Compositional Model Checking with Incremental Counter-Example Construction

Anton Wijs and Thomas Neele

Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands
{A.J.Wijs, T.S.Neele}@tue.nl

Abstract. In compositional model checking, the approach is to reason about the correctness of a system by lifting results obtained in analyses of subsystems to the system-level. The main challenge, however, is that requirements, in the form of temporal logic formulae, are usually specified at the system-level, and it is not obvious how to relate these to subsystem-local behaviour. In this paper, we propose a new approach to checking regular safety properties, which we call Incremental Counter-Example Construction (ICC). Its main strong point is that it performs a series of model checking procedures, and that each one only explores a small part of the entire state space. This makes ICC an excellent approach in those cases where state space explosion is an issue. Moreover, it is frequently much faster than traditional explicit-state model checking, particularly when the model satisfies the verified property, and in most cases not significantly slower. We explain the technique, and report on experiments we have conducted using an implementation of ICC, comparing the results to those obtained with other approaches.

1 Introduction

Model checking [3] is an automatic technique to verify that a given specification of a concurrent system meets a particular functional property. The specification of a concurrent system describes a finite number of *components*, or processes, and how these can interact. Model checking involves very time and memory demanding computations. Most computations rely on state space exploration. This involves interpreting the specification, resulting in building a graph, or *state space*, describing all its potential behaviour.

However, model checking suffers from the *state space explosion problem*, meaning that a linear growth of the model tends to lead to an exponential growth of the corresponding state space. Over the years, a whole range of techniques have been proposed to mitigate this problem. One prominent technique is *compositional model checking* [10]. The aim is to break down the model checking problem into several subproblems, and solve these individually, thereby achieving a compositional approach.

The main challenge in compositional model checking is that on the one hand, one wishes to reason about the correctness of subsystems or components and lift those results to the system level, but on the other hand, the functional property

to be checked is usually expressed directly at the system level. Furthermore, the possible interactions between the components need to be taken into account when verifying, therefore only checking components in isolation does not suffice.

In this paper, we present a new approach to compositional model checking, which we call *Incremental Counter-Example Construction* (ICC). The main idea is that the system components are placed in a fixed order, and a sequence of verification checks is performed, each involving a single component M in the system in the specified order. Furthermore, each check involves a version of the negation of the functional property φ at the relevant level of abstraction, and a partially built counter-example c. The goal of each check is to extend c with behaviour of M in such a way that (the abstract version of) φ is still violated. If one is able to extend a counter-example with behaviour of all components in the system, then a complete counter-example has been successfully constructed. If extending c fails in some check, ICC backtracks to an earlier check to produce a new counter-example candidate. Rejected candidates are added to checks as constraints to prevent them from being proposed again.

The main benefit of ICC is that it is often very memory-efficient; frequently the individual checks explore state spaces that are orders of magnitude smaller than the full system state space. For models with sufficiently large state spaces, we observe that ICC allows us to check those models, while traditional model checking runs out of memory.

Another benefit is that, while reducing the memory-use, ICC is actually not significantly slower than traditional, explicit-state model checking. In fact, it is frequently even much faster, particularly in those cases where individual checks can quickly discard large parts of the state space.

The structure of the paper is as follows: Section 2 presents the preliminaries. In Section 3, the ICC procedure is presented. Optimisations of this algorithm are discussed in Section 4. Then, experimental results are presented in Section 5. Related work is discussed in Section 6, and finally, Section 7 contains conclusions and pointers to future work.

2 Preliminaries

Concurrent system semantics. We capture the formal semantics of single components in concurrent systems in *Labelled Transition Systems*.

Definition 1 (Labelled Transition System). *An* LTS $\mathcal{G}$ *is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^{in} \rangle$*, with*

- $\mathcal{S}$ *a finite set of states;*
- $\mathcal{A}$ *a set of action labels, not containing the special internal, or hidden, system action* τ*;*
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \cup \{\tau\} \times \mathcal{S}$ *a transition relation;*
- $s^{in} \in \mathcal{S}$ *the initial state.*

An LTS $\mathcal{G}$ with accepting states *has an additional tuple element* $\mathcal{F}_{\mathcal{G}} \subseteq \mathcal{S}$*, which is called the set of* accepting states.

The set $\mathcal{A} \cup \{\tau\}$ is denoted by $\mathcal{A}_\tau$. Action labels in $\mathcal{A}$ are denoted by a, b, c, etc, while actions in $\mathcal{A}_\tau$ are denoted by ℓ. A transition $(s, \ell, s') \in \mathcal{T}$, or $s \xrightarrow{\ell} s'$ for short, denotes that LTS $\mathcal{G}$ can move from state s to state s' by performing the ℓ-action. Whenever we want to make explicit that $s \xrightarrow{\ell} s'$ is a transition of $\mathcal{G}$, we write $s \xrightarrow{\ell}_{\mathcal{G}} s'$. We call $\mathcal{G}$ *deterministic* iff for all $\ell \in \mathcal{A} \cup \{\tau\}$ and $s, s' \in \mathcal{S}$, if $s \xrightarrow{\ell} s'$, then there exists no $s'' \in \mathcal{S}$ with $s' \neq s''$ and also $s \xrightarrow{\ell} s''$. The reflexive, transitive closure of $\xrightarrow{\tau}$ is indicated by $\Longrightarrow$.

A *path* $\sigma = \langle s^{in} \xrightarrow{\ell_1} \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} s_n \rangle$ through $\mathcal{G}$ of length n is a sequence of n transitions, starting from the initial state, that all exist in $\mathcal{T}$. We call a state $s \in \mathcal{S}$ *reachable* iff there exists at least one path from s^{in} to s. The *trace* described by σ is the sequence of actions $w(\sigma) = \langle \ell_1, \ldots, \ell_m \rangle \in \mathcal{A}_\tau^*$ as they appear in σ. The trace $w(\sigma_1)$ of path σ_1 is a *prefix* of $w(\sigma_2)$ of path σ_2 iff $w(\sigma_2)$ can be obtained by extending $w(\sigma_1)$. A trace v is said to be accepted by LTS $\mathcal{G}$ iff there is at least one path σ through $\mathcal{G}$ leading to a state in $\mathcal{F}_\mathcal{G}$ and $w(\sigma) = v$. When relevant, we denote this by $v\checkmark$. We refer to the empty trace with ϵ.

We write $1..n$ for the set of integers ranging from 1 to n. A vector $\bar{v}$ of size n contains n elements indexed from 1 to n. For all $i \in 1..n$, $\bar{v}_i$ represents the i^{th} element of vector $\bar{v}$.

LTSs can be combined using *parallel composition*, for which we use the convention that LTSs must synchronise on common actions, while actions unique to one LTS represent independent actions. An exception to this is the τ-action: internal steps of an LTS are not synchronised with those of another.

Definition 2 (Parallel composition). *Given two LTSs* $\mathcal{G}_1 = \langle \mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, s_1^{in} \rangle$ *and* $\mathcal{G}_2 = \langle \mathcal{S}_2, \mathcal{A}_2, \mathcal{T}_2, s_2^{in} \rangle$*, we say that* $\mathcal{M} = \mathcal{G}_1 \,||\, \mathcal{G}_2$ *is the* parallel composition of $\mathcal{G}_1$ and $\mathcal{G}_2$*. Its LTS* $\mathcal{M} = \langle \mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, \bar{s}_\mathcal{M}^{in} \rangle$ *is defined as follows:*

- $\bar{s}_\mathcal{M}^{in} = \langle s_1^{in}, s_2^{in} \rangle$*;*
- $\mathcal{T}_\mathcal{M}$ *and* $\mathcal{S}_\mathcal{M}$ *are the smallest relation and set, respectively, satisfying* $\bar{s}_\mathcal{M}^{in} \in \mathcal{S}_\mathcal{M}$ *and for all* $\bar{s} \in \mathcal{S}_\mathcal{M}, \ell \in \mathcal{A}_1 \cup \mathcal{A}_2 \cup \{\tau\}$*:*
  - $\bar{s}_1 \xrightarrow{\ell}_1 t \wedge \ell \notin \mathcal{A}_2 \implies \bar{s} \xrightarrow{\ell}_\mathcal{M} \langle t, \bar{s}_2 \rangle \wedge \langle t, \bar{s}_2 \rangle \in \mathcal{S}_\mathcal{M}$*;*
  - $\bar{s}_2 \xrightarrow{\ell}_2 t \wedge \ell \notin \mathcal{A}_1 \implies \bar{s} \xrightarrow{\ell}_\mathcal{M} \langle \bar{s}_1, t \rangle \wedge \langle \bar{s}_1, t \rangle \in \mathcal{S}_\mathcal{M}$*;*
  - $\bar{s}_1 \xrightarrow{\ell}_1 t \wedge \bar{s}_2 \xrightarrow{\ell}_2 t' \wedge \ell \neq \tau \implies \bar{s} \xrightarrow{\ell}_\mathcal{M} \langle t, t' \rangle \wedge \langle t, t' \rangle \in \mathcal{S}_\mathcal{M}$*.*
- $\mathcal{A}_\mathcal{M} = \{a \mid \exists \bar{s}, \bar{s}' \in \mathcal{S}_\mathcal{M} . \bar{s} \xrightarrow{a}_\mathcal{M} \bar{s}'\} \setminus \{\tau\}$*.*

Besides the parallel composition as defined in Def. 2, we also use a special parallel composition operator $|||$, which is identical to $||$ except for the (non-)synchronisation of τ-actions: contrary to $||$, $|||$ also forces synchronisation between LTSs on τ-actions. For that reason, we refer to the latter form of parallel composition as *fully synchronised parallel composition*.

Encoding and Verifying Regular Safety Properties. A safety property φ is a linear time property that describes which infinite traces in $\mathcal{A}^*$ are considered correct. Therefore, its negation $\neg\varphi$ describes which traces violate φ by listing all finite bad prefixes of those traces. If this set of bad prefixes constitutes a regular

language, then φ is said to be regular [3]. The negation $\neg\varphi$ can be encoded in an LTS with accepting states $\mathcal{P}^{\neg\varphi} = \langle \mathcal{S}_\mathcal{P}, \mathcal{A}_\mathcal{P}, \mathcal{T}_\mathcal{P}, s^{in}_\mathcal{P}, \mathcal{F}_\mathcal{P} \rangle$.

Verifying whether a system $\mathcal{M}$, consisting of n components of the form $\mathcal{M}_i = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, s^{in}_i \rangle$ $(i \in 1..n)$ satisfies a regular safety property φ boils down to checking whether in the parallel composition $\mathcal{M}_1 \mid\mid \cdots \mid\mid \mathcal{M}_n \mid\mid \mathcal{P}^{\neg\varphi}$ a system state $\langle s_1, \ldots, s_n, s' \rangle$ is reachable from $\langle s^{in}_1, \ldots, s^{in}_n, s^{in}_\mathcal{P} \rangle$ in which $s' \in \mathcal{F}_\mathcal{P}$. For convenience, we also call such a system state an accepting state. In fact, in this paper, we use a generalised version of this definition of accepting state: in a parallel composition of LTSs $\mathcal{G}_1 \mid\mid \cdots \mid\mid \mathcal{G}_n$, we say that a system state $\langle s_1, \ldots, s_n \rangle$ is accepting iff for all $\mathcal{G}_i$ containing accepting states, i.e., $\mathcal{F}_{\mathcal{G}_i} \neq \emptyset$, we have that $s_i \in \mathcal{F}_{\mathcal{G}_i}$.

Trace equivalences. As equivalence relations between LTSs, we use both trace equivalence and weak trace equivalence [7]. In contrast to trace equivalence, weak trace equivalence is sensitive to internal actions. These equivalences can be used to minimise an LTS, i.e., obtain a reduced LTS in which all the (visible) traces are preserved that are present in the original one. To define these equivalences, we first define for an LTS with accepting states $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^{in}, \mathcal{F} \rangle$ the set of traces and weak traces of a state $s \in \mathcal{S}$. Sets $\mathcal{A} \cup \{\checkmark\}$ and $\mathcal{A}_\tau \cup \{\checkmark\}$ are denoted by $\mathcal{A}_\checkmark$ and $\mathcal{A}_{\tau,\checkmark}$, respectively.

Definition 3 (Traces of a state). *For a state $s \in \mathcal{S}$,* $\mathrm{Traces}(s)$ *is the minimal set satisfying:*

- $\epsilon \in \mathrm{Traces}(s)$*;*
- $\checkmark \in \mathrm{Traces}(s)$ *iff* $s \in \mathcal{F}$*;*
- *For all* $\ell \in \mathcal{A}_\tau, \sigma \in \mathcal{A}^*_{\tau,\checkmark}$*, we have* $\ell\sigma \in \mathrm{Traces}(s)$ *iff there exists an* $s' \in \mathcal{S}$ *such that* $s \xrightarrow{\ell} s'$ *and* $\sigma \in \mathrm{Traces}(s')$*.*

Definition 4 (Weak traces of a state). *For a state $s \in \mathcal{S}$,* $\mathrm{WTraces}(s)$ *is the minimal set satisfying:*

- $\epsilon \in \mathrm{WTraces}(s)$*;*
- $\checkmark \in \mathrm{WTraces}(s)$ *iff* $s \in \mathcal{F}$*;*
- *For all* $a \in \mathcal{A}, \sigma \in \mathcal{A}^*_\checkmark$*, we have* $a\sigma \in \mathrm{WTraces}(s)$ *iff there exists an* $s' \in \mathcal{S}$ *such that* $s \xrightarrow{a} s'$ *and* $\sigma \in \mathrm{WTraces}(s')$*;*
- *For all* $\sigma \in \mathcal{A}^*_\checkmark$*, we have* $\sigma \in \mathrm{WTraces}(s)$ *iff there exists an* $s' \in \mathcal{S}$ *such that* $s \xrightarrow{\tau} s'$ *and* $\sigma \in \mathrm{WTraces}(s')$*.*

Definition 5 (Trace equivalence). *States s, s' are* trace equivalent *iff*

$$\mathrm{Traces}(s) = \mathrm{Traces}(s')$$

Definition 6 (Weak trace equivalence). *States s, s' are* weak trace equivalent *iff*

$$\mathrm{WTraces}(s) = \mathrm{WTraces}(s')$$

We say that two LTSs $\mathcal{G}_1 = \langle \mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, s_1^{in}, \mathcal{F}_1 \rangle$ and $\mathcal{G}_2 = \langle \mathcal{S}_2, \mathcal{A}_2, \mathcal{T}_2, s_2^{in}, \mathcal{F}_2 \rangle$ are trace equivalent and weak trace equivalent iff their initial states s_1^{in} and s_2^{in} are trace equivalent and weak trace equivalent, respectively. Finally, given an LTS $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^{in}, \mathcal{F} \rangle$, we refer with $\mathrm{Traces}(\mathcal{G})$ and $\mathrm{WTraces}(\mathcal{G})$ to $\mathrm{Traces}(s^{in})$ and $\mathrm{WTraces}(s^{in})$, respectively.

It is known that linear-time properties are preserved by trace equivalence [3], i.e., if an LTS $\mathcal{G}_1$ satisfies a linear-time property φ and $\mathcal{G}_1$ is trace equivalent to $\mathcal{G}_2$, then also $\mathcal{G}_2$ satisfies φ. The same holds for weak trace equivalence, as long as φ does not refer to the internal action τ. The standard powerset construction algorithm to determinise finite automata [40] can be used to reduce LTSs w.r.t. trace and weak trace equivalence. Although this algorithm has worst-case complexity $O(2^{|\mathcal{S}|})$, reducing small LTSs of system components can still be done relatively fast. As an intermediate step, one could consider first reducing the LTS w.r.t. branching bisimulation, which can be done in $O(|\mathcal{T}| \cdot (\log |\mathcal{A}| + \log |\mathcal{S}|)$ [26].

Abstraction. To raise the abstraction level of an LTS, we define *action hiding* of an LTS w.r.t. a set of actions A.

Definition 7 (Action hiding). *Given an LTS $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^{in} \rangle$, we define the LTS $\mathcal{G}' = \langle \mathcal{S}, \mathcal{A}', \mathcal{T}', s^{in} \rangle$ resulting from action hiding $\mathcal{G}$ w.r.t. A as follows:*

- $\mathcal{A}' = \mathcal{A} \cap A$;
- $\mathcal{T}' = \{(s, \ell, s') \mid (s, \ell, s') \in \mathcal{T} \wedge \ell \in A\} \cup \{(s, \tau, s') \mid (s, \ell, s') \in \mathcal{T} \wedge \ell \notin A\}$.

With $\mathcal{G}_{\downarrow A}$, we denote the LTS resulting from first action hiding $\mathcal{G}$ w.r.t. A, and subsequently applying weak trace equivalence reduction on the action hidden LTS. Similarly, with $\mathcal{G}_{\downarrow}$ we refer to the LTS obtained by applying trace equivalence reduction on $\mathcal{G}$. Note that $\mathcal{G}_{\downarrow}$ is in general not equivalent to $\mathcal{G}_{\downarrow \mathcal{A}}$, in particular when τ-transitions are present in $\mathcal{G}$.

3 Incremental Counter-Example Construction

In this section, we introduce the basic approach to compositionally verify whether a system $\mathcal{M}$ satisfies a regular safety property φ via ICC.

3.1 The ICC Algorithm

We first illustrate how the algorithm works by using an example.

Example. Consider the two LTSs and the property LTS depicted in Figure 1, where the doubly lined state denotes an accepting state, and states with a detached incoming arrow are initial. For this system, the ICC procedure works as follows: first, we place LTSs $\mathcal{M}_1$ and $\mathcal{M}_2$ in some order, say the order in which they are depicted in Figure 1. Then, we analyse the parallel composition of the first LTS and an abstract version of the property LTS w.r.t. the action set $\mathcal{A}_1$: $\mathcal{M}_1 \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1}$. A Breadth-First Search exploration of the state space will reach

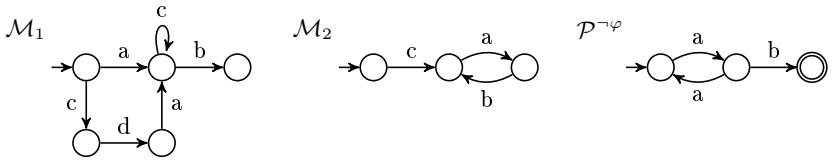

Fig. 1: Example with two LTSs and the property "after an odd amount of a's no b can be performed". The doubly-lined state is an accepting state.

an accepting state via a path describing the traces ac^*b (it should be noted that the correctness of ICC does not depend on the exploration strategy). Next, we replace $\mathcal{M}_1$ by an LTS $\mathcal{L}_1$ with Traces($\mathcal{L}_1$) = $\{ab\checkmark, acb\checkmark, accb\checkmark, \ldots\}$, include $\mathcal{M}_2$ in the analysis, and consider another abstract version of the property LTS. This means that we search for a counter-example in $\mathcal{L}_1 \mid\mid \mathcal{M}_2 \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1 \cup \mathcal{A}_2}$. In this case, note that since $\mathcal{L}_1$ has an accepting state, a system state is accepting iff both $\mathcal{L}_1$ and $\mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1 \cup \mathcal{A}_2}$ are in accepting states. Also note that the property LTS is now abstracted w.r.t. actions in both LTSs. In this system, no accepting state is reachable, therefore we have to go back to $\mathcal{M}_1 \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1}$ to find another accepted trace. Now, we find a path describing the traces $cdac^*b$, which we use to construct a new $\mathcal{L}'_1$. Since an accepting state can be reached in $\mathcal{L}'_1 \mid\mid \mathcal{M}_2 \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1 \cup \mathcal{A}_2}$, we conclude that the property does not hold.

The $\mathcal{L}_i$ and $\mathcal{L}'_i$ referred to in the example above can actually be directly extracted from the state space that was explored up to the point when the first accepting state was reached. Consider having to check the parallel composition of $\mathcal{L}_1 \mid\mid \cdots \mid\mid \mathcal{L}_{i-1} \mid\mid \mathcal{M}_i \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_i}$, with $i \in 1..n$. When the state space is explored, an accepting state is searched on-the-fly, and, once detected, the state space exploration is terminated. Subsequently, *all the traces* of component $\mathcal{M}_i$ that are accepted in the state space explored so far, are taken into account to construct an LTS for the next check. Such an LTS can be constructed as follows from an LTS $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s^{in}, \mathcal{F} \rangle$ representing the state space explored so far, where $\mathcal{F}$ is the singleton set $\{s\}$.

1. Remove all states from $\mathcal{G}$ from which s cannot be reached. This can be efficiently done by exploring $\mathcal{G}$ in the opposite direction, starting from s, and after doing so, removing all the unreached states. Let us call the resulting LTS $\mathcal{G}'$.
2. Action hide $\mathcal{G}'$ w.r.t. $\mathcal{A}_i$, thereby only keeping the behaviour of $\mathcal{M}_i$ visible. In order for this to work correctly, it is required that during construction of $\mathcal{G}$, the explored τ-transitions that originated from $\mathcal{M}_i$ have been labelled in such a way that they can be distinguished from other τ-transitions (for instance with the label τ').
3. Finally, relabel the τ'-transitions in $\mathcal{G}'_{\downarrow \mathcal{A}_i}$ back to τ-transitions.

Alg. 1 presents the basic ICC technique. We iterate over the components of the system (lines 1-2), and in each iteration i, we construct a verification task $Check_i$ (line 3). Note that the order here in which the components are considered

Algorithm 1 Incremental Counter-Example Construction

Require: $\langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle$, $\mathcal{P}^{\neg\varphi}$
Ensure: **true** is returned if $\mathcal{M}$ satisfies φ, otherwise a counter-example is returned

$i \leftarrow 1$
2: **while** $i \leq n$ **do**
  $result \leftarrow Check_i$ (= explore $\mathcal{L}_1 \,||\, \cdots \,||\, \mathcal{L}_{i-1} \,||\, (\mathcal{M}_i \,|||\, \mathcal{R}_{i\downarrow}) \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow \bigcup_{k \in 1..i} \mathcal{A}_k}$)
4: **if** $\neg result$ **then**
    construct LTS $\mathcal{L}_i$ containing all the accepted traces of $\mathcal{M}_i$ in the state space
    // New counter-example found, update
6:    $i \leftarrow i + 1$ // Go to next $Check_i$
  else if $i = 1$ **then**
8:    **return true** // Property is satisfied
  else
10:    identify the smallest $j < i$ for which $\mathcal{L}_j$ caused $Check_i$ to not reach an accepting state
      $updatePreviousRestrictions(j)$ // Update restrictions
12:    $resetRestrictions(j + 1, i)$ // Reset restriction LTSs in range [$j + 1$,i]
      $i \leftarrow j$ // Backtrack to $Check_j$
14: **return** counter-example from the final state space

coincides with the order in which they appear in the system. Prior to performing ICC, one can determine a suitable ICC order. For more on this, see Section 4.

Initially, $Check_1$ entails placing LTS $\mathcal{M}_1$ in parallel composition with $\mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1}$, that is, a version of the property LTS in which we have abstracted away all actions that are not present in $\mathcal{M}_1$, and on which we have applied weak trace equivalence reduction. In addition, we involve a trace equivalence reduced version of *restriction LTS* $\mathcal{R}_1$. In general, the purpose of restriction LTS $\mathcal{R}_i$ is to enable iterating over the possible traces through $\mathcal{M}_i$. We place $\mathcal{M}_i$ in a fully synchronised parallel composition with $\mathcal{R}_{i\downarrow}$ (line 3). Every time we have learned that at least one selected trace through some $\mathcal{M}_j$ cannot be part of a counter-example, we update $\mathcal{R}_j$ in such a way, that this trace is no longer accepted by $\mathcal{R}_j$, and thereby cannot be produced anymore by $\mathcal{M}_j \,|||\, \mathcal{R}_{j\downarrow}$. More on the restriction LTSs and how updating is done in the next subsection. Initially, $\mathcal{R}_i$ accepts all possible traces that can be produced by $\mathcal{M}_i$.

Verifying whether we can reach an accepting state in $(\mathcal{M}_1 \,|||\, \mathcal{R}_{1\downarrow}) \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_1}$ will produce one of two possible results: the first possibility is that an accepting state was detected ($result =$ **false**). In that case, we extract all explored and accepted behaviour of $\mathcal{M}_1$ from the state space explored so far, using the previously described procedure, which results in an LTS $\mathcal{L}_1$ (line 5). After that, we increment i (line 6).

The second option is that no accepting state was reachable. Then, we may conclude that $\mathcal{M}$ satisfies φ, since we are considering an over-approximation of the behaviour of $\mathcal{M}_1$ within the context given by $\mathcal{M}$ (parallel composition with other components can only restrict $\mathcal{M}_1$ (Def. 2)). Since after the first check, we have $i = 1$, the algorithm returns **true** and terminates (lines 7-8).

In iterations $i > 1$, we construct a verification task $Check_i$ by combining the selected traces $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$ from previous iterations with $\mathcal{M}_i$, $\mathcal{R}_i$ and the property LTS at the right level of abstraction, i.e. $\mathcal{P}^{\neg\varphi}{}_{\downarrow \bigcup_{k \in 1..i} \mathcal{A}_k}$. When performing $Check_i$, we determine whether the partial counter-example obtained so far involving $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$, represented by LTSs $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$, is allowed by $\mathcal{M}_i$.

If so, then again, we extract from the state space explored so far the accepted traces of $\mathcal{M}_i$, create an LTS $\mathcal{L}_i$ exactly containing these traces, and increment i (lines 5-6).

Alternatively, we should identify the smallest $j < i$ for which $\mathcal{L}_j$ caused $Check_i$ to not result in finding an accepting state (line 10, we skip lines 7-8 since $i > 1$). This can be achieved as follows, performing at most $i-2$ subsequent checks $Check'_1, \ldots, Check'_{i-2}$, where each check $Check'_l$ ($l \in 1..i-2$) is defined as follows:

$$Check'_l = \text{explore } \mathcal{L}_1 \mid\mid \cdots \mid\mid \mathcal{L}_l \mid\mid (\mathcal{M}_i \mid\mid\mid \mathcal{R}_{i\downarrow}) \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \bigcup_{k \in 1..l \cup \{i\}} \mathcal{A}_k}$$

When performing the checks in the order specified by their indices, then as soon as one of these checks results in not reaching an accepting state, we have found the smallest j and can stop this procedure. If all checks result in reaching an accepting state, then we select $j = i - 1$. It is important that we find the smallest j, as opposed to directly selecting $i-1$, since failure to backtrack as far as possible up the ICC order of components will result in performing redundant checks.

Next, we have to reject the current combination of traces $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$, and we do this using the value of j. Namely, we update the restriction LTS $\mathcal{R}_j$ of $\mathcal{M}_j$ in procedure *updatePreviousRestrictions* (line 11). In this case, instead of extracting the accepted traces of $\mathcal{M}_i$ from the state space explored so far, we extract a constraint concerning $\mathcal{L}_j$ from the state space that resulted either from the final $Check'_j$ (if $j < i-1$) or from $Check_i$ (if $j = i-1$). This can be done using almost the same procedure that is used to extract accepted traces of $\mathcal{M}_i$ (except that we skip step 1, since no accepting state was reached) provided that the state space was adequately annotated with additional information during construction. After constructing the constraint, procedure *updatePreviousRestrictions* adds this constraint to $\mathcal{R}_j$. How to extract constraints and update restriction LTSs is explained in detail in the next section.

Having updated $\mathcal{R}_j$, we reset all restriction LTSs in the range $[j+1, i]$, since those restrictions were only relevant for the combination of traces $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$ (line 12), and jump back to verification task $Check_j$ (line 13).

Finally, if at any moment, $i > n$, then we have successfully constructed a complete counter-example. This result is returned at line 14.

3.2 Constraints and Restriction LTSs

Extracting a constraint LTS. Whenever a check at line 3 of Alg. 1 has failed to reach an accepting state, and subsequently, the smallest index j has been identified corresponding with an $\mathcal{L}_j$ that causes accepting states to be unreachable (line 10), we must extract relevant information from the corresponding state space to update the restriction LTS $\mathcal{R}_j$.

In order to make this possible in the first place, we annotate, while constructing, the state space resulting from each check with information regarding

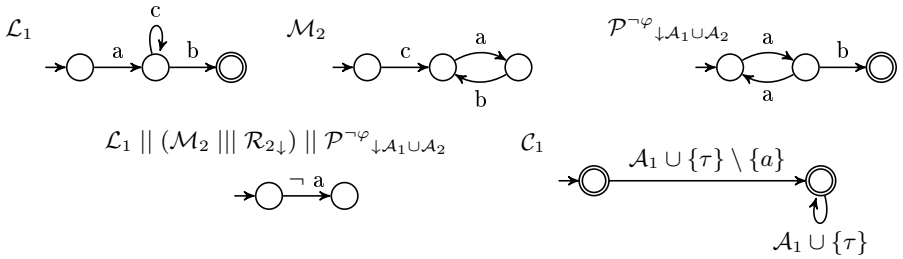

Fig. 2: Example with two LTSs and the property "after an odd amount of a's no b can be performed". The state space is constructed with annotations expressing which behaviour of $\mathcal{L}_1$ is not possible. From the result, a constraint $\mathcal{C}_1$ can be constructed.

the *impossibility* to perform behaviour of the component directly preceding component $\mathcal{M}_i$, i.e., component $\mathcal{L}_{i-1}$ in the checks $Check_i$ at line 3, and $\mathcal{L}_l$ in the checks $Check'_l$ performed at line 10.

Again consider the example system in Fig. 1. As illustrated in Section 3.1, after the first check, a path is found representing the traces ac^*b. Based on this, we construct an LTS $\mathcal{L}_1$ that accepts exactly these traces. Now, the setup is as illustrated in Fig. 2, and ICC moves on to the next check, which involves exploring $\mathcal{L}_1 \,||\, (\mathcal{M}_2 \,|||\, \mathcal{R}_{2\downarrow}) \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow\mathcal{A}_1\cup\mathcal{A}_2}$. This is a rather straightforward task in this case, since the outgoing transition from the initial state of $\mathcal{L}_1$ cannot synchronise with behaviour of $\mathcal{M}_2$, and hence the exploration is finished. But instead of only producing a single state with no transitions, we add a special *sink state* and a transition from the initial state to that sink state labelled $\neg a$, to make explicit that at that point in the exploration, an a-transition of $\mathcal{L}_1$ was not enabled. In general, we annotate each state in a state space in this manner, and furthermore, we also mark states in which the 'preceding' component state is accepting, but the overall system state is not, with a selfloop labelled $\neg accept$.

The purpose of doing this is that with the additional information, it is possible to construct a *constraint LTS* based on the result of the check. Again consider the example. Similar to the procedure of extracting accepting traces from a state space, we first action hide the state space w.r.t. $\mathcal{A}_1$ and reduce the outcome w.r.t. weak trace equivalence. Next, we add a new accepting sink state, and make the LTS complete w.r.t. $\mathcal{A}_1 \cup \{\tau\}$, by adding transitions from each state s to the sink state for all labels in $\mathcal{A}_1 \cup \{\tau\}$ not occurring already on an outgoing transition of s (either normally or in negated form) and adding selfloops for all actions in $\mathcal{A}_1 \cup \{\tau\}$ to the sink state. Next, we make all states without a $\neg accept$-selfloop accepting, and finally, remove all transitions with a negated label from the LTS. Note that in the example, the resulting constraint LTS $\mathcal{C}_1$ accepts all traces except for the traces starting with an a. For convenience, we have labelled transitions with sets of actions here, to indicate that for every action in the set, a transition exists from the indicated source state to the indicated target state.

Updating a restriction LTS. For each $\mathcal{M}_i$, we maintain a restriction LTS $\mathcal{R}_i$ to allow iterating over the traces through $\mathcal{M}_i$. Initially, for each $\mathcal{M}_i$, the structure of $\mathcal{R}_i$ is as illustrated in Figure 3: there is a single state which is initial and accepting and it has selfloops for all labels in $\mathcal{A}$ and for τ.

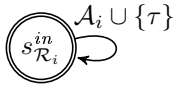

Fig. 3: Initial $\mathcal{R}_i$

First of all, note that this LTS is deterministic; this is required to prevent the state space of $\mathcal{M}_i \,|||\, \mathcal{R}_i$ from becoming very large. When updating $\mathcal{R}_i$ with new constraints, we make sure that $\mathcal{R}_i$ remains deterministic. Secondly, note that the initial $\mathcal{R}_i$ does not actually restrict the behaviour of $\mathcal{M}_i$ in $\mathcal{M}_i \,|||\, \mathcal{R}_i$, since all traces in $(\mathcal{A}_i \cup \{\tau\})^*$ are accepted by it.

With this in mind, updating a restriction LTS $\mathcal{R}_i$ with a constraint $\mathcal{C}$ can be performed by computing the language intersection [40] of $\mathcal{R}_i$ and $\mathcal{C}$, i.e., an $\mathcal{R}'_i$ is constructed such that the language of $\mathcal{R}'_i$ (the set of accepted traces) is equal to the language of $\mathcal{R}_i$ intersected with the language of $\mathcal{C}$. In this way, we remove the bad behaviour that is encoded in $\mathcal{C}$. The intersection of LTSs is defined by Def. 8.

Definition 8 (Intersection of LTSs with accepting states). *Given two LTSs with accepting states that have a total transition relation* $\mathcal{G}_1 = \langle \mathcal{S}_1, \mathcal{A}, \mathcal{T}_1, s_1^{in}, \mathcal{F}_1 \rangle$ *and* $\mathcal{G}_2 = \langle \mathcal{S}_2, \mathcal{A}, \mathcal{T}_2, s_2^{in}, \mathcal{F}_2 \rangle$ *(note that they have the same alphabet), we call* $\mathcal{K} = \mathcal{G}_1 \cap \mathcal{G}_2$ *the* intersection *of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$*. Its LTS is defined as* $\mathcal{K} = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{A}, \mathcal{T}_\mathcal{K}, \langle s_1^{in}, s_2^{in} \rangle, \mathcal{F}_\mathcal{K} \rangle$*, where:*

- $\mathcal{T}_\mathcal{K} = \{\langle s_1, s_2 \rangle \xrightarrow{\ell} \langle s'_1, s'_2 \rangle \mid \ell \in \mathcal{A}, s_1 \xrightarrow{\ell}_{\mathcal{T}_1} s'_1, s_2 \xrightarrow{\ell}_{\mathcal{T}_2} s'_2\}$;
- $\mathcal{F}_\mathcal{K} = \{\langle s_1, s_2 \rangle \mid s_1 \in \mathcal{F}_1, s_2 \in \mathcal{F}_2\}$.

Note that the intersection of $\mathcal{G}_1$ and $\mathcal{G}_2$ can actually be computed by constructing the state space of $\mathcal{G}_1 \,||\, \mathcal{G}_2$. By applying trace equivalence reduction on the resulting LTS, and involving the reduced version in subsequent ICC checks (see line 3 of Alg. 1), we restrict state space explosions caused by parallel composition as much as possible. Furthermore, by our interpretation of accepting system state, note that a system state is only accepting if the involved state in the corresponding restriction LTS is also accepting, i.e., if the restriction LTS accepts the trace.

Finally, resetting a restriction LTS, as referred to at line 12 of Alg. 1, amounts to reverting it to its initial structure. One possible optimisation involves updating the initial restriction LTSs whenever applicable, such that resetting a restriction LTS does not always mean that all the learned restrictions are discarded. For more on this, see Section 4.

3.3 Soundness and completeness

We provide an informal proof that ICC is both sound and complete. Completeness relies on the fact that the state space is finite-state, and hence from a finite number of states it is possible to reach an accepting state.

Lemma 1. *Alg. 1 is sound and complete: it returns* ***true*** *if and only if* $\mathcal{M} \vDash \phi$.

Proof. We split the proof into two parts, one for each direction.

- $\Rightarrow$ The result *true* implies that at some point, $Check_1$ returned *true*. This value indicates that no new accepted trace could be generated from $(\mathcal{M}_1 \,|||\, \mathcal{R}_{1\downarrow}) \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow\mathcal{A}_1}$. Since we have tried all traces of $\mathcal{M}_1$ that are accepted by $\mathcal{P}^{\neg\varphi}$, and each has been rejected by other checks involving other components, there is no path in $\mathcal{M}$ with a trace accepted by $\mathcal{P}^{\neg\varphi}$. Therefore, the property holds ($\mathcal{M} \vDash \phi$).
- $\Leftarrow$ $\mathcal{M} \vDash \phi$ implies that there is no trace accepted by $\mathcal{M} \,||\, \mathcal{P}^{\neg\varphi}$. Therefore, there is also no trace accepted by $\mathcal{M}_1 \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow\mathcal{A}_1}$ that is accepted by the other components. The traces accepted by $\mathcal{M}_1 \,||\, \mathcal{P}^{\neg\varphi}{}_{\downarrow\mathcal{A}_1}$ can be captured in finitely many LTSs $\mathcal{L}_1$, $\mathcal{L}'_1$, ..., since there are only finitely many states in $\mathcal{M}_1$. The traces in each of these LTSs will be rejected by a subsequent check in ICC. Therefore, after having considered all these LTSs, execution of $Check_1$ returns *true*, causing the procedure to return *true*. □

4 Optimisations

The basic ICC procedure, as explained in the previous section, is correct, but its performance in practice highly depends on applying several optimisations. In this section, we discuss the ones we identified and implemented. Identifying more opportunities to further optimise ICC remains future work.

Heuristics to select an initial component order. In Algorithm 1, the *ICC order* of the components $\mathcal{M}_1, \ldots, \mathcal{M}_n$ is fixed to the order in which they appear in the system. However, this is not required. In fact, it seems more reasonable to base such an order on the *dependency w.r.t.* φ. In general, the dependency relation D can be defined as follows:

$$D = \{(i,j) \mid i,j \in 1..n \wedge \mathcal{A}_i \cap \mathcal{A}_j \neq \emptyset\}$$

Relation D can be used to partition the components based on their dependency distance from $\mathcal{P}^{\neg\varphi}$. If we say that $\mathcal{P}^{\neg\varphi}$ has index $n+1$ in the combination of $\mathcal{M}$ and $\mathcal{P}^{\neg\varphi}$, then we place all components directly related via D to $n+1$ in one equivalence class E_1, all components with an index directly related to at least one of the components in E_1 in an equivalence class E_2, etc. Then, when choosing an order, we first select all components from E_1, then those from E_2, and so on. Within an equivalence class, a further ordering can be applied, for instance based on the number of states in the LTSs, or the number of transitions that require synchronisation with preceding LTSs in the ICC order. In our implementation, we currently use relation D and do not try to further order the LTSs in each class, but we are planning to investigate this further in the future.

Dynamically changing the ICC order during analysis. In Alg. 1, the ICC order, once selected, remains fixed during execution of ICC. This is not necessary for the procedure to be correct. In fact, it may be fruitful to frequently change the position of components in the order. So far, we have identified two situations in which changing the order frequently affects the performance of ICC positively.

First of all, consider the situation that after a $Check_i$ has returned *true*, with $i > 1$, at line 3 of Alg. 1, an LTS $\mathcal{L}_j$ is identifed at line 10 to be rejected. Just before line 13, in which we move to component $\mathcal{M}_j$ to perform the next check, it may be smart to move $\mathcal{M}_i$ in the ICC order to the position just after $\mathcal{M}_j$, i.e., to position $j + 1$. Apparently, the behaviour relevant for $\mathcal{P}^{\neg\varphi}$ of $\mathcal{M}_j$ depends to some extent on the behaviour of $\mathcal{M}_i$, making it likely that the next traces selected for $\mathcal{M}_j$ in $Check_j$, if they have to be rejected, will also be rejected by $\mathcal{M}_i$.

Second of all, another place where the order can be reconsidered is just before the next check is performed (line 3 of Alg. 1). Based on the selected traces accepted by $\mathcal{L}_{i-1}$ the next component can be selected. For instance, the shortest trace accepted by $\mathcal{L}_{i-1}$ can be identified, and from the set of components still to be involved in a check, we select one of the components with the strongest dependency (in terms of number of actions and/or transitions) on that trace.

In our implementation, we have incorporated both strategies to dynamically change the order. In the second case, we use the number of actions in the shortest trace that need to synchronise with a component to select the next component for a check. Changing the order can also be done in a number of ways; we have chosen to shift each component at position $i + 1$ or higher to the right, where the component ending up at position $n + 1$ is moved to position $i + 1$, until the selected component has ended up at position $i + 1$. An alternative possibility is to swap the positions of two components, but in that way, the initially selected ICC order tends to be erased more quickly.

One final remark about changing the ICC order of components: in order not to make the procedure incorrect, the restriction LTSs of components that are moved to the left should be reset. The reason for this is that before moving such a component, say $\mathcal{M}_i$, the constraints learned about $\mathcal{M}_i$ depend on the traces that have been selected for $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$, represented by $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$. Once any of these are changed, the constraints learned so far for $\mathcal{M}_i$ have to be reset, similar to how the constraints also need to be reset when new constraints are added to the restriction LTS of a previous component (lines 11-12 in Alg. 1). This observation directly leads us to the next possible optimisation.

Updating initial restriction LTSs. The restriction LTS of a component that is moved to the left in the ICC order needs to be reset. However, the constraints learned about $\mathcal{M}_1$ actually never require this, since that component cannot be moved to the left. This can be further explained by noting that the validity of the constraints for $\mathcal{M}_1$ does not depend on previously selected traces of other components being sufficient to construct a counter-example. In that respect, the contraints learned about $\mathcal{M}_1$ are more valuable for the progress of ICC then the constraints learned about any of the other components, since the former

constraints are always relevant. For this reason, these constraints can safely be added to the *initial* restriction LTS of $\mathcal{M}_1$.

In order to also learn constraints about other components that are persistent to updates applied to restriction LTSs, an additional check $Check''$ can be added right after line 10 in Alg. 1, at the moment when the smallest j has been identified. This check can be defined as follows:

$$Check'' = \text{explore } \mathcal{L}_j \mid\mid (\mathcal{M}_i \mid\mid\mid \mathcal{R}_{i\downarrow}) \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \mathcal{A}_j \cup \mathcal{A}_i}$$

The purpose of performing $Check''$ is to determine whether the traces of $\mathcal{L}_j$ should also be rejected when placed in parallel composition only with $\mathcal{M}_i$. If this is the case, then those traces should never be selected anymore. If we add this insight as a new constraint to the initial restriction LTS of $\mathcal{M}_j$, then every time $\mathcal{M}_j$'s current restriction LTS is reset, we revert to an initial restriction LTS that has these constraints still in them. In our implementation, we have added this optimisation.

First adding an abstract version of a component to a check. As a final optimisation, we propose to implement the check at line 3 of Alg. 1 in two steps instead of one. When introducing $\mathcal{M}_i$ into check $Check_i$, note that for the possible rejection of traces in $\mathcal{L}_1, \ldots, \mathcal{L}_{i-1}$, it is only relevant to consider the behaviour in $\mathcal{M}_i$ that requires synchronisation with components $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$; all other behaviour can be abstracted away. Also, the restriction LTS of $\mathcal{M}_i$ is not relevant for the rejection of traces, only for the case when traces through $\mathcal{M}_i$ can be selected to extend the current partial counter-example. In that case, the restriction LTS ensures that no traces will be selected that have been selected previously.

The possibility to only consider an abstract version of $\mathcal{M}_i$ provides the potential to reduce the size of state spaces in those cases where $\mathcal{M}_i$ rejects previously selected traces. In cases where no traces can be rejected, a subsequent check as defined at line 3 still has to be performed, since the abstract version of $\mathcal{M}_i$ does not suffice to extend the counter-example. Therefore, this proposed optimisation may primarily have a positive effect on the memory use of ICC, and to a lesser extent on the running time.

Formally, we redefine $Check_i$ at line 3 now as the following two checks $Check_i^1$ and $Check_i^2$:

$$Check_i^1 = \text{explore } \mathcal{L}_1 \mid\mid \cdots \mid\mid \mathcal{L}_{i-1} \mid\mid \mathcal{M}_{i\downarrow \bigcup_{k\in 1..i-1} \mathcal{A}_k} \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \bigcup_{k\in 1..i} \mathcal{A}_k}$$

$$Check_i^2 = \text{explore } \mathcal{L}_1 \mid\mid \cdots \mid\mid \mathcal{L}_{i-1} \mid\mid (\mathcal{M}_i \mid\mid\mid \mathcal{R}_{i\downarrow}) \mid\mid \mathcal{P}^{\neg\varphi}{}_{\downarrow \bigcup_{k\in 1..i} \mathcal{A}_k}$$

This optimisation has also been incorporated in our implementation of ICC.

5 Experiments

To validate the effectiveness of ICC, we conducted a number of representative experiments, using the DAS-5 cluster [4], with nodes equipped with an INTEL

Haswell E5-2630-v3 2.4 GHz CPU, 64 GB memory, and running CentOS Linux 7.2. The selected models have been taken from various sources, namely the BEEM benchmark set [37], the CADP toolbox distribution [24], and the mCRL2 toolset distribution [15]. Table 1 lists the models, together with their state space characteristics, the type of safety property checked, and whether or not the property holds. The models suffixed '.1' are altered versions of the standard models. The alterations resulted in larger state spaces.

Regarding the property types, *limited action occurrence* states that at most two occurrences of a given action a are allowed between two consecutive occurrences of another action b. The *mandatory precedence* property says that an action a is always preceded by an action b. *Bounded response* states that after an occurrence of a, a b of a given set of actions must occur. *Limited action exclusion* is a property in which an action a cannot occur between two consecutive occurrences of actions b and c. In *exact occurrence number*, it is required that an action a occurs an exact number of times, if action b has previously occurred. *Mutual exclusion* refers to the standard property regarding access to critical sections.

We compared the following approaches:

- ICC refers to an implementation of ICC (single-threaded) in the Refiner exploration tool [45]. We have implemented the optimisations proposed in Section 4.
- OTF refers to on-the-fly property checking. We also used Refiner for this, running on a single thread. It explores the state space, checks on-the-fly whether the property holds and terminates if a counter-example is found. Even though there exist much faster state space exploration tools, having both ICC and OTF use the same implementation results in a fair comparison. A better implementation of state space exploration could be used to speed up both ICC and OTF, since this procedure is the main performance bottleneck for both.
- PMC refers to partial model checking [23, 32]. In PMC, the state space is incrementally constructed by adding processes and minimising the intermediate results. The property can be checked once the state space is constructed. We used the (single-threaded) Pmc tool of the Cadp toolbox for this.

We have not compared ICC with other compositional model checking techniques, such as Assume-Guarantee [30, 17, 9, 27, 9, 34], since no implementations were available to us that are directly applicable on the type of models we consider, namely networks of LTSs. In future work, we plan to perform an extensive comparison between ICC and Assume-Guarantee. In this paper, we focus on determining whether ICC is effective in breaking down the classical OTF analyses into smaller checks. We have also not compared to the Spin model checker [31]. For the BEEM models we consider here, Promela models exist, however, the number of states in the resulting state spaces differ significantly from the numbers produced here.

Table 2 presents the results, providing for each approach the runtime in seconds. "T/O" indicates a timeout, which was set to 3 hours. The maximum number of states involved in a check at some point during the analysis is also

Table 1: Characteristics of the performed experiments

model	#states	#transitions	property	satisfied
1394	69,518	123,614	Limited action occurrence	Y
1394.1	563,040	1,154,447	Limited action occurrence	Y
transit	3,480,248	37,394,212	Bounded response	N
wafer_stepper.1	6,099,751	29,028,530	Mandatory precedence	N
Lamport8	62,669,317	269,192,485	Mutual exclusion	N
Lann5	993,914	3,604,487	Mutual exclusion	Y
Gas station c2	165	276	Bounded response	Y
Gas station c3	1,197	2,478	Bounded response	Y
HAVi3.2	19,554,248	80,704,326	Bounded response	Y
Peterson7	142,471,098	626,952,200	Mandatory precedence	Y
Szymanski5	79,518,740	922,428,824	Mutual exclusion	Y

reported; for OTF, this is the number of explored states, for PMC this refers to the largest LTS constructed during the construction, and for ICC, this is the maximum number of states involved in a single ICC check, i.e., the total number of states in the restriction LTSs, plus the number of explored states in the check. Finally, for ICC, also the total number of performed checks is reported (#iters.).

In terms of the maximum number of involved states, which provides an indication for the maximum amount of memory used, ICC is very effective in breaking down the monolithic analysis performed by OTF into smaller analyses, particularly when the model satisfies the property. For the Peterson7 case, only 0.00002% of the state space was ever explored in one check. In this respect, ICC was much more effective than both OTF and PMC, which timed out. Notable exceptions to this are the 1394 and the gas station models. We will further investigate the exact cause of ICC not performing very well in those cases in the near future.

It is to be expected that for models that do not satisfy the property, OTF is much more effective than the compositional model checking approaches. An important concern for the latter techniques is that the size of the state space is kept small, and therefore the state space is iteratively built. A straightforward approach that directly explores the state space may therefore run into a counter-example much more quickly. However, in the cases we considered, the runtimes of ICC and OTF were still comparable. Moreover, in two of the three cases where the property is violated, ICC outperformed PMC both in runtime and the number of explored states.

As already mentioned, ICC seems to be particularly effective when the model satisfies the property. In a number of cases, ICC was even much faster than OTF. In those cases, the rejection of tested path prefixes in ICC quickly led to rejecting all potential candidates, and more importantly, it could avoid the exploration of many states. This effect is absent when checking incorrect models. In those cases, a counter-example can be constructed, but there are also many traces that are initially promising, but need to be rejected later on.

Concluding, individual ICC checks are often very small, and the runtime of ICC is often comparable to OTF. Furthermore, it should be noted that we have not yet attempted to optimise the implementation of ICC, so it is very likely that

Table 2: Experimental results for OTF, PMC, and ICC; Times in seconds

model	property satisfied	OTF time	OTF #states	PMC time	PMC #states	ICC time	ICC #states	ICC #iters.
1394	Y	9.35	69,518	26.13	1,061	10.48	5,659	3
1394.1	Y	51.04	563,040	36.15	1,061	155.66	219,981	5
transit	N	7.76	50,970	1,044.03	1,437,433	5.69	10,443	5
wafer_stepper.1	N	20.27	60,809	68.09	3,821	18.67	28,227	8
Lamport8	N	2.66	30,041	56.52	301,711	11.78	22,552	6
Lann5	Y	289.65	993,914	T/O		1.39	33	35
Gas station c2	Y	0.08	165	0.54	342	0.91	595	11
Gas station c3	Y	0.21	1,197	12.53	4,532	9.15	4,930	21
HAVi3.2	Y	T/O		21.24	12	8.89	167	57
Peterson7	Y	T/O		T/O		7.11	34	73
Szymanski5	Y	T/O		T/O		2.67	48	21

the reported runtimes can be further improved. Finally, the frequently drastic reduction in memory use of model checking when using ICC is very encouraging regarding the scalability of ICC. We expect that we can go far beyond what can currently be analysed using OTF.

6 Related Work

Regarding compositional model checking, a number of prominent approaches need to be mentioned. First of all, *partial model checking* [23, 32] is an approach in which it is attempted to incrementally construct a state space bisimilar to the original one, without actually constructing the latter. It is attempted to keep the constructed state space small by carefully combining component LTSs and applying bisimulation reduction on the intermediate results. However, the order in which component LTSs are introduced in the analysis usually heavily influences the effectiveness of the technique, and the best order is a-priori not clear [16]. *Saturation* is similar to partial model checking, in that they both attempt to incrementally construct a version of the system state space [35]. Instead, ICC never involves more than one complete component LTS in a single check. Moreover, if a component order is initially chosen which is not efficiently leading to a solution, ICC with optimisations can change this order dynamically.

In [13, 36], it has been investigated what the best system decompositions are for a set of benchmarks, and what the best order is in which to combine components again. Even though ICC is adaptive in this respect to some extent, we experience that its performance is greatly affected by the initially selected ICC order. In the future, we will study the results in the literature on this topic.

Another approach is to impose interaction constraints and to find relevant invariants for combinations of components [5, 6]. The use of *interface automata* [1] allows reasoning about the possible interactions between components. In ICC checks, we always assume that a component being checked can interact with components not involved in the check. Subsequent checks will detect cases where this assumption was not valid. It would be interesting to investigate how the above techniques could positively influence the running time of ICC.

Assume-Guarantee (AG) [30, 17, 9, 27, 34] is another prominent technique. It construct assumptions with the goal to prove that the system satisfies the property. Given a system $M_1 \mid\mid M_2$ and property φ, it tries to establish that both M_1 satisfies a set of assumptions A, and that M_2 satisfies φ under assumptions A. If this holds, then $M_1 \mid\mid M_2$ satisfies φ. Circular AG extends this approach to N instead of 2 components, and constructs assumption LTSs using SAT solving [18].

Like restriction LTSs in ICC, assumptions are expressed in LTSs in AG. How to keep these LTSs minimal is hard, as they tend to grow rapidly. L^* [2] is frequently applied [12, 38], but sometimes, this seems to be unnecessary [39], and other approaches have been investigated as well [28, 21, 20]. For ICC, we have not experienced that the sizes of the restriction LTSs became problematic. This is probably because ICC and AG attack the problem from opposite directions: AG tries to establish that the property holds, whereas ICC tries to construct a counter-example. In AG, the goal is that an assumption LTS overapproximates a component while still reasoning about the property, whereas in ICC, the function of a restriction LTS is merely to block certain traces in the component, and can therefore often remain a much coarser approximation of the component LTS. Finally, a fundamental difference between ICC and AG is that the latter tries to avoid involving the actual component LTSs in the verification checks and instead tries to establish that the assumptions are sufficient to prove that the property is satisfied. In this way of working, it frequently happens that spurious counter-examples are constructed, so any identified counter-examples in the complete, abstract system must first be checked against the original system to establish whether the counter-example is real. ICC, on the other hand, involves component LTSs from the very start, and selects part of their behaviour for subsequent checks instead of the complete component LTSs, with the goal to keep the parallel composition of component behaviour small. As a result, ICC never produces spurious counter-examples. Only partially constructed counter-examples can be rejected in one of the checks, but once a complete counter-example has been successfully constructed, it is by definition a real one.

Counterexample-Guided Abstraction Refinement (CEGAR) [11] is a very well established technique that computes abstractions of programs and refines them based on spurious counter-examples. In spirit, ICC and CEGAR are very similar, but the latter does not operate in a compositional manner and in contrast to ICC, reasons with behavioural over-approximations of the program being verified. Finally, the same observation regarding spurious counter-examples can be made as above for Assume-Guarantee.

Although more tailored towards programs than models, *thread-modular reasoning* [22, 29] is another related technique, designed to compositionally reason about threads in multi-threaded programs. Besides the obvious similarities, the fact that the inputs of the two approaches are very different makes it hard to provide a clear comparison, or learn from these techniques to further improve ICC.

One of the motivations behind the development of ICC is to reduce the memory requirements. This makes ICC related to other memory-saving tech-

niques [19, 33, 25], but different from most other techniques of this type, we observe that besides memory savings, also the runtimes can be positively affected by ICC.

Finally, ICC is pleasantly parallel, since different ICC orders can be inspected fully independently. Other parallel techniques, such as [8, 41], still require frequent communication between workers. In the future, we plan to investigate the potential to perform ICC in parallel.

7 Conclusions

We presented a new compositional model checking technique, called Incremental Counter-example Construction. Experiments point out that it can very effectively reduce the number of states involved in a single check, thereby demonstrating great potential for scaling up the technique to larger models. Moreover, the runtime is frequently comparable to a traditional on-the-fly analysis, and in cases where the model is correct, ICC can actually by significantly faster.

Future work. ICC also seems applicable to check linear-time liveness or branching time properties. For those, checks could be performed using Nested Depth-First Search [14] or by solving Boolean Equation Systems [32], respectively. We plan to investigate this.

We also plan to investigate performing ICC in parallel, by having different threads inspect different ICC orders, perhaps in line of our earlier work [42–44, 46]. We will further investigate the potential to optimise ICC, in addition to the optimisations discussed in Section 4.

Finally, we will investigate to what extent the ICC approach is suitable for symbolic model checking techniques.

References

1. de Alfaro, L., Henzinger, T.: Interface automata. ACM SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
4. Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., Wijshoff, H.: A medium-scale distributed system for computer science research: Infrastructure for the long term. IEEE Computer 49(5), 54–63 (May 2016)
5. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Incremental component-based construction and verification using invariants. In: FMCAD. pp. 257–266. IEEE (2010)
6. Bensalem, S., Bozga, M., Legay, A., Nguyen, T.H., Sifakis, J., Yan, R.: Component-based verification using incremental design and invariants. Softw. Syst. Model. 15(427) (2016)
7. Brookes, S., Hoare, C., Roscoe, A.: A Theory of Communicating Sequential Processes. Journal of the ACM 31(3), 560–599 (1984)

8. Camilli, M., Bellettini, C., Capra, L., Monga, M.: CTL Model Checking in the Cloud Using MapReduce. In: SYNACS. pp. 333–340. IEEE (2014)
9. Chen, Y.F., Clarke, E., Farzan, A., Tsai, M.H., Tsay, Y.K., Wang, B.Y.: Automated assume-guarantee reasoning through implicit learning. In: CAV. LNCS, vol. 6174, pp. 511–526. Springer (2010)
10. Clarke, E., Long, D., McMillan, K.: Compositional Model Checking. In: LICS. pp. 353–362. IEEE (1989)
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided Abstraction Refinement. In: CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000)
12. Cobleigh, J., Giannakopoulou, D., Păsăreanu, C.: Learning Assumptions for Compositional Verification. In: TACAS. LNCS, vol. 2619, pp. 331–346. Springer (2003)
13. Cobleigh, J., Avrunin, G., Clarke, L.: Breaking Up is Hard to Do: An Evaluation of Automated Assume-Guarantee Reasoning. ACM Trans. on Software Engineering and Methodology 17(2), 7 (2008)
14. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: CAV. LNCS, vol. 531, pp. 233–242. Springer (1990)
15. Cranen, S., Groote, J., Keiren, J., Stappers, F., de Vink, E., Wesselink, W., Willemse, T.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: TACAS. LNCS, vol. 7795, pp. 199–213. Springer (2013)
16. Crouzen, P., Lang, F.: Smart Reduction. In: FASE. LNCS, vol. 6603, pp. 111–126. Springer (2011)
17. Elkader, K., Grumberg, O., Păsăreanu, C., Shoham, S.: Automated Circular Assume-Guarantee Reasoning. In: FM. LNCS, vol. 9109, pp. 23–39. Springer (2015)
18. Elkader, K., Grumberg, O., Păsăreanu, C., Shoham, S.: Automated Circular Assume-Guarantee Reasoning with N-way Decomposition and Alphabet Refinement. In: CAV(1). LNCS, vol. 9779, pp. 329–351. Springer (2016)
19. Evangelista, S., Pradat-Peyre, J.F.: Memory Efficient State Space Storage in Explicit Software Model Checking. In: SPIN. LNCS, vol. 3639, pp. 43–57. Springer (2005)
20. Finkbeiner, B., Peter, H.J., Schewe, S.: RESY: Requirement Synthesis for Compositional Model Checking. In: TACAS. LNCS, vol. 4963, pp. 463–466. Springer (2008)
21. Finkbeiner, B., Schewe, S., Brill, M.: Automatic Synthesis of Assumptions for Compositional Model Checking. In: FORTE. LNCS, vol. 4229, pp. 143–158. Springer (2006)
22. Flanagan, C., Freund, S., Qadeer, S., Seshia, S.: Modular Verification of Multithreaded Programs. TCS 338(1-3), 153–183 (2005)
23. Garavel, H., Lang, F., Mateescu, R.: Compositional verification of asynchronous concurrent systems using CADP. Acta Informatica 52(4-5), 337–392 (2015)
24. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2), 89–107 (2013)
25. Geldenhuys, J.: State Caching Reconsidered. In: SPIN. LNCS, vol. 2989, pp. 23–38. Springer (2004)
26. Groote, J.F., Wijs, A.: An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In: TACAS. LNCS, vol. 9636, pp. 607–624. Springer (2016)
27. Grumberg, O., Meller, Y.: Learning-Based Compositional Model Checking of Behavioral UML Systems. In: Dependable Software Systems Engineering, NATO Science for Peace and Security Series - D: Information and Communication Security, vol. 45, pp. 117–136. IOS Press (2016)

28. Gupta, A., McMillan, K., Fu, Z.: Automated Assumption Generation for Compositional Verification. In: CAV. LNCS, vol. 4590, pp. 420–432. Springer (2007)
29. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A Constraint-based Verifier for Multi-Threaded Programs. In: CAV. LNCS, vol. 6806, pp. 412–417. Springer (2011)
30. Henzinger, T., Qadeer, S., Rajamani, S.: You assume, we guarantee: methodology and case studies. In: CAV. LNCS, vol. 1427, pp. 440–451. Springer (1998)
31. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
32. Lang, F., Mateescu, R.: Partial Model Checking using Networks of Labelled Transition Systems and Boolean Equation Systems. Logical Methods in Computer Science 9(4) (2013)
33. Mateescu, R., Wijs, A.: Hierarchical Adaptive State Space Caching Based on Level Sampling. In: TACAS. LNCS, vol. 5505, pp. 215–229. Springer (2009)
34. Mendoza, L., Capel, M., Pérez, M., Benghazi, K.: Compositional Model-Checking Verification of Critical Systems. In: Enterprise Information Systems, LNBIP, vol. 19, pp. 213–225. Springer (2008)
35. Molnár, V., Vörös, A., Darvas, D., Bartha, T., Majzik, I.: Component-wise incremental LTL model checking. Formal Aspects of Computing 28(3), 345–379 (2016)
36. Nam, W., Alur, R.: Learning-Based Symbolic Assume-Guarantee Reasoning with Automatic Decomposition. In: ATVA. LNCS, vol. 4218, pp. 170–185. Springer (2006)
37. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: SPIN. LNCS, vol. 4595, pp. 263–267. Springer (2007)
38. Păsăreanu, C., Giannakopoulou, D., Bobaru, M., Cobleigh, J., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
39. Siirtola, A., Tripakis, S., Heljanko, K.: When Do We (Not) Need Complex Assume-Guarantee Rules? In: ACSD. pp. 30–39. IEEE (2015)
40. Sudkamp, T.: Languages and machines - an introduction to the theory of computer science. Addison-Wesley (1988)
41. Verstoep, K., Bal, H., Barnat, J., Brim, L.: Efficient large-scale model checking. In: IPDPS. pp. 1–12. IEEE (2009)
42. Wijs, A.: The HIVE Tool for Informed Swarm State Space Exploration. In: PDMC. Electronic Proceedings in Theoretical Computer Science, vol. 72, pp. 91–98. Open Publishing Association (2011)
43. Wijs, A.: Towards Informed Swarm Verification. In: NASA Formal Methods. LNCS, vol. 6617, pp. 422–437. Springer (2011)
44. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT 18(2), 169–185 (2016)
45. Wijs, A., Engelen, L.: REFINER: Towards Formal Verification of Model Transformations. In: NASA Formal Methods. LNCS, vol. 8430, pp. 258–263. Springer (2014)
46. Wijs, A., Lisser, B.: Distributed Extended Beam Search for Quantitative Model Checking. In: MoChArt. LNAI, vol. 4428, pp. 165–182. Springer (2007)