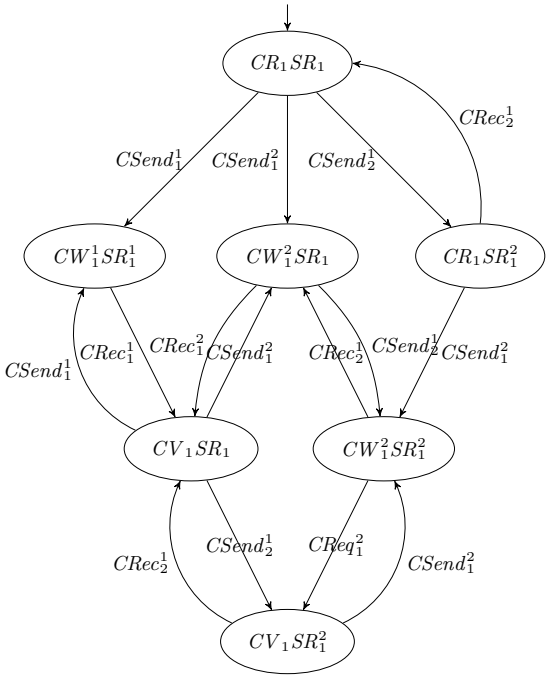


Fig. 11. Synchronised product of SOGs $\mathcal{T}_{<1,2>} = (\mathcal{T}_{C1}, \mathcal{T}_{S1})$

be expressed by the LTL formula $\varphi_2 = \Box(CSend_1^i \Rightarrow \Diamond CSend_1^j)$ with $i \neq j$ and $i, j \in \{1, 2\}$.

Let us consider the first component, viz. the SOG of the first client (given in Figure 10(a)). We immediately notice that φ_2 is not satisfied, and a counterexample σ is deduced which is the infinite path composed by $(CSend_1^1 CRec_1^1)^\infty$. Using our algorithm to check the validity of the counterexample (given in Figure 8), we can deduce that σ is a valid counterexample that can be run on $\mathcal{T}_{<1,2>}$, $\mathcal{T}_{<1,2,3>}$ and $\mathcal{T}_{<1,2,3,4>} = \mathcal{T}$. Therefore we can deduce that $\mathcal{T} \not\models \varphi_2$. Hence, our counterexample-based approach is more efficient for checking a formula which is not satisfied by the system, because we could prove the non-validity of the formula directly from a single component.

5 Conclusion

We proposed here an incremental and compositional verification approach based on [16]. We improved that approach by incrementally verifying the counterexample on incremental partial decompositions of the LTS. Our approach has the following advantages. First, by composing the LTSs using SOGs [15], we strongly reduce the complexity of this verification when compared to monolithic

verification. In the worst case, i.e. if the formula is indeed valid, we verify it on the whole set of LTSs; this remains much more efficient than monolithic verification, due to the use of SOGs. Second, it allows the verification of systems containing black box (or gray box) subsystems: if one does not want to provide some part of the system (e.g. due to confidentiality issues) one may still provide the corresponding abstraction under the form of its SOG, thus allowing verification without disclosing the precise implementation. This also allows for reusing some components under the form of their SOG. Several issues can be investigated in the future: Given a decomposition $\langle \mathcal{T}_1, \ldots, \mathcal{T}_n \rangle$, we considered so far that the actions of the LTL\X formula φ all appear in $\mathcal{T}_1$ only (see Section 3.3). If these actions appear in further LTSs, one idea would be to decompose the formula in subformulae, using for instance [19], each to be checked on the underlying subcompoenent. Also, we suppose that the decomposition of the system is already given. As done in [16], one can build a decomposition of the system which is guided by the formula to be checked.

An efficient implementation of our approach is ongoing. It will both strengthen the initial results on examples of moderate size and allow for comparing the approach developed here with the monolithic verification on the one hand, and with the approach of [16] on the other hand.

References

1. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, SPIN '01, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc. 3
2. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992. 7
3. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169. Springer, 2000. 3
4. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS'89*, pages 353–362, 1989. 2
5. J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):7:1–7:52, May 2008. 2
6. S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 51–, Washington, DC, USA, 2001. IEEE Computer Society. 3
7. A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg. Self-loop aggregation product – a new hybrid approach to on-the-fly LTL model checking. In *Automated Technology for Verification and Analysis: 9th International Conference, ATVA 2011, Taipei, Taiwan, ROC, October 11-14, 2011. Proceedings*, volume 6996 of *Lecture Notes in Computer Science*. Springer, 2011. 2
8. U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In R. Cleaveland, editor, *CONCUR*, pages 222–236, 1992. 10

9. S. Haddad, J.-M. Ilié, and K. Klai. Design and evaluation of a symbolic and abstraction-based model checker. In F. Wang, editor, *ATVA*, volume 3299 of *LNCS*, pages 196–210. Springer, 2004. 1, 2, 7, 10
10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002. 3
11. C. Hoare. Communicating sequential process. *Communication of the ACM*, 21(8):666–677, August 1978. 6
12. R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In R. Cleaveland, editor, *CONCUR*, pages 207–221, 1992. 5
13. K. Klai, S. Haddad, and J.-M. Ilié. Modular verification of petri nets properties: A structure-based approach. In *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 2005. 3
14. K. Klai and H. Ochi. Modular verification of inter-enterprise business processes. In *eKNOW*, pages 155–161, 2012. 3
15. K. Klai and L. Petrucci. Modular construction of the symbolic observation graph. In J. Billington, Z. Duan, and M. Koutny, editors, *ACSD*, pages 88–97. IEEE, 2008. 1, 2, 3, 7, 18
16. K. Klai, L. Petrucci, and M. Reniers. An incremental and modular technique for checking LTL\X properties of Petri nets. In J. Derrick and J. Vain, editors, *FORTE*, volume 4574 of *LNCS*, pages 280–295. Springer-Verlag, 2007. 1, 2, 3, 14, 15, 17, 18, 19
17. K. Klai and D. Poitrenaud. MC-SOG: An LTL model checker based on symbolic observation graphs. In K. M. van Hee and R. Valk, editors, *Petri Nets*, volume 5062 of *LNCS*, pages 288–306. Springer, 2008. 1, 2, 7
18. C. Lakos and L. Petrucci. Modular analysis of systems composed of semiautonomous subsystems. pages 185–194, 2004. 2
19. A. Lehmann, N. Lohmann, and K. Wolf. Stubborn sets for simple linear time properties. In S. Haddad and L. Pomello, editors, *Petri Nets*, LNCS. Springer-Verlag, June 2012. 19
20. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992. 2
21. D. Peled, A. Valmari, and I. Kokkarinen. Relaxed visibility enhances partial order reduction. *Formal Methods in System Design*, 19(3):275–289, 2001. 2
22. A. Pnueli. Logics and models of concurrent systems. chapter In transition from global to modular temporal reasoning about programs, pages 123–144. Springer-Verlag New York, Inc., 1985. 2
23. A. Puhakka and A. Valmari. Weakest-congruence results for livelock-preserving equivalences. In J. C. M. Baeten and S. Mauw, editors, *CONCUR*, pages 510–524, 1999. 10
24. H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 377–396, London, UK, UK, 2000. Springer-Verlag. 3
25. A. Valmari. On-the-fly verification with stubborn sets. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 1993. 2
26. A. Valmari. Compositionality in state space verification methods. In *Application and Theory of Petri Nets*, volume 1091 of *LNCS*, pages 29–56. Springer, 1996. 2