Introducing Fairness into Compositional Verification via Unidirectional Counters

Antti Siirtola*, Antti Puhakka†, Gerald Lüttgen‡
*Department of Information Processing Science, University of Oulu, P.O.B. 3000, 90014 University of Oulu, Finland
antti.siirtola@oulu.fi
†Bitwise Oy, Viinikankatu 1 C, 33100 Tampere, Finland
antti.puhakka@bitwise.fi
‡Faculty of Information Systems and Applied Computer Sciences, University of Bamberg, 96045 Bamberg, Germany
gerald.luettgen@swt-bamberg.de

***Abstract*—We equip labelled transition systems (LTSs) with unidirectional counters (UCs) which can be initialised to an arbitrary positive value and decremented but not incremented. This formalism, called UCLTSs, enables one to express fairness properties of concurrent systems in a finite form and reason about them in a compositional, refinement-based fashion. Technically, we first show how to apply CSP-style parallel composition and hiding directly on UCLTSs while maintaining compositionality. As our main result, we prove that the refinement checking of UCLTSs under the chaos-free failures-divergences semantics can be reduced to two decidable tasks that can be solved by the popular FDR2 and NuSMV model checkers, respectively.**

***Keywords*-formal verification, process algebra, unidirectional counter, refinement, fairness, decidability.**

I. Introduction

We consider the algorithmic verification of concurrent reactive systems where liveness properties play an important role [1], [2]. A problem with the verification of such properties is that they may be violated in executions where an (unwanted) event, like the loss of a message, occurs infinitely often consecutively. As the probability of such scenarios is typically zero, liveness properties can usually be proved correct when the system behaves in a fair way, e.g., a message is not lost infinitely often without being delivered.

On the one hand, in temporal-logic model checking [1], fairness is a well-studied problem. However, these approaches do not easily lend support to compositional analysis as it is not generally possible to replace a subsystem with a semantically equivalent one while preserving the semantics of the whole. Practical implications are that many state-space reduction techniques cannot be applied during the construction of the system and that generic parameterized verification techniques [3], [4] cannot be used.

On the other hand, in process algebras, where both system implementation and specification are processes and verification is done by checking whether the implementation is a refinement of the specification, compositionality is a standard feature [2]. However, introducing fairness into a process algebra without losing compositionality has turned out to be difficult to achieve. This is because most compositional refinement preorders are either too fine-grained for practical verification or do not preserve enough information on the enabledness of transitions in infinite executions [5].

Our research is motivated by the work of Puhakka who has introduced a congruence preserving way to use fairness constraints in process-algebraic verification [5]. In his approach, fairness constraints are formulated as temporal logic formulae that correspond to infinite-state labelled transition systems (LTSs) [2]. When an (unfair) finite-state process is composed in parallel with the fairness constraint and internal communication is hidden, the resulting infinite-state LTS can be represented as (or at least abstracted to) a finite one. Puhakka provides an algorithm for generating the abstraction, which basically means removing unfair divergences, i.e., livelocks. He also sketches an algorithm for checking the completeness of the abstraction. However, if the abstraction turns out to be incomplete, the method may result in false negative or, if it is applied on the specification side, false positive verification results.

Contribution: The fundamental source of infinity in Puhakka's construction is that certain actions are allowed to occur arbitrarily but not infinitely often unless some other action takes place every now and then. This can be thought as an implicit use of a unidirectional counter (UC), which can be initialised to an arbitrary positive value and decremented but not incremented. We extend Puhakka's work by equipping LTSs explicitly with such counters. Although this does not increase the expressiveness of finite LTSs from the viewpoint of finite behaviours (traces and failures [2], [6]), UCs enable expressing infinite behaviours (divergences and infinite traces) that would normally require the use of infinite-state LTSs. UCs can be employed to impose strong fairness constraints on infinite executions, which means that they are useful in modelling fairness properties and introducing fairness in a system model.

As our first contribution, we give operational semantics to an LTS with UCs (UCLTS) as an infinite-state LTS. Moreover, we provide UCLTSs with typical process-algebraic operators, namely CSP-style parallel composition and hiding operators [2], and show that every compositional refinement preorder on LTSs, like CFFD (Chaos-Free Failures-Divergences) semantics [7], is a compositional preorder on

UCLTSs, too.

As our main contribution, we show how the refinement checking of finite UCLTSs under CFFD semantics can be reduced to two decidable tasks: a failures refinement check on finite LTSs and a language containment check on Streett automata (SAs) [8]. The former task can be solved using the refinement checker FDR2 [2]. The latter task [9] in general involves the complementation of an SA, which is of exponential complexity also in the best case [10]. However, if the specification automaton is deterministic, then the language containment problem for SAs can be formulated as a temporal logic model checking task [11] and solved using existing tools such as NuSMV [12]. This enables the *compositional* verification of systems in the presence of fairness constraints.

Note that our work using UCLTSs forms a proper extension of Puhakka's approach since our reduction of CFFD refinement checking to the above two tasks preserves all finite and infinite behaviours of a process and nothing more or less, i.e., no over- or under-approximation is introduced. This is especially important from the viewpoint of specification processes that cannot be over-approximated without the risk of false positive verification results.

Related Work: Other efforts to combine compositionality and fairness have focused on developing operators and semantics that neglect unfair executions [13], [14], [15]. The fair parallel composition operator of Hennessy guarantees that the parallel composition of two processes makes infinite progress only if both processes make infinite progress [13]. The fair fixed-point operator of Cleaveland and Lüttgen allows for the finite but unbounded unwinding of recursion [14], and the semantics used by Rensink and Vogler ignores leavable livelocks [15].

Unlike these approaches and since UCLTSs with parallel composition and hiding are finite representations for fairly behaving infinite-state LTSs with CSP-style parallel composition and hiding, we basically employ an existing process model with standard operators and well-known *compositional* semantics. A benefit of this approach is that it can be implemented on top of existing tools.

Furthermore, although the approaches in [13], [14], [15] are based on semantics and process models incomparable to ours, some of their aspects can be simulated by our formalism, too. More precisely, a fair parallel composition can be realised by using two UCs that guarantee that, if one process executes an action infinitely often, then the other process must execute an action infinitely often, too. A fair fixed-point operator can be implemented with the aid of a single UC because the maximum number of unwindings is chosen arbitrarily when a recursion is entered and, every time a recursion is unwound, the number is decremented until it becomes zero or the recursion is exited. Leavable livelocks can be achieved by decrementing a UC when a process takes an internal action that is an alternative to a visible one, and by initialising the UC when the process takes a visible action. Hence, the combination of compositionality and UCs is a universal construct able to express many forms of fairness.

Proofs: Due to space constraints, all proofs of our results can be found in an online appendix [16].

II. MODELS OF COMPUTATION

Labelled Transition Systems (LTSs): An LTS [2] is a graph whose vertices are called states, edges are labelled by actions and called transitions, and at least one of the states is marked as initial. We assume that there is a unique *invisible* action, denoted by τ, while the other actions are *visible*. Formally, an LTS is a four-tuple (S, A, R, I), where S is a non-empty set of *states*, A is a set of visible actions (an *alphabet*), $R \subseteq S \times (A \cup \{\tau\}) \times S$ is a set of *transitions*, and $I \subseteq S$ is a non-empty set of *initial* states.

An LTS $\mathcal{L}$ is *finite* if its alphabet is finite and if it has only finitely many states. Otherwise, the LTS is *infinite*. A finite alternating sequence $s_0 a_1 s_1 \ldots a_n s_n$ of states and actions of $\mathcal{L}$ is a *(finite) path in $\mathcal{L}$ (from s_0) (to s_n)* if (s_{i-1}, a_i, s_i) is a transition of $\mathcal{L}$ for every $i \in \{1, \ldots, n\}$. A finite path from an initial state is called a *(finite) execution (of $\mathcal{L}$)*. An *infinite path* and an *infinite execution* are defined analogously.

In process algebraic verification, both system specification and implementation are modelled as LTSs, denoted by $\mathcal{L}_{spec}$ and $\mathcal{L}_{impl}$, respectively. The system LTS $\mathcal{L}_{impl}$ is typically a parallel composition of smaller LTSs representing its components and, before it is compared against $\mathcal{L}_{spec}$, the actions irrelevant to $\mathcal{L}_{spec}$ are hidden.

Let $\mathcal{L}_i$ be an LTS (S_i, A_i, R_i, I_i) for $i = 1, 2$. The *parallel composition* of LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted by $(\mathcal{L}_1 \parallel \mathcal{L}_2)$, is a four-tuple $(S_1 \times S_2, A_1 \cup A_2, R_\parallel, I_1 \times I_2)$, where $R_\parallel$ is the set of all triples $((s_1, s_2), a, (s'_1, s'_2))$ such that $(s_i, a, s'_i) \in R_i$ for $i = 1, 2$ and $a \neq \tau$; or $(s_i, a, s'_i) \in R_i$, $a \notin A_j$, $s_j \in S_j$ and $s_j = s'_j$ for different elements i, j in $\{1, 2\}$.

It is easy to see that $(\mathcal{L}_1 \parallel \mathcal{L}_2)$ is an LTS, where $\mathcal{L}_1$ and $\mathcal{L}_2$ can execute a visible action jointly if and only if both agree on its execution, whereas the visible actions only in the alphabet of one LTS and the invisible action τ are executed individually (i.e., interleaved). This is essentially the associative and commutative parallel composition operator of CSP [2], with the synchronization alphabet consisting of the intersection of the LTSs' alphabets.

Let $\mathcal{L} = (S, A, R, I)$ be an LTS and B a set of visible actions. $\mathcal{L}$ *after hiding* B, denoted by $(\mathcal{L} \setminus B)$, is a four-tuple $(S, A \setminus B, R_\setminus, I)$, where $R_\setminus$ is the set of all triples $(s, a, s') \in R$ such that $a \notin B$; or $a = \tau$ and there is some $b \in B$ such that $(s, b, s') \in R$. Hence, $(\mathcal{L} \setminus B)$ is obtained from $\mathcal{L}$ by substituting τ for the actions in B. Obviously, $(\mathcal{L} \setminus B)$ is an LTS.

A system implementation $\mathcal{L}_{impl}$ is considered correct with respect to the specification $\mathcal{L}_{spec}$ if $\mathcal{L}_{impl}$ is a *refinement* of $\mathcal{L}_{spec}$, denoted by $\mathcal{L}_{impl} \preceq \mathcal{L}_{spec}$. There are many

different refinement relations reported in the literature [6], which enable different properties to be checked. However, refinement relations are typically compositional preorders, i.e., reflexive and transitive relations such that whenever $\mathcal{L}_1 \preceq \mathcal{L}_2$ then $\mathcal{L}_1 \parallel \mathcal{L} \preceq \mathcal{L}_2 \parallel \mathcal{L}$ and $\mathcal{L}_1 \setminus B \preceq \mathcal{L}_2 \setminus B$ for all LTSs $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}$ and sets B of visible actions.

Given a preorder $\preceq$ on LTSs, its kernel is the relation $\{(\mathcal{L}_1, \mathcal{L}_2) \mid \mathcal{L}_1 \preceq \mathcal{L}_2, \mathcal{L}_2 \preceq \mathcal{L}_1\}$. It is easy to see that the kernel is an equivalence and, if $\preceq$ is compositional, then the kernel is compositional, too.

CFFD Semantics: We consider refinement checking under the popular chaos-free failures-divergences (CFFD) semantics [7], which enables the analysis of safety, liveness, and deadlock properties. For this purpose, LTSs are read as sets of traces, stable failures and divergences.

Let $\mathcal{L}$ be an LTS. A finite sequence of visible actions is a *(finite) trace (of $\mathcal{L}$)* if there is an execution of $\mathcal{L}$ such that the sequence is obtained from the execution by removing all states and invisible actions. An *infinite trace* is defined analogously. The sets of all traces and infinite traces of $\mathcal{L}$ are denoted by $\mathrm{tr}(\mathcal{L})$ and $\mathrm{it}(\mathcal{L})$, respectively.

An action a is *enabled* in a state s, if there is a state s' such that (s, a, s') is a transition of $\mathcal{L}$. A state s *refuses* a set A of visible actions if no action in $A \cup \{\tau\}$ is enabled at s. The pair (t, A) of a trace and a set of actions is a *stable failure (of $\mathcal{L}$)*, or a *failure* for short, if there is an execution of $\mathcal{L}$ to a state refusing A such that t is obtained from the execution by removing all states and invisible actions. The set of all failures of $\mathcal{L}$ is denoted by $\mathrm{sf}(\mathcal{L})$.

A finite trace t of $\mathcal{L}$ is a *divergence (of $\mathcal{L}$)* if there is an infinite execution such that t is obtained from the execution by removing all states and invisible actions. The set of all divergences of $\mathcal{L}$ is denoted by $\mathrm{dv}(\mathcal{L})$.

An LTS $\mathcal{L}_1$ is a *failures refinement* of an LTS $\mathcal{L}_2$, written $\mathcal{L}_1 \preceq_{\mathrm{F}} \mathcal{L}_2$, if $\mathcal{L}_1$ and $\mathcal{L}_2$ have the same alphabet, $\mathrm{tr}(\mathcal{L}_1) \subseteq \mathrm{tr}(\mathcal{L}_2)$ and $\mathrm{sf}(\mathcal{L}_1) \subseteq \mathrm{sf}(\mathcal{L}_2)$. LTS $\mathcal{L}_1$ is a *CFFD refinement* of $\mathcal{L}_2$, written $\mathcal{L}_1 \preceq_{\mathrm{CFFD}} \mathcal{L}_2$, if $\mathcal{L}_1 \preceq_{\mathrm{F}} \mathcal{L}_2$, $\mathrm{dv}(\mathcal{L}_1) \subseteq \mathrm{dv}(\mathcal{L}_2)$ and $\mathrm{it}(\mathcal{L}_1) \subseteq \mathrm{it}(\mathcal{L}_2)$ [7]. The corresponding kernels are denoted by $=_{\mathrm{F}}$ and $=_{CFFD}$. CFFD refinement is a compositional preorder, and $=_{CFFD}$ is a compositional equivalence [7].

Streett Automata (SAs): An ω-automaton is a finite state automaton that accepts infinite words. The most common ω-automata are Büchi, Muller, Rabin, Streett and parity automata, which differ in their acceptance conditions. However, except for deterministic Büchi automata, they all recognise the same class of languages [8]. Here, we use an SA whose acceptance condition consists of a finite number of (strong) fairness constraints:

An SA is a five-tuple $\mathcal{S} := (Q, \Sigma, \Delta, \hat{q}, F)$, where Q is a finite non-empty set of *states*, Σ is a finite set of *input symbols* (an *alphabet*), $\Delta \subseteq Q \times \Sigma \times Q$ is a set of *transitions*, $\hat{q} \in Q$ is the *initial state*, and $F \subseteq \mathbb{P}(Q) \times \mathbb{P}(Q)$ is an *acceptance condition*. $\mathcal{S}$ is *complete*, if for all $q \in Q$ and $a \in \Sigma$, there is some $q' \in Q$ such that $(q, a, q') \in \Delta$. It is *deterministic* if, for all $q \in Q$ and $a \in \Sigma$, there is at most one $q' \in Q$ such that $(q, a, q') \in \Delta$.

A finite alternating sequence $q_0 a_1 q_1 \ldots a_n q_n$ of states and input symbols of $\mathcal{S}$ is a *(finite) path in $\mathcal{S}$ (from q_0) (to q_n)* if $(q_{i-1}, a_i, q_i) \in \Delta$ for all $i \in \{1, \ldots, n\}$. An *infinite path* is defined analogously. A state q is *reachable (in $\mathcal{S}$)* if there is a path in $\mathcal{S}$ from the initial state $\hat{q}$ to q. If ρ is an (infinite) path in $\mathcal{S}$, then $\inf(\rho)$ denotes the set of all states that occur infinitely often in ρ. Path ρ is *accepting* if, for all $(Q_1, Q_2) \in F$, $\inf(\rho) \cap Q_1 = \emptyset$ or $\inf(\rho) \cap Q_2 \neq \emptyset$. An infinite path in $\mathcal{S}$ from $\hat{q}$ is called a *run (of $\mathcal{S}$)*. The automaton $\mathcal{S}$ *accepts* an infinite word $w \in \Sigma^\omega$ if there is an accepting run ρ of $\mathcal{S}$ such that w is obtained from ρ by removing all states. The set of all infinite words accepted by $\mathcal{S}$ is called the *language (of $\mathcal{S}$)* and denoted by $L(\mathcal{S})$.

Kripke Structures (KSs): Like an LTS, a KS [1] is a directed graph whose nodes and edges are called states and transitions, respectively, and one of the states is marked as the initial state. The difference is that instead of transitions, the emphasis is on states, each of which is assigned a set of elements called *atomic propositions*.

Formally, a KS is a five-tuple $\mathcal{K} := (S, P, l, R, \hat{s})$, where S is a non-empty set of *states*, P is a set of atomic propositions, $l : S \mapsto \mathbb{P}(P)$ is a *labelling function*, $R \subseteq S \times S$ is a set of *transitions*, and $\hat{s} \in S$ is the initial state. $\mathcal{K}$ is finite if it has a finite number of states and atomic propositions. An infinite sequence of states $s_0 s_1 s_2 \ldots$ of $\mathcal{K}$ is an *(infinite) path (in $\mathcal{K}$) (from s_0)* if $(s_i, s_{i+1}) \in R$ for all $i \in \mathbb{N}$. A path from the initial state $\hat{s}$ is called an *execution* of $\mathcal{K}$.

Modal Logic: When system implementations are modelled as KSs, specifications are often given in temporal logic [1]. Here, we use the simplest modal logic where the only temporal operators are *always* and *eventually*, but which is strong enough to express fairness.

The formulae ϕ of a *modal logic* are given by the grammar $\phi ::= p \mid (\neg\phi) \mid (\phi \vee \phi) \mid (\Box\phi)$, where p stands for an atomic proposition. Let $\mathcal{K}$ be a KS with the labelling function l and $\sigma = s_0 s_1 s_2 \ldots$ a path in $\mathcal{K}$. We define the satisfaction relation $\models$ between $\mathcal{K}, \sigma$ and a modal formula inductively on the structure of formulae as follows:

1) $\mathcal{K}, \sigma \models p$ if $p \in l(s_0)$.
2) $\mathcal{K}, \sigma \models \neg\phi$ if $\mathcal{K}, \sigma \not\models \phi$, i.e., not $\mathcal{K}, \sigma \models \phi$.
3) $\mathcal{K}, \sigma \models \phi_1 \vee \phi_2$ if $\mathcal{K}, \sigma \models \phi_1$ or $\mathcal{K}, \sigma \models \phi_2$.
4) $\mathcal{K}, \sigma \models \Box\phi$ if $\mathcal{K}, s_i s_{i+1} s_{i+2} \ldots \models \phi$ for all $i \in \mathbb{N}$.

We say that $\mathcal{K}$ *satisfies* ϕ, denoted by $\mathcal{K} \models \phi$, if $\mathcal{K}, \sigma \models \phi$ for every execution σ of $\mathcal{K}$.

We use the standard abbreviations $(\phi_1 \rightarrow \phi_2)$, $(\phi_1 \wedge \phi_2)$ and $\Diamond\phi$ for $((\neg\phi_1) \vee \phi_2)$, $\neg((\neg\phi_1) \vee (\neg\phi_2))$ and $\neg\Box\neg\phi$, respectively. Since conjunction $\wedge$ and disjunction $\vee$ are associative and commutative, we use their generalised versions when conjunction or disjunction is taken over a finite set.

Whenever P_1 and P_2 are finite sets of atomic propositions, we write $\phi_{\mathcal{F}}(P_1; P_2)$ for the modal formula

$(\Box\Diamond \bigvee_{p_1 \in P_1} p_1) \rightarrow (\Box\Diamond \bigvee_{p_2 \in P_2} p_2)$ which expresses the fairness property that, whenever some proposition in P_1 is true infinitely often, then some proposition in P_2 must be true infinitely often, too. We say that a path σ in $\mathcal{K}$ is (P_1, P_2)-*fair* if $\mathcal{K}, \sigma \models \phi_{\mathcal{F}}(P_1; P_2)$.

III. LTSs WITH UCs

For the automatic verification of complex systems, it is often advantageous to be able to capture the behaviour of a system implementation and specification in small finite models. By using finite LTSs with congruence preserving semantics, one can exploit compositional construction and state-space reduction techniques to achieve the goal. Unfortunately, sometimes it is impossible to avoid modelling unrealistic unfair behaviours.

As an example, consider a shared resource system (SRS) where two users compete for access to a shared resource. To access the resource, a user has to obtain a lock that can be held by at most one user at a time. We wish to formally prove that, if the locking policy is fair, i.e., the lock requests of one user are not favoured all the time, then both users can access the resource infinitely often.

Because a user is allowed to access the resource arbitrarily but not infinitely many times consecutively, we observe that any attempt to formalize the specification leads essentially to the infinite-state LTS (or equivalent) of Fig. 1. The use of the resource by user i is denoted by two actions, ub_i and ue_i, which refer to the beginning and end of the operation, respectively. When user 1 (or 2) accesses the resource, the specification LTS proceeds upwards (or to the left). Because the LTS supports only finitely many consecutive steps upwards or to the left from any state, it does not allow for a single user to use the resource forever. However, as one proceeds upwards or to the left, one can simultaneously and silently move to any column or row (except for the first one), respectively, by picking a suitable τ-transition from an a_{yz}-labelled state. This means that, after one user has accessed the resource at least once, the other user is allowed to access it again. Therefore, the LTS reflects that both users are able to access the resource infinitely often.

Even though the LTS is infinite, it has a regular structure that can be exploited to express the LTS finitely; the states are of the form x_{yz}, where x can be viewed as one of the five control states a, b, c, d, e, and y and z as the values of counters c_1 and c_2, respectively. When user 1 is about to access the resource, i.e., the LTS proceeds upwards, counter c_1 is decremented. At the same time, counter c_2 can be initialised to any positive value since the LTS can simultaneously move to any column except for the first one. Similarly, when user 2 is about to access the resource, c_2 is decremented and c_1 is initialised to a positive value. Hence, via unidirectional counters (UCs) that support initialising to an arbitrary positive value (denoted **init**) and decrementing

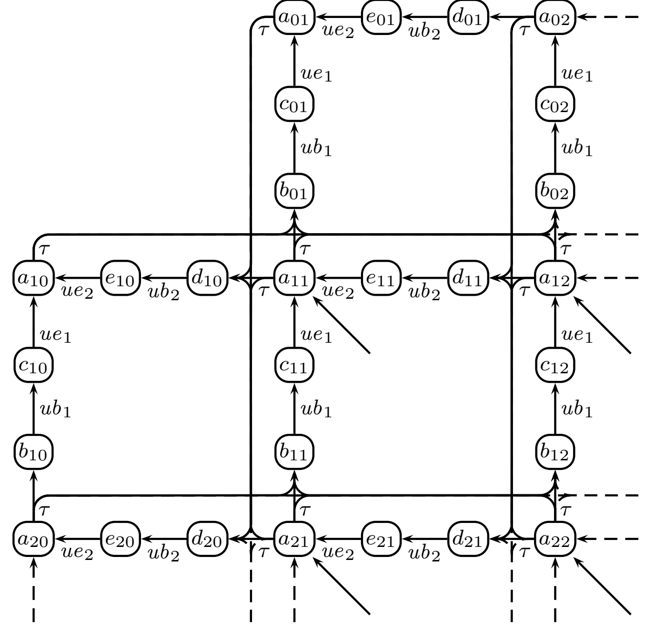

Figure 1. Infinite-state specification LTS.

a positive value (denoted **dec**), we can express the specification LTS as the finite-state system of Fig. 2.

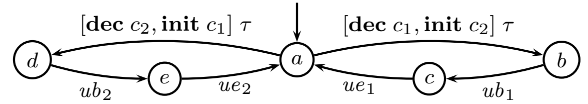

Figure 2. The formal specification *Spec* of SRS represented as an LTS with two UCs.

To formalize the concept of an LTS with UCs, we write $\mathbb{O}$ for the set $\{\mathbf{init}, \mathbf{dec}, \mathbf{nop}\}$ of counter operations, where **nop** is an operation that preserves the value of a counter.

Definition 1 (UCLTS)**.** An *LTS with UCs (UCLTS)* is a five-tuple $(S, A, C, R, \hat{s})$, where S is a finite set of *states*, A is a finite set of actions (an *alphabet*), C is a finite set of *counters*, $R \subseteq S \times \prod_{c \in C} \mathbb{O} \times (A \cup \{\tau\}) \times S$ is a set of *transitions* ($\prod_{c \in C} \mathbb{O}$ denotes the set of all functions $\delta : C \mapsto \mathbb{O}$), and $\hat{s} \in S$ is the *initial state*.

A UCLTS is finite by definition but it intuitively corresponds to an infinite LTS. Each state of the infinite LTS consists of a state s of a UCLTS and a function f that maps counters to their current values. Initially, the counters may take any positive value and, if the LTS is in a state (s, f), it can execute an action a if there is a transition (s, δ, a, s') of the UCLTS such that f maps every counter that the transition decrements to a positive value. After executing a, the LTS enters the state (s', f'), where $f'(c) > 0$ if the transition initialises c (i.e., $\delta(c) = \mathbf{init}$), $f'(c) = f(c) - 1$ if the transition decrements c (i.e., $\delta(c) = \mathbf{dec}$) and $f'(c) = f(c)$ if the transition preserves c (i.e., $\delta(c) = \mathbf{nop}$). Such an LTS

is called the *interpretation* of a UCLTS:

Definition 2 (Interpretation)**.** Let $\mathcal{C}$ be a UCLTS $(S, A, C, R, \hat{s})$. The *interpretation (of $\mathcal{C}$)*, denoted by $[\![\mathcal{C}]\!]$, is a four-tuple (S', A, R', I'), where

- $S' = S \times \prod_{c\in C} \mathbb{N}$,
- R' is the set of all triples $((s, f), a, (s', f')) \in S' \times (A \cup \{\tau\}) \times S'$ for which there is $\delta : C \mapsto \mathbb{O}$ such that $(s, \delta, a, s') \in R$ and, for all $c \in C$,
  - if $\delta(c) = \mathbf{nop}$, then $f(c) = f'(c)$,
  - if $\delta(c) = \mathbf{init}$, then $f(c) \in \mathbb{N}$ and $f'(c) \geq 1$,
  - if $\delta(c) = \mathbf{dec}$, then $f(c) > 0$ and $f'(c) = f(c)-1$,
- $I' = \{\hat{s}\} \times \prod_{c\in C} \mathbb{Z}_+$.

It is easy to see that the interpretation of a UCLTS is an LTS and that the interpretation of the UCLTS of Fig. 2 is the infinite LTS of Fig. 1. We can now give semantics to UCLTSs via their interpretations:

Definition 3 (Preorder on UCLTSs)**.** Let $\preceq$ be a preorder on LTSs. Then, $\widehat{\preceq}$ is a preorder on UCLTSs defined by $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1 \mathrel{\widehat{\preceq}} \mathcal{C}_2$ if and only if $[\![\mathcal{C}_1]\!] \preceq [\![\mathcal{C}_2]\!]$.

To model the SRS implementation we formalize the behaviour of user i as the UCLTS $User_i$ of Fig. 3. It states that the user can repeatedly request the lock (denoted by lck_i), use the resource and release the lock (denoted by unl_i). In any state, the user can also perform other activities, denoted by a τ-action. To make sure that the user does not perform other activities forever, the model uses a UC $c_{i,\tau}$ that is decremented whenever a τ-transition is taken and initialised whenever a visible action is executed.

The behaviour of the lock is captured in the UCLTS $Lock$ of Fig. 3. It formally states that at most one user can hold the lock at any time and that the lock is not continuously granted to the same user. The latter property is achieved by using two UCs in the same fashion as in the formal specification.

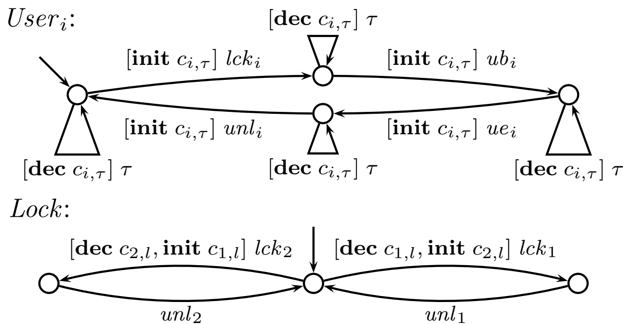

Figure 3. UCLTSs $User_i$ and $Lock$, representing the behaviour of user i ($i = 1, 2$) and the lock of SRS, respectively.

To construct the implementation model and to verify it against the specification, we should put the UCLTSs $User_1$, $User_2$ and $Lock$ in parallel and hide the locking actions. This motivates the lifting of the parallel composition and hiding operators on LTSs to UCLTSs:

Definition 4 (Parallel composition on UCLTSs)**.** Let $\mathcal{C}_i$ be the UCLTS $(S_i, A_i, C_i, R_i, \hat{s}_i)$, for $i = 1, 2$, such that C_1 and C_2 are disjoint. The *parallel composition (of $\mathcal{C}_1$ and $\mathcal{C}_2$)*, denoted by $\mathcal{C}_1 \mathbin{\widehat{\|}} \mathcal{C}_2$, is the five-tuple $(S_1 \times S_2, A_1 \cup A_2, C_1 \cup C_2, R_{\widehat{\|}}, (\hat{s}_1, \hat{s}_2))$, where $R_{\widehat{\|}}$ is the set of all four-tuples $((s_1, s_2), \delta, a, (s'_1, s'_2))$ such that $\delta : C_1 \cup C_2 \mapsto \mathbb{O}$ and

- $(s_i, \delta|_{C_i}, a, s'_i) \in R_i$ for both $i = 1, 2$ and $a \neq \tau$, or
- for distinct elements $i, j \in \{1, 2\}$, $(s_i, \delta|_{C_i}, a, s'_i) \in R_i$, $a \notin A_j$, $s_j \in S_j$, $s_j = s'_j$, and $\delta|_{C_j}$ is the constant function $\mathbf{nop}$ ($\delta|_{C_k}$ denotes the restriction of δ to C_k).

Hence, the parallel composition of UCLTSs treats actions like its LTS counterpart and preserves counter operations.

Definition 5 (Hiding on UCLTSs)**.** Let $\mathcal{C}$ be the UCLTS $(S, A, C, R, \hat{s})$ and B a set of visible actions. The UCLTS *$\mathcal{C}$ after hiding B*, denoted by $\mathcal{C} \mathbin{\widehat{\setminus}} B$, is the five-tuple $(S, A \setminus B, C, R', \hat{s})$, where R' is the set of all four-tuples (s, δ, a, s') such that $(s, \delta, a, s') \in R$ and $a \notin B$; or $a = \tau$ and there is some $b \in B$ for which $(s, \delta, b, s') \in R$.

Hence, hiding works exactly as in the case of LTSs; it affects only actions and does not change counter operations.

Obviously, the parallel composition of two UCLTSs as well as a UCLTS after hiding a set of visible actions are UCLTSs. Thus, $Sys := (User_1 \mathbin{\widehat{\|}} User_2) \mathbin{\widehat{\|}} Lock$ is a UCLTS; it is depicted in Fig. 4 (without isolated states). Moreover, when we hide the set $LA := \{lck_1, unl_1, lck_2, unl_2\}$ of locking actions we get the UCLTS $Sys \mathbin{\widehat{\setminus}} LA$ that is obtained from Sys by substituting τ for every occurrence of an action in LA. Hence, the question on the correctness of SRS can be formally expressed as the refinement checking task $Sys \mathbin{\widehat{\setminus}} LA \mathrel{\widehat{\preceq}_{\mathrm{CFFD}}} Spec$.

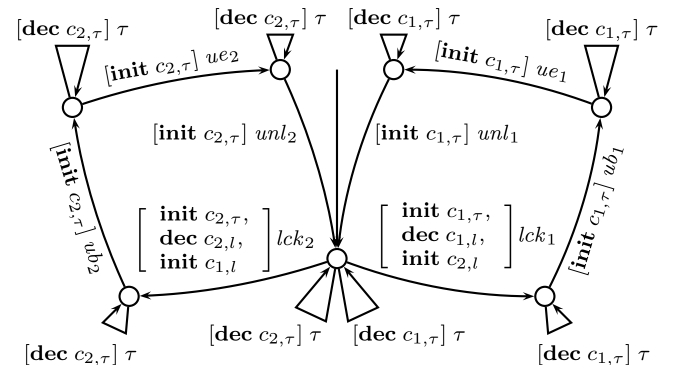

Figure 4. UCLTS $Sys = (User_1 \mathbin{\widehat{\|}} User_2) \mathbin{\widehat{\|}} Lock$.

The interpretation of $Sys \mathbin{\widehat{\setminus}} LA$ is practically the same LTS as the one obtained by putting the LTS interpretations of the UCLTSs $User_1$, $User_2$ and $Lock$ in parallel and hiding the locking actions. This is because the interpretation of the parallel composition of UCLTSs matches the parallel composition of the interpretations of the UCLTSs modulo the

naming of states, and taking the interpretation of a UCLTS after hiding a set of actions yields the same result as taking the interpretation of the UCLTS first and hiding the actions thereafter. Formally, let $\mathcal{L}_i$ be the LTS (S_i, A_i, R_i, I_i) for $i = 1, 2$. The LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$ are *isomorphic*, denoted by $\mathcal{L}_1 \simeq \mathcal{L}_2$, if $A_1 = A_2$ and there is a bijection $g : S_1 \mapsto S_2$ such that $R_2 = \{(g(s), a, g(s')) \mid (s, a, s') \in R_1\}$ and $I_2 = \{g(s) \mid s \in I_1\}$.

Lemma 6. *Let $\mathcal{C}_1$, $\mathcal{C}_2$ be UCLTSs and B a visible action set. Then, $[\![\mathcal{C}_1 \mathbin{\widehat{\|}} \mathcal{C}_2]\!] \simeq [\![\mathcal{C}_1]\!] \parallel [\![\mathcal{C}_2]\!]$ and $[\![\mathcal{C}_1 \mathbin{\widehat{\setminus}} B]\!] = [\![\mathcal{C}_1]\!] \setminus B$.*

This lemma leads to our first result, namely that the compositionality of preorders on LTSs is preserved when operating directly on UCLTSs:

Theorem 7. *Let $\preceq$ be a compositional preorder on LTSs coarser than $\simeq$, i.e., $\simeq \subseteq \preceq$. Then, $\widehat{\preceq}$ is a compositional preorder on UCLTSs.*

As all preorders used in practice are coarser than isomorphism, every compositional preorder on LTSs is a compositional preorder on UCLTSs, too. Especially, CFFD refinement $\widehat{\preceq}_{\mathrm{CFFD}}$ is a compositional preorder on UCLTSs.

IV. Refinement Checking UCLTSs

Due to their infiniteness, the LTS interpretations of UCLTSs are not suitable for automatic verification. Thus, we develop finite versions that can be used for the algorithmic checking of the finite and infinite part of CFFD refinement, respectively. Our approach consists of two parts.

Finite Behaviours: First, we abstract the value of each counter into a single bit, which tells us whether the counter has a positive value. The resulting structure, called the *finite interpretation* of a UCLTS, is defined like the interpretation before, except for counter initialisation and decrementing. The former operation sets the value of a counter to one and the latter chooses it non-deterministically to be zero or one.

Definition 8 (Finite interpretation)**.** Let $\mathcal{C}$ be the UCLTS $(S, A, C, R, \hat{s})$. The *finite interpretation (of $\mathcal{C}$)*, denoted by $[\![\mathcal{C}]\!]_{\mathit{fin}}$, is the four-tuple $(S', A, R', \{(\hat{s}, \hat{f}_C)\})$, where

- $S' = S \times \prod_{c \in C} \{0, 1\}$,
- R' is the set of all triples $((s, f), a, (s', f')) \in S' \times (A \cup \{\tau\}) \times S'$ for which there is $\delta : C \mapsto \mathbb{O}$ such that $(s, \delta, a, s') \in R$ and, for all $c \in C$,
  - if $\delta(c) = \mathbf{nop}$, then $f(c) = f'(c)$,
  - if $\delta(c) = \mathbf{init}$, then $f(c) \in \{0, 1\}$ and $f'(c) = 1$,
  - if $\delta(c) = \mathbf{dec}$, then $f(c) = 1$ and $f'(c) \in \{0, 1\}$,
- $\hat{f}_C$ is the constant function 1.

It is easy to see that the finite interpretation of a UCLTS is a finite LTS whose size is linear in the size of the UCLTS and exponential in the number of counters. Especially, $[\![\mathit{Spec}]\!]_{\mathit{fin}}$ is the LTS of Fig. 5. The state labels are of the form x_{yz}, where x refers to the state of *Spec* and y and z are the abstracted values of the counters c_1 and c_2, respectively. Moreover, one can see that $[\![\mathit{Spec}]\!]_{\mathit{fin}}$ is obtained from $[\![\mathit{Spec}]\!]$ by applying the signum function, sgn, to counter values. This generally holds for UCLTSs.

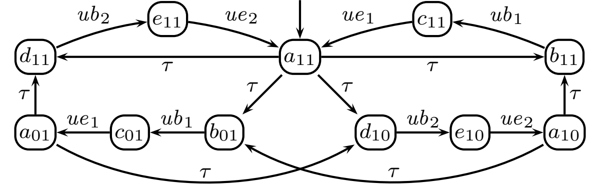

Figure 5. The finite interpretation of the formal specification *Spec*.

Lemma 9. *Let $\mathcal{C}$ be the UCLTS $(S, A, C, R, \hat{s})$. Then, the finite interpretation $[\![\mathcal{C}]\!]_{\mathit{fin}}$ is a finite LTS. Moreover,*

1) *if $(s_0, f_0)a_1(s_1, f_1) \ldots a_n(s_n, f_n)$ is an execution of $[\![\mathcal{C}]\!]$, then $(s_0, \mathrm{sgn} \circ f_0)a_1(s_1, \mathrm{sgn} \circ f_1) \ldots a_n(s_n, \mathrm{sgn} \circ f_n)$ is an execution of $[\![\mathcal{C}]\!]_{\mathit{fin}}$, and*
2) *if $(s_0, f_0)a_1(s_1, f_1) \ldots a_n(s_n, f_n)$ is an execution of $[\![\mathcal{C}]\!]_{\mathit{fin}}$, then there are functions $f'_0, \ldots, f'_n : C \mapsto \mathbb{N}$ such that $(s_0, f'_0)a_1(s_1, f'_1) \ldots a_n(s_n, f'_n)$ is an execution of $[\![\mathcal{C}]\!]$ and $f_i = \mathrm{sgn} \circ f'_i$ for all $i \in \{0, \ldots, n\}$.*

Because the signum abstraction of counter values preserves the enabledness and disabledness of transitions, it does not affect finite behaviours. Therefore, the finite interpretation of a UCLTS is failures equivalent to the interpretation of the UCLTS, and one can check the finite traces and failures of UCLTSs using their finite interpretations.

Theorem 10. *For every UCLTS $\mathcal{C}$, $[\![\mathcal{C}]\!]_{\mathit{fin}} =_{\mathrm{F}} [\![\mathcal{C}]\!]$.*

For our example, we encoded the finite interpretations of $\mathit{Sys} \mathbin{\widehat{\setminus}} \mathit{LA}$ and *Spec* in the CSP language and ran the FDR2 refinement checker [2] to establish failures refinement. Within a second, FDR2 gave a positive answer which means that $\mathit{Sys} \mathbin{\widehat{\setminus}} \mathit{LA} \mathrel{\widehat{\preceq}_{\mathrm{F}}} \mathit{Spec}$. In other words, only one of the users can access the resource at any time, and at least one user can access the resource infinitely often. However, we do not know whether the users are treated fairly because the abstraction may introduce infinite behaviours that are not present in the original system; if the finite interpretation decrements a counter arbitrary many times consecutively, it can do so infinitely often, too, which is not possible for the interpretation. This is why finite interpretations cannot be used for the analysis of infinite behaviours.

Infinite Behaviours: The other part of the solution is to convert a UCLTS $\mathcal{C}$ into an ω-automaton that accepts precisely the infinite executions that give rise to a divergence or an infinite trace of $[\![\mathcal{C}]\!]$. We use a Streett automaton (SA) for this purpose since its acceptance condition naturally allows us to concentrate on infinite executions that contain infinitely many initialisations or only finitely many decrements of each counter. Such an automaton is called the ω-interpretation of

a UCLTS and defined as follows:

Definition 11 (ω-interpretation)**.** Let $\mathcal{C}$ be the UCLTS $(S, A, C, R, \hat{s})$. The *ω-interpretation (of $\mathcal{C}$)*, denoted by $[\![\mathcal{C}]\!]_\omega$, is the five-tuple $(Q, A \cup \{\tau\}, \Delta, (\hat{s}, \hat{\delta}_C), F)$, where

- $Q = S \times \prod_{c \in C} \mathbb{O}$,
- Δ is the set of all triples $((s, \delta), a, (s', \delta'))$ such that $\delta : C \mapsto \mathbb{O}$ and $(s, \delta', a, s') \in R$,
- $\hat{\delta}_C$ is the constant function $\mathbf{nop}$,
- F is the set that consists of, for all $c \in C$, the pairs $(\{(s, \delta) \in Q \mid \delta(c) = \mathbf{dec}\}, \{(s, \delta) \in Q \mid \delta(c) = \mathbf{init}\})$.

It is obvious that the ω-interpretation of a UCLTS $\mathcal{C}$ is an SA whose size is linear in the size of the UCLTS and exponential in the number of counters. The SA of Fig. 6 is the ω-interpretation of the UCLTS *Spec*, where the states (s, δ) of $[\![Spec]\!]_\omega$ are represented in the form $s_{\delta(c_1),\delta(c_2)}$. By taking a closer look, one sees that the infinite sequences of actions that $[\![Spec]\!]$ can execute are precisely the words accepted by $[\![Spec]\!]_\omega$. This holds for all UCLTSs.

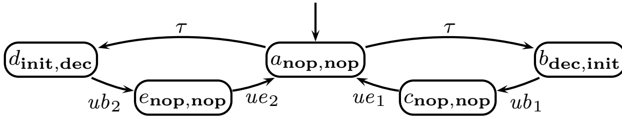

Figure 6. The reachable part of the ω-interpretation of *Spec*.

Lemma 12. *Let $\mathcal{C}$ be the UCLTS $(S, A, C, R, \hat{s})$. Then, the ω-interpretation $[\![\mathcal{C}]\!]_\omega$ is an SA. Moreover,*

1) *if $(s_0, f_0)a_1(s_1, f_1)a_2(s_2, f_2)\ldots$ is an infinite execution of $[\![\mathcal{C}]\!]$, then there are functions $\delta_0, \delta_1, \ldots : C \mapsto \mathbb{O}$ such that $(s_0, \delta_0)a_1(s_1, \delta_1)a_2(s_2, \delta_2)\ldots$ is an accepting run of $[\![\mathcal{C}]\!]_\omega$, and*
2) *if $(s_0, \delta_0)a_1(s_1, \delta_1)a_2(s_2, \delta_2)\ldots$ is an accepting run of $[\![\mathcal{C}]\!]_\omega$, then there are functions $f_0, f_1, \ldots : C \mapsto \mathbb{N}$ such that $(s_0, f_0)a_1(s_1, f_1)a_2(s_2, f_2)\ldots$ is an infinite execution of $[\![\mathcal{C}]\!]$.*

Because an SA treats all input symbols (actions) in a uniform way, also τ-symbols are visible in the language of the ω-interpretation of a UCLTS $\mathcal{C}$. This is why, by the lemma above, the set of infinite behaviours (divergences and infinite traces) of $[\![\mathcal{C}]\!]$ matches the language of $[\![\mathcal{C}]\!]_\omega$ only after erasing τ-symbols. Therefore, we cannot phrase the containment problem of the infinite behaviours of two UCLTSs directly as the language containment problem of their ω-interpretations. We need to modify the ω-interpretation of a UCLTS on the specification side in such a way that it can skip all τ-symbols between two visible actions and perform an arbitrary finite number of τ-symbols from any state.

To enable $[\![\mathcal{C}]\!]_\omega$ to perform an arbitrary number of τ-symbols from any state, we add a τ-loop to each state. However, to ensure that the automaton does not execute the loop infinitely often without taking another transition, we insert one bit of information to each state that encodes whether the previously taken transition was an added τ-transition. We also modify the acceptance condition such that, whenever the automaton executes an added τ-transition infinitely often, it executes another transition infinitely often, too. Thus, the states of the modified automaton become pairs (q, b), where q is a state of $[\![\mathcal{C}]\!]_\omega$ and $b \in \{0, 1\}$ indicates whether the previous transition was an added one.

To enable the automaton to skip τ-symbols between two visible actions, we add an a-labelled transition ($a \neq \tau$) from every reachable state q that is either the initial state or follows a visible action to a state q' that can be reached from q by taking an arbitrary number of τ-transitions first and then an a-labelled one. For each counter, we store only the greatest operation on the skipped path where operations are ordered by $\mathbf{nop} < \mathbf{dec} < \mathbf{init}$. This is sufficient since, when we consider a finite (but possibly infinitely often repeating) part of an infinite path from the viewpoint of acceptance, multiple occurrences of a counter operation on the finite path are insignificant and, if a counter is initialised on the finite path it does not matter whether it is decremented. After adding such shortcut transitions, the transitions on the skipped paths can be discarded. However, to preserve information on divergences we do the following: if $[\![\mathcal{C}]\!]_\omega$ has an accepting path of τ-transitions from q, we insert a τ-transition from q to a special divergent state from which τ (and only τ) can be executed infinitely often.

To present this construction formally, we introduce a counter function composition operator $\bullet$ and the progress states of a UCLTS. Whenever C is a set of counters and $\delta_1, \delta_2 : C \mapsto \mathbb{O}$ are counter functions, $\delta_1 \bullet \delta_2$ denotes a function $\delta' : C \mapsto \mathbb{O}$ such that $\delta'(c) = \max\{\delta_1(c), \delta_2(c)\}$ for all $c \in C$. The *progress states* of a UCLTS $\mathcal{C}$ are the initial state and all states s for which there is a state s' and a visible action a such that (s', a, s) is a transition in $\mathcal{C}$. In our example, *Spec*'s progress states are the states a, c, e.

Definition 13 (τ-closed ω-interpretation)**.** Let $\mathcal{C}$ be a UCLTS with a set C of counters and $[\![\mathcal{C}]\!]_\omega = (Q, \Sigma, \Delta, \hat{q}, F)$ its ω-interpretation. The *τ-closed ω-interpretation (of $\mathcal{C}$)*, denoted by $\tau^*([\![\mathcal{C}]\!]_\omega)$, is the five-tuple $(Q', \Sigma, \Delta', (\hat{q}, 0), F')$ where

- $Q' = (Q \cup \{\top_\mathcal{C}\}) \times \{0, 1\}$ and $\top_\mathcal{C} \notin Q$ is a *divergent state* unique to $\mathcal{C}$,
- Δ' is the smallest subset of $Q' \times \Sigma \times Q'$ such that
  - S1) whenever s is $\top_\mathcal{C}$ or a progress state of $\mathcal{C}$, then $(((s, \delta), b), \tau, ((s, \delta), 1)) \in \Delta'$ for all $\delta : C \mapsto \mathbb{O}$ and both $b = 0, 1$,
  - S2) whenever s_0 is a progress state of $\mathcal{C}$ and $(s_0, \delta_0)\tau(s_1, \delta_1)\ldots\tau(s_{n-1}, \delta_{n-1})a(s_n, \delta_n)$ is a path in $[\![\mathcal{C}]\!]_\omega$ such that $n > 0$ and $a \neq \tau$, then $(((s_0, \delta_0), b), a, ((s_n, \delta_1 \bullet \ldots \bullet \delta_n), 0)) \in \Delta'$ for both $b = 0, 1$,
  - S3) whenever s_0 is a progress state of $\mathcal{C}$ and $(s_0, \delta_0)\tau(s_1, \delta_1)\tau(s_2, \delta_2)\ldots$ is an accepting path

in $[\![\mathcal{C}]\!]_\omega$, then $(((s_0,\delta_0),b),\tau,(\top_\mathcal{C},0)) \in \Delta'$ for both $b=0,1$ and $((\top_\mathcal{C},0),\tau,(\top_\mathcal{C},0)) \in \Delta'$,

- F' is the set of $((Q \cup \{\top_\mathcal{C}\}) \times \{1\},(Q \cup \{\top_\mathcal{C}\}) \times \{0\})$ and all pairs $(Q_1 \times \{0,1\}, Q_2 \times \{0,1\})$ such that $(Q_1,Q_2) \in F$.

Obviously, the τ-closed ω-interpretation of a UCLTS is an SA whose size is linear in the size of the ω-interpretation. In our example, $\tau^*([\![\mathit{Spec}]\!]_\omega)$ is the SA of Fig. 7. The τ-transitions are obtained from the progress states of $[\![\mathit{Spec}]\!]_\omega$ by using S1 and the other transitions from paths of $[\![\mathit{Spec}]\!]_\omega$ by applying S2. In this case, S3 is unused because $[\![\mathit{Spec}]\!]_\omega$ does not contain cycles of τ-transitions.

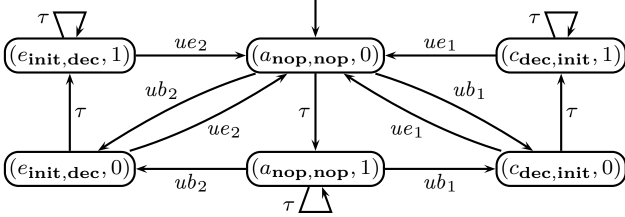

Figure 7. The reachable part of the τ-closed ω-interpretation of *Spec*.

To construct the τ-closed ω-interpretation algorithmically we determine the progress states first. This can be done, e.g., via a depth-first search on the state-space of $\mathcal{C}$.

S1: For each progress state s, we first insert $(((s,\delta),b),\tau,((s,\delta),1))$ into Δ' for all $\delta : C \mapsto \mathbb{O}$ and both $b=0,1$.

S2: Next, we run a depth-first search on the state-space of $[\![\mathcal{C}]\!]_\omega$ from $(s,\hat{\delta}_C)$ to determine the shortcut transitions. (Recall that $\hat{\delta}_C$ denotes a function that maps all counters of $\mathcal{C}$ to $\mathbf{nop}$.) During the search, the states on the path from $(s,\hat{\delta}_C)$ to the current state are stored in a stack. Initially, the stack contains only $(s,\hat{\delta}_C)$. If there is an element (s',δ') on top of the stack and a transition $((s',\delta'),a,(s'',\delta''))$ that is not marked visited, we mark the transition visited and take the following actions. If $a=\tau$, then $(s'',\delta' \bullet \delta'')$ is pushed on the stack. Otherwise, $a \neq \tau$ and $(((s,\delta),b),a,((s'',\delta' \bullet \delta''),0))$ is inserted in Δ' for all $\delta : C \mapsto \mathbb{O}$ and both $b=0,1$. If every transition $((s',\delta'),a,(s'',\delta''))$ is marked visited, then (s',δ') is popped from the stack and the search backtracks.

S3: When the stack becomes empty we check whether there is an accepting path (from $(s,\hat{\delta}_C)$) within the visited τ-transitions. For this purpose, the transitions are converted into a KS by neglecting the transition labels (all of which are τ) and considering states (s''',δ''') to be labelled by δ''' treated as a set. The existence of an accepting path can now be determined by checking whether $\neg \bigwedge_{c\in C} \phi_\mathcal{F}(\{(c,\mathbf{dec})\};\{(c,\mathbf{init})\})$ is false. The formula states that some counter is decremented infinitely often but initialised only finitely many times. If the formula is not satisfied, then $((\top_\mathcal{C},0),\tau,(\top_\mathcal{C},0))$ and $(((s,\delta),b),\tau,(\top_\mathcal{C},0))$ are inserted in Δ' for all $\delta : C \mapsto \mathbb{O}$ and both $b=0,1$.

Finally, the visited marks of the transitions are cleared and the depth-first search is run for the next progress state, until all of them have been considered.

Lemma 14. *Let $\mathcal{C}$ be a UCLTS. Then, the τ-closed ω-interpretation $\tau^*([\![\mathcal{C}]\!]_\omega)$ is an SA and satisfies the following:*

1) *If $(q_0,b_0)a_1(q_1,b_1)a_2(q_2,b_2)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$, then $(q_0,0)a_{i_1}(q_{i_1},0)a_{i_2}(q_{i_2},0)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$, where $i_1,i_2,\ldots$ is the increasing sequence of all $i \in \mathbb{Z}_+$ such that $b_i=0$.*
2) *If $(q_0,0)a_1(q_1,0)a_2(q_2,0)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$, then for every $i \in \mathbb{N}$, there are $n_i \in \mathbb{Z}_+$ and states $q_{i,1},\ldots,q_{i,n_i}$ of $[\![\mathcal{C}]\!]_\omega$ such that $\hat{q}\tau q_{0,1}\ldots\tau q_{0,n_0-1}a_1q_{0,n_0}\tau q_{1,1}\ldots\tau q_{1,n_1-1}a_2q_{1,n_1}\tau$ $q_{2,1}\ldots\tau q_{2,n_2-1}\ldots$ is an accepting run of $[\![\mathcal{C}]\!]_\omega$.*
3) *If $q_0a_1q_1a_2q_2\ldots$ is an accepting run of $[\![\mathcal{C}]\!]_\omega$, then there are $q'_0,q'_1,\ldots$ such that q'_i is a state of $[\![\mathcal{C}]\!]_\omega$ or the divergent state $\top_\mathcal{C}$ for all $i \in \mathbb{N}$ and, moreover, $(q'_0,0)a_{i_1}(q'_1,0)a_{i_2}(q'_2,0)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$, where $i_1,i_2,\ldots$ is the increasing sequence of all $i \in \mathbb{N}$ such that $a_i \neq \tau$ or $a_ia_{i+1}\ldots = \tau\tau\ldots$.*
4) *If $(q_0,0)a_1(q_1,0)a_2(q_2,0)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$, then $(q_0,0)(\tau\ (q_0,1))^{n_0}a_1(q_1,0)(\tau\ (q_1,1))^{n_1}a_2(q_2,0)\ldots$ is an accepting run of $\tau^*([\![\mathcal{C}]\!]_\omega)$ for all $n_0,n_1,\ldots \in \mathbb{N}$.*

Intuitively, the lemma states that an execution of $[\![\mathcal{C}]\!]_\omega$ can be recovered from an execution of $\tau^*([\![\mathcal{C}]\!]_\omega)$ by removing the added τ-transitions (Item 1) and restoring the skipped τ-transitions (Item 2). Further, an execution of $\tau^*([\![\mathcal{C}]\!]_\omega)$ can be obtained from an execution of $[\![\mathcal{C}]\!]_\omega$ by removing all τ-symbols between two visible actions (Item 3) and inserting any number of τ-transitions in any location (Item 4). Together with Lemma 12, this means that the containment problem of the infinite behaviours of UCLTSs $\mathcal{C}_{impl}$ and $\mathcal{C}_{spec}$ can be formulated as the language containment problem between $[\![\mathcal{C}_{impl}]\!]_\omega$ and $\tau^*([\![\mathcal{C}_{spec}]\!]_\omega)$.

Theorem 15. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be UCLTSs. Then,* $\mathrm{dv}(\mathcal{C}_1) \subseteq \mathrm{dv}(\mathcal{C}_2)$ *and* $\mathrm{it}(\mathcal{C}_1) \subseteq \mathrm{it}(\mathcal{C}_2)$ *iff* $L([\![\mathcal{C}_1]\!]_\omega) \subseteq L(\tau^*([\![\mathcal{C}_2]\!]_\omega))$.

By Theorem 10, this implies that we can prove the implementation UCLTS $\mathcal{C}_{impl}$ correct with respect to the specification UCLTS $\mathcal{C}_{spec}$ under CFFD semantics, by checking failures refinement between $[\![\mathcal{C}_{impl}]\!]_{fin}$ and $[\![\mathcal{C}_{spec}]\!]_{fin}$ and language containment between $[\![\mathcal{C}_{impl}]\!]_\omega$ and $\tau^*([\![\mathcal{C}_{spec}]\!]_\omega)$.

Corollary 16. *Let $\mathcal{C}_1$, $\mathcal{C}_2$ be UCLTSs. Then, $\mathcal{C}_1 \,\widehat{\preceq}_{\mathrm{CFFD}}\, \mathcal{C}_2$ iff $[\![\mathcal{C}_1]\!]_{fin} \preceq_\mathrm{F} [\![\mathcal{C}_2]\!]_{fin}$ and $L([\![\mathcal{C}_1]\!]_\omega) \subseteq L(\tau^*([\![\mathcal{C}_2]\!]_\omega))$.*

The worst-case complexity of checking failures refinement between $[\![\mathcal{C}_{impl}]\!]_{fin}$ and $[\![\mathcal{C}_{spec}]\!]_{fin}$ is exponential in the size of $[\![\mathcal{C}_{spec}]\!]_{fin}$ [2]. Constructing $\tau^*([\![\mathcal{C}_{spec}]\!]_\omega)$ involves running a depth-first search and performing model checking for each progress state. As the complexity of model checking is exponential in the length of the property formula [1], the

complexity of this phase is polynomial in the size of $\mathcal{C}_{spec}$ and exponential in the number of counters in $\mathcal{C}_{spec}$. The language containment between $[\![\mathcal{C}_{impl}]\!]_\omega$ and $\tau^*([\![\mathcal{C}_{spec}]\!]_\omega)$ is solved by checking whether the intersection of the languages of $[\![\mathcal{C}_{impl}]\!]_\omega$ and the complement of $\tau^*([\![\mathcal{C}_{spec}]\!]_\omega)$ is empty. Since the complementation of an SA is of exponential complexity also in the best case [10], the worst-case complexity of checking CFFD refinement between $\mathcal{C}_{impl}$ and $\mathcal{C}_{spec}$ is linear in the size of $\mathcal{C}_{impl}$, exponential in the number of counters in $\mathcal{C}_{impl}$ and in the size of $\mathcal{C}_{spec}$, and doubly exponential in the number of counters in $\mathcal{C}_{spec}$.

When considering the complexity of the algorithm, it is important to note that specification processes are typically small. Moreover, CFFD refinement is compositional so that large systems can be checked correct by first checking the correctness of their smaller subsystems. This is the key advantage of (compositional) refinement-based model checking over (monolithic) temporal-logic model checking.

Deterministic Specifications: Even though Corollary 16 enables the algorithmic refinement checking of UCLTSs, we are unable to complete our SRS case study in practice. This is because tool support for Streett complementation is non-existent. To the best of our knowledge, the only tool for the complementation of ω-automata is GOAL [17], which currently supports Büchi automata but not SAs.

Fortunately, if the automaton on the specification side is deterministic, as is typically the case, the language containment problem for SAs can be solved without complementation. Clarke et al. have shown how the language containment problem can be formulated in temporal logic and hence solved using existing model checking algorithms [11]. Their approach involves constructing the product automaton of the implementation and specification automata, and representing it as a KS. The property being checked states that the satisfaction of the acceptance condition of the implementation implies the satisfaction of the acceptance condition of the specification. The authors assume that the automata are complete and have disjoint sets of states, which is not necessarily true in our case. However, the latter problem is easy to solve by renaming states where necessary, and the former problem can be overcome by adding transitions to a special sink state that cannot be part of any accepting run.

Definition 17 (Completion)**.** Let $\mathcal{S} = (Q, \Sigma, \Delta, \hat{q}, F)$ be an SA. We write $\overline{\mathcal{S}}$ for the five-tuple $(Q_\perp, \Sigma, \Delta \cup \Delta_\perp, \hat{q}, F \cup \{(\{\perp_\mathcal{S}\}, Q)\})$, where

- $Q_\perp = Q \cup \{\perp_\mathcal{S}\}$ and $\perp_\mathcal{S} \notin Q$ is a unique *sink state*;
- $\Delta_\perp$ is the set of all triples $(q, a, \perp_\mathcal{S})$, where $q \in Q_\perp$ and $a \in \Sigma$ such that $(q, a, q') \notin \Delta$ for all $q' \in \Delta$.

Obviously, $\overline{\mathcal{S}}$ is a complete SA that is deterministic if and only if $\mathcal{S}$ is, and that accepts the same language as $\mathcal{S}$.

Definition 18 (Kripke product)**.** Let $\mathcal{S}_i = (Q_i, \Sigma, \Delta_i, \hat{q}_i, F_i)$ be a complete SA, for $i = 1, 2$, such that Q_1 and Q_2 are disjoint. (Note that $\mathcal{S}_1$ and $\mathcal{S}_2$ have the same alphabet.) The *Kripke product (of $\mathcal{S}_1$ and $\mathcal{S}_2$)*, denoted by $\mathcal{S}_1 \times_\mathcal{K} \mathcal{S}_2$, is the five-tuple $(Q_1 \times Q_2, Q_1 \cup Q_2, l, R, (\hat{q}_1, \hat{q}_2))$, where

- l is the function $Q_1 \times Q_2 \mapsto Q_1 \cup Q_2$ such that $l((q_1, q_2)) = \{q_1, q_2\}$ for all $q_1 \in Q_1, q_2 \in Q_2$, and
- R is the set of all pairs $((q_1, q_2), (q_1', q_2'))$ such that $(q_1, a, q_1') \in \Delta_1$ and $(q_2, a, q_2') \in \Delta_2$ for some $a \in \Sigma$.

Obviously, the Kripke product is a KS. The result of Clarke et al. can now be expressed as follows:

Theorem 19 ([11])**.** *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be complete SAs with acceptance conditions F_1 and F_2, respectively, disjoint sets of states and the same alphabet. If $\mathcal{S}_2$ is deterministic, then $L(\mathcal{S}_1) \subseteq L(\mathcal{S}_2)$ if and only if $\mathcal{S}_1 \times_\mathcal{K} \mathcal{S}_2$ satisfies*

$$\Big(\bigwedge_{(Q_1, Q_1') \in F_1} \phi_\mathcal{F}(Q_1; Q_1') \Big) \to \Big(\bigwedge_{(Q_2, Q_2') \in F_2} \phi_\mathcal{F}(Q_2; Q_2') \Big) .$$

Combining this theorem with Corollary 16 provides a way to refinement-check UCLTSs with the aid of existing tools. Note that the worst case complexity is still dominated by the failures refinement check, but the best case complexity is improved since Streett complementation is avoided.

Because the τ-closed ω-interpretation treats τ as a normal input symbol, $\overline{\tau^*([\![Spec]\!]_\omega)}$ in Fig. 7 is deterministic and the approach can be applied to our SRS construction. We encoded the Kripke product of $\overline{[\![Sys \mathbin{\widehat{\setminus}} LA]\!]_\omega}$ and $\overline{\tau^*([\![Spec]\!]_\omega)}$ in the input language of the NuSMV model checker [12] and ran the tool to test for language containment. The tool gave a positive answer which means that $L([\![Sys \mathbin{\widehat{\setminus}} LA]\!]_\omega) \subseteq L(\tau^*([\![Spec]\!]_\omega))$. Together with the failures refinement of the finite interpretations of $Sys \mathbin{\widehat{\setminus}} LA$ and $Spec$, this means that the system works as expected, i.e., both users are able to access the resource infinitely often. Moreover, if SRS is part of a larger system, then $Spec$ can be used in place of $Sys \mathbin{\widehat{\setminus}} LA$ in further analysis.

In our SRS example, NuSMV needed about a minute to complete. This is quite a lot when compared with the time the refinement checking took. The reason lies in the complexity of model checking, which is exponential in the number of counters in the system implementation and specification UCLTSs. Fortunately, we can chop the formula into smaller pieces in two ways. Firstly, we can check each fairness property $\phi_\mathcal{F}(\cdot, \cdot)$ related to the specification process separately. Secondly, the fairness properties related to the implementation process can be considered as fairness constraints that limit the state space under inspection. Specifying fairness constraints explicitly is more efficient than encoding them in the property formula. Hence, for specifications with the deterministic τ-closed ω-interpretation, our result can be represented in the following form, which enables practical refinement and model checking:

Corollary 20. *Let $\mathcal{C}_1$, $\mathcal{C}_2$ be UCLTSs with disjoint states such that $\tau^*([\![\mathcal{C}_2]\!]_\omega)$ is deterministic. Further, let F_1 and*

F_2 be the acceptance conditions of $\overline{[\![\mathcal{C}_1]\!]_\omega}$ and $\overline{\tau^*([\![\mathcal{C}_2]\!]_\omega)}$, respectively. Then, $\mathcal{C}_1 \mathrel{\widehat{\preceq}_{\mathrm{CFFD}}} \mathcal{C}_2$ iff $[\![\mathcal{C}_1]\!]_{fin} \preceq_{\mathrm{F}} [\![\mathcal{C}_2]\!]_{fin}$ and

$$\overline{[\![\mathcal{C}_1]\!]_\omega} \times_{\mathcal{K}} \overline{\tau^*([\![\mathcal{C}_2]\!]_\omega)}, \sigma \models \phi_{\mathcal{F}}(Q_2, Q_2') \ ,$$

whenever $(Q_2, Q_2') \in F_2$ and σ is a (Q_1, Q_1')-fair execution of the Kripke product for all $(Q_1, Q_1') \in F_1$.

Using this corollary, NuSMV established the containment $L([\![Sys \mathbin{\widehat{\setminus}} LA]\!]_\omega) \subseteq L(\tau^*([\![Spec]\!]_\omega))$ within a second.

We were also able to consider a version of the alternating bit protocol with two users (two senders and receivers) and one-slot message and acknowledgement channels. In this case, the verification took a couple of minutes.

V. Conclusions and Future Work

We extended the algebra of (finite) labelled transition systems with unidirectional counters (UCs). This formalism of UCLTSs enables expressing fairness while preserving compositionality. Although similar constructs are implicitly used by other researchers [5], [14], to our knowledge we are the first ones who promote the explicit use of UCs for *compositional*, process-algebraic verification involving fairness. Most importantly, we showed that refinement on UCLTSs can be automatically checked under the compositional CFFD semantics, by utilising existing model checkers. This — a decision procedure with partial tool support — is another novelty that distinguishes our work from others [5], [13], [14], [15].

Although UCs can express many forms of fairness, a problem when equipping processes with them is that unwanted failures and deadlocks may arise. The reason is that transitions that decrement a UC become blocked when the value of the UC reaches zero. In the case of our SRS example, there is no such problem because the decrementing transitions start from an unstable state (after hiding). A more general solution is to avoid states that in some environment always lead to a situation where a UC reaches zero. This could be realised by marking such states *unimplementable* in the style of [14], although it is an open question as to how this can be done algorithmically.

Other future work includes implementing the method, making more experiments and considering state space reduction techniques for UCLTSs. Applying our results to the parameterized process-algebraic setting of [18] and I/O labelled transition systems [19] is within our interest, too.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[2] A. W. Roscoe, *Understanding Concurrent Systems*. Springer, 2010.

[3] R. P. Kurshan and K. L. McMillan, "A structural induction theorem for processes," *Inform. Comput.*, vol. 117, no. 1, pp. 1–11, 1995.

[4] A. Siirtola, "A parameterisation and a cut-off theorem for process algebras," http://www.tol.oulu.fi/users/antti.siirtola/papers/cutoff.pdf, 2012.

[5] A. Puhakka, "Using fairness constraints in process-algebraic verification," in *ICTAC '05*, ser. LNCS. Springer, 2005, vol. 3722, pp. 546–561.

[6] R. van Glabbeek, "The linear time – branching time spectrum II," in *CONCUR '93*, ser. LNCS. Springer, 1993, vol. 715, pp. 66–81.

[7] A. Valmari and M. Tienari, "An improved failures equivalence for finite-state systems with a reduction algorithm," in *PSTV '91*. North-Holland, 1991, pp. 3–18.

[8] B. Farwer, "ω-automata," in *Automata, Logics, and Infinite Games*, ser. LNCS. Springer, 2002, vol. 2500, pp. 265–274.

[9] O. Kupferman and M. Vardi, "Complementation constructions for nondeterministic automata on infinite words," in *TACAS '05*, ser. LNCS. Springer, 2005, vol. 3440, pp. 206–221.

[10] Y. Cai and T. Zhang, "A tight lower bound for Streett complementation," *CoRR*, vol. abs/1102.2963, 2011, http://arxiv.org/abs/1102.2963.

[11] E. Clarke, A. Browne, and R. Kurshan, "A unified approach for showing language containment and equivalence between various types of ω-automata," in *CAAP '90*, ser. LNCS. Springer, 1990, vol. 431, pp. 103–116.

[12] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An open source tool for symbolic model checking," in *CAV '02*, ser. LNCS. Springer, 2002, vol. 2404, pp. 241–268.

[13] M. Hennessy, "An algebraic theory of fair asynchronous communicating processes," in *ALP 12*, ser. LNCS. Springer, 1985, vol. 194, pp. 260–269.

[14] R. Cleaveland and G. Lüttgen, "A logical process calculus," in *EXPRESS '02*, ser. ENTCS, vol. 68, no. 2. Elsevier, 2002, pp. 33–50.

[15] A. Rensink and W. Vogler, "Fair testing," *Inform. Comput.*, vol. 205, no. 2, pp. 125–198, 2007.

[16] A. Siirtola, A. Puhakka, and G. Lüttgen, "The proofs of the paper," http://www.tol.oulu.fi/users/antti.siirtola/papers/ucltsapp.pdf.

[17] Y.-K. Tsay, Y.-F. Chen, M.-H. Tsai, W.-C. Chan, and C.-J. Luo, "GOAL extended: Towards a research tool for omega automata and temporal logic," in *TACAS '08*, ser. LNCS. Springer, 2008, vol. 4963, pp. 346–350.

[18] A. Siirtola, "Algorithmic multiparameterised verification of safety properties. Process algebraic approach," Ph.D. dissertation, University of Oulu, 2010.

[19] T. Chen, C. Chilton, B. Jonsson, and M. Kwiatkowska, "A compositional specification theory for component behaviours," in *ESOP '12*, ser. LNCS. Springer, 2012, vol. 7211, to appear.