Asynchronous Communication

Raffael Dzikowski · Andre Moelle · Wolfgang Reisig · Jan Sürmeli

Abstract Asynchronous communication, i.e., processes sending and receiving information along buffers, is a fundamental computation paradigm. Is is widely employed when large systems are to be designed, and it provides a basis for the fundamental software architecture of service-oriented computing.

This contribution suggests a generic model for this paradigm, in analogy to what process algebras provide for synchronous communication. We substantiate the genericity of the model by:

- a technically simple notion of composition
- the compatibility of composition and simulation
- a systematic view on classes of partners of a process that assert particular properties.

Contents

1 Introduction .......................................................... 1
2 Buffered Processes and Their Composition ....................... 4
3 Property-Asserting Partners ...................................... 14
4 Further partner classes and problems .............................. 17
5 Related Work ......................................................... 21
6 Appendix A (Transition Systems and Linear Time Properties) 22
7 Appendix B (Proofs) .................................................. 25

1 Introduction

According to the classical computing paradigm, an algorithm computes a partial function, with all input available initially and all output produced upon termination. This assumption contrasts modern algorithms that interact with their environment while computing. The shift from classical to interactive algorithms is fundamental. For example, an infinite, non-terminating run of a classical algorithm is flawed (as no output is produced), whereas interactive algorithms and
systems frequently are intended to perform non-terminating runs. Consequently, fundamental notions such as expressive power, equivalence etc., well established for classical algorithms, require a new definition for interactive algorithms. This observation has been constitutive for process algebras such as CCS, CSP, ACP and others.

Essentials of interactive systems can be demonstrated already by means of toy examples such as the beverage vending machine, $V$, of Figure 1.

![Fig. 1 The beverage vending machine, $V$](image)

Intuitively, $V$ models a system consisting of four states $A, \ldots, D$ and five steps, depicted as arrows. Each arrow is inscribed by an action. A question mark tags each action that requires an item from the system’s environment. An exclamation mark indicates delivery of an item to the system’s environment. This way, "coin?" indicates the receipt of a coin, "tea?" the receipt of an "I want tea" message, and "beverage!" indicates the system’s delivery of a beverage.

This kind of systems are usually denoted as processes. A process may be embedded in an environment, which is again a process.

![Fig. 2 Six different clients.](image)

Figure 2 presents a series of processes, modeling six different clients of the vending machine $V$ of Figure 1. Composing $V$ with one of the clients $C_i$ allows both of them to communicate with one another. This way, system interaction is
Asynchronous Communication

realized as process communication. Two variants of communication are usually distinguished:

**Synchronous** communication assumes composed processes to jointly execute actions. For example, a user’s insertion of a coin, \( c \), is coincident with the vending machine’s receipt of the coin, \( c' \). Joint execution of actions yields an *internal* action, \( \tau \), that is no longer accessible to any other process. For example, the synchronous composition \( V|C_1 \) (starting jointly in state \( (A,D) \)), would yield three \( \tau \)-steps to reach the state \( (A,H) \). Process \( C_1 \) is completed in this state, and \( V \) may serve another process. \( V|C_4 \) yields an infinite path of \( \tau \)-actions. \( V|C_5 \) gets stuck in state \( (C,W) \). To continue, a third process with an initial \( b? \)-step was necessary. Finally, \( V|C_6 \) can not jointly execute any step.

**Asynchronous** communication assumes processes to send and receive items along buffers. For example, a client may insert a coin into the buffer \( c \). The vending machine eventually may receive the coin. Hence a state is intermediately reached, in which the coin has already been inserted into \( c \), but not yet removed from \( c \). For example, the asynchronous composition \( V|C_1 \) (starting jointly in the state \( (A,D) \)) yields two paths, each of them comprising six steps, to reach the state \( (A,H) \). Process \( C_1 \) is completed in this state, and \( V \) is ready to serve an other process. Each of the three buffers \( c \), \( t \), and \( h \) intermediately carry an item. This resembles the synchronous case, and so do \( V|C_2 \),...,\( V|C_4 \). \( V|C_5 \) yields an infinite behavior, producing an unbounded number of beverages. \( V|C_6 \) yields one of the two paths of \( V|C_1 \).

It has frequently been propagated that synchronous handshaking and asynchronous message passing communication can mutually be simulated by one another. However, any given real-world system is either synchronous or asynchronous and this fact manifests itself in an immutable interface requiring the former or the latter mode of communication, respectively. The corresponding environment, the set of adequate partners, etc., look quite different for the two cases.

Modern software architectures are mostly based on asynchronously communicating processes. Typical examples include component based and service oriented architectures. The software industry fosters this kind of architecture for a long time. A solid, systematic, conceptual approach to the essentials of such architectures is however still missing. This paper is intended to contribute to such an enterprise.

We suggest a generic model for buffered communication, just as process algebras are generic for handshake communication: Both models equip the neutral structure of transition systems with the basic means for a specific form of communication.

The rest of this paper is structured as follows:

We begin by introducing the data structure of *buffers* ([Section 2.1](#)) and equip conventional transition systems with such buffers ([Sections 2.2](#) and [2.3](#)).

Given a process \( L \), we are frequently interested in partners \( R \) of \( L \) such that the composed system \( R \oplus L \) meets a certain property, such as (weak) termination bounded communication, etc. This implies a huge number of problems, concerning decision- and construction procedures, classifications of partners, etc. In [Section 3](#) we consider the elementary class of linear time safety properties. We conceive this section as a paradigm and as a starting point for much more involved classes of...
properties. In fact, more involved single properties have been investigated, mostly in an ad-hoc manner, and tuned to specific real-life applications. Section 4 surveys corresponding results from the literature. We compare our approach with the state of the art in Section 5.

2 Buffered Processes and Their Composition

Asynchronously communicating processes assume buffers as a fundamental data structure: Processes insert or remove items into or out of buffers. We consider the case where the items in a buffer are not individually identified, but just counted (items may be distinguished by storing them in different buffers). Consequently, a state of a process assigns each buffer a number. A step may increment or decrement this number for one of the buffers (thus representing insertion or removal of an item), or may leave the number unchanged (in case of an internal action). A computation, i.e. a sequence of steps, is feasible as long as each state holds a non-negative number. Technically it turns out very useful however to consider infeasible, “barred” steps and computations, too.

Summarizing, we model a buffered process \( P \) as transition system with distinctly formed states and steps: Each state of \( P \) contains information about its location in \( P \), and about the content of each buffer. Each step of \( P \) may update the location, or the buffer contents, or both.

2.1 Buffer Contents, Buffer Actions, and Buffered Steps

**General Assumption 1** In the remainder of this paper we assume a finite set \( B \) of buffer symbols (or just buffers).

**Definition 2.1 (Buffer Contents)**

(i) A buffer contents \( C \) of \( B \) is a general multiset over \( B \).

(ii) A buffer contents \( C \) is feasible iff \( C \geq [\] \).

**Definition 2.2 (Action, Action Set)**

(i) For each buffer \( b \), the actions of \( b \) are the general multisets \( [b] \) (insertion action) and \( [-b] \) (removal action), written as \( b! \) and \( b? \), respectively.

(ii) The empty multiset \( [\] \) is the silent action, written as \( \tau \).

(iii) Let \( B! \) and \( B? \) denote the set of all insertion actions and removal actions, respectively, of all buffers in \( B \). The set \( \text{ACT} = \{b! \cup B? \cup \{[\] \} \} \) is the action set of \( B \).

In close analogy to CSP and other process calculi, an action inserts or removes an item into or out of a buffer. Insertion of an item into a buffer \( b \) technically corresponds to the addition of \( [b] \) to the actual buffer contents. Likewise, the removal of an item from \( b \) is the subtraction of \( [b] \), i.e. the addition of \( [-b] \).
A buffered process has states composed of a location and a buffer contents.

**Definition 2.3 (Buffered States, Feasible States)**

(i) A buffered state \(s\) is shaped \((l, C)\) with \(C\) being a buffer content. We call \(l\) the location of \(p\).

(ii) A buffered state \((l, C)\) is feasible if \(C\) is feasible.

In graphical representations (such as forthcoming Figure 3), a state \((l, C)\) is usually written as “\(lC\)”.

As usual, a step links two states. In addition, a buffered step is labeled by an action that describes the update of the two involved states:

**Definition 2.4 (Buffered Step, Feasible Buffered Step)**

(i) Let \((l, C)\) be a buffered state, let \(m\) be a location and let \(a \in \text{ACT}\). Then the triple \((((l, C), a, (m, C + a))\) is a buffered step. It is usually written \((l, C) \xrightarrow{a} (m, C + a)\).

(ii) A buffered step \(p \xrightarrow{a} q\) is feasible iff both \(p\) and \(q\) are feasible.

**Figure 3** Examples of buffered steps with locations \(\{s, t\}\) and \(b, c \in B\). States \((l, C)\) are written “\(lC\)”. Infeasible parts are shaded.

**Figure 3** depicts buffered states and steps. Non-feasible states and steps are shaded. The above definition of buffered steps is somewhat redundant: In a step \((l, C) \xrightarrow{a} (m, D)\), the set \(D\) can be derived from \(C\) and \(a\). Likewise, \(C\) can be derived from \(D\) and \(a\), and \(a\) can be derived from \(C\) and \(D\). We exploit this observation in the sequel.

2.2 Buffered processes

Technically, a buffered process is a transition system with each state assigned a buffer contents. In the initial state, all buffers are empty. A step may increment and decrement at most one buffer’s contents by 1.

Different steps of a buffered process may increment and decrement the contents of the same buffer. This effect seems to intuitively spoil the idea of buffers as a means to connect different processes. However, this effect arises naturally in each process that is composed of two more elementary processes.

It may appear even more surprising that we also include negative buffer contents. In fact, each path \(w\) of a buffered process has a feasible prefix, consisting of states with non-negative buffer contents. A state with negative contents of a buffer \(b\)bars \(w\) to continue. An other process may however insert an item into \(b\), thus
extending the feasible prefix of \( w \). This is the asynchronous buffered counterpart of the synchronous handshake communication as it appears in the central idea of process algebras (where a step shaped \( s \xrightarrow{\tau} s' \) turns into an internal \( \tau \)-step in case an other process in the environment contributes \( \bar{a} \)).

**Definition 2.5 (Buffered Process)** Let \( P \) be a transition system, such that

1. each state \( p \) of \( P \) and each step \( u \) of \( P \) is buffered,
2. the initial state of \( P \) is formed \((\ell,[])\) for some location \( \ell \).

Then \( P \) is a buffered process.

Figures 4 and 5 show examples of buffered processes. A distinguished buffered process is the empty process ZERO, consisting of just one, the initial, state. Choice of its location is irrelevant:

**Definition 2.6 (ZERO Process)** Let \( \ell \) be a location and let \( p =_{def} (\ell,[]) \). the process \( \text{ZERO} = (\{p\},\emptyset,p) \) is the empty buffered process.

**Notation 2.1** We write \( \text{BP} \) for the set of all buffered processes.

Fig. 4 A buffered process with locations \( \{E,F,G,H\} \) and \( c,t,b \in B \).

Fig. 5 A buffered process with locations \( \{A,B,C,D\} \) and \( b,c,j,t \in B \).

Sometimes we are interested in only the feasible part of a buffered process.

**Definition 2.7 (Feasible part)** Let \( P \) be a buffered process. The feasible part \( \mathcal{F}(P) \) of \( P \) is the buffered process, inductively defined as follows:

1. The initial state of \( \mathcal{F}(P) \) is the initial state of \( P \).
2. Let \( p \) be a state of \( \mathcal{F}(P) \) and let \( u = p \xrightarrow{\alpha} q \) be a feasible step of \( P \). Then, \( q \) is a state of \( \mathcal{F}(P) \) and \( u \) is a step of \( \mathcal{F}(P) \).
The non-shaded part of Figures 4 and 5 show the feasible part of these processes, respectively.

**Observation 2.1** All states and steps of \( \mathcal{F}(P) \) are feasible. All states and steps of \( \mathcal{F}(P) \) are states and steps of \( P \), respectively. Each path of \( \mathcal{F}(P) \) is a path of \( P \). In general, a maximal path of \( \mathcal{F}(P) \) is not necessarily maximal in \( P \). Still, each infinite (and thus maximal) path of \( \mathcal{F}(P) \) is a maximal path of \( P \). The converse does not hold: There may exist infinite paths of \( P \) not being paths of \( \mathcal{F}(P) \).

### 2.3 Complete Processes and Process Schemata

We introduce the subclass of complete buffered processes, which are matching our intuition of buffers: The intuitive notion of buffers implies for an action \( a \in \text{ACT} \), and two locations \( s \) and \( s' \) of a process \( P \), that either \( P \) has steps shaped

\[
(s, C) \xrightarrow{a} (s', C')
\]

for all buffer contents \( C \) (many of which are not feasible), or \( P \) has no such steps at all. Put differently, a complete process chooses its next steps solely on its location, and not on actual buffer contents.

**Definition 2.8 (Complete Process)** Let \( P \) be a buffered process. \( P \) is complete if and only if for each buffer contents \( E \) holds:

(i) for each state \( (s,C) \) of \( P \), \( (s,E) \) is a state of \( P \), and

(ii) for each step \( (s,C) \xrightarrow{a} (t,D) \) of \( P \),

\[
(s,C + E) \xrightarrow{a} (t,D + E)
\]

is also a step of \( P \).

The left part of Figure 6 outlines a finite part of a complete buffered process.

**Notation 2.2** We write \( \mathcal{C}P \) for the set of complete processes.

We frequently assume buffered processes to be complete. Complete processes have a series of compelling properties. Most importantly, each complete process \( P \) with finitely many locations has a finite representation: The schema of \( P \). Intuitively, the schema of a buffered process is the projection to its locations. Thus, the schema abstracts from concrete buffer contents.

**Definition 2.9 (Schema)** Let \( P \) be a buffered process. Then the schema of \( P \) is the transition system \( \text{schema}(P) \) with:

(i) \( s \in \text{Q}_{\text{schema}(P)} \) iff for some buffer contents \( C \) holds: \( (s,C) \in \text{Q}_P \)

(ii) \( s \xrightarrow{a}_{\text{schema}(P)} t \) iff for some buffer contents \( C, D \) holds: \( (s,C) \xrightarrow{a}_P (t,D) \)

(iii) \( s^0 \) is the initial state of \( \text{schema}(P) \) iff \( (s^0,[]) \) is the initial state of \( P \).
Figure 6 outlines a technical example for a complete buffered process and the corresponding schema. We conceive the transition systems of Figures 1 and 2 as schemata of buffer complete processes: The beverage vending machine $V$ of Figure 1 is the schema of the process as outlined in Figure 5. Likewise, the tea drinker $C_1$ of Figure 2 is the schema of a buffer complete process that includes Figure 4.

We show that the schema of a complete buffered process $P$ carries all information of $P$. To this end, we define unfolding of a transition system $P$; that is, the translation of a schema to a buffered process.

**Definition 2.10 (Unfolding)** Let $P$ be a transition system. The unfolding $\text{unfold}(P)$ of $P$ is the buffered process satisfying

(i) Let $s$ be a state of $P$ and let $C$ be a buffer contents. Then, $(s, C)$ is a state of $\text{unfold}(P)$.

(ii) Let $u = s \xrightarrow{a} t$ be a step of $P$. Let $v = (s, C) \xrightarrow{a} (t, D)$ be a buffered step. Then, $v$ is a step of $\text{unfold}(P)$.

(iii) Let $s^0$ be the initial state of $P$. Then, $(s^0, [\text{ ]})$ is the initial state of $\text{unfold}(P)$.

The left part of Figure 6 outlines a finite part of a complete buffered process.

**Lemma 2.1** Let $P$ be a buffered process. Then $\text{unfold}(\text{schema}(P))$ is a complete process.

See Section 7.1 for the proof.

Unfolding a schema of a buffered process $P$ yields $P$ again (Figure 6 visualizes this relationship):

**Theorem 2.1** Let $P$ be a complete buffered process. Then

$$\text{unfold}(\text{schema}(P)) = P$$

See Section 7.2 for the proof.

Thus, the schema of a complete process $P$ is in fact an excellent representation of $P$: It is finite iff $P$ contains only finitely many locations, and it is a lossless representation of $P$.

Each schema $\text{schema}(P)$ of a complete process $P$ can be represented as a Petri net [27], consisting of a state machine for $\text{schema}(P)$, and an unbounded place for each buffer. Process $\text{schema}(P)$ is then isomorphic to the Petri net’s reachability graph. Figure 7 shows examples.
2.4 The composition of buffered processes

As discussed in the introduction, we now search for a representation of two processes $L$ and $R$, asynchronously communicating along buffers they share. We achieve this by composing $L$ and $R$, yielding a process $L \oplus R$. To this end, we compose states of $L$ and $R$, yielding a new buffered state.

**Definition 2.11 (Buffered state composition)** Let $(\ell, C)$, $(r, D)$ be buffered states. The buffered state $(\ell, C) \oplus (r, D) = \text{def} ((\ell, r), C + D)$ is the composition of $(\ell, C)$ and $(r, D)$.

We assume the sets $[p]$ and $[q]$ of atomic propositions valid in $p$ and $q$ to determine the set of atomic propositions valid in $p \oplus q$. We postpone further details to Section 3.

The paths of the composition of $L$ and $R$ shuffle the paths of $L$ with the paths of $R$. As $L$ and $R$ are transition systems, the transition system product $L \times R$ (cf. Definition 6.3) is a good candidate for $L \times R$. We replace each state $(p, q)$ in $L \times R$ with $p \oplus q$, representing the effect of both $L$ and $R$ to the buffers. Therefore, each path $w$ of $L \times R$ shuffles two paths $w_L$ and $w_R$ of $L$ and $R$, respectively, and the buffer contents of the resulting state represents the combined effects of $w_L$ and $w_R$ on the buffers.

**Definition 2.12 (Composition of buffered processes)** Let $L$ and $R$ be two buffered processes. The composition $L \oplus R$ of $L$ and $R$ is the buffered process defined as follows:

1. For each state $(p, q)$ of $L \times R$, the buffered state $p \oplus q$ is a state of $L \oplus R$.
2. For each step $(p, q) \xrightarrow{a} (p', q')$ of $L \times R$, the buffered step $p \oplus q \xrightarrow{a} p' \oplus q'$ is a step of $L \oplus R$.
3. For the initial state $(p, q)$ of $L \times R$, the buffered state $p \oplus q$ is the initial state of $L \oplus R$. 

![Petri net representation of the vending machine from Figure 1 and two clients from Figure 2.](image)
Fig. 8 composition of two buffered processes

Fig. 9 Petri net representations of processes in Figures 1 and 2 and their composition.

Figure 8 shows a technical example. Figure 10 outlines the composition of the unfoldings of the vending machine from Figure 1, and the first client (see Figure 2).

One easily realizes that the operator $\oplus$ is commutative ($L \oplus R = R \oplus L$) and associative (($L \oplus R) \oplus U = L \oplus (R \oplus U)$) up to obvious bracketing. Furthermore, the ZERO process (see Definition 2.6) is a neutral element, viz. for each buffered process $P$ holds: $\text{ZERO} \oplus P = P \oplus \text{ZERO} = P$. All this yields a well-known algebraic structure:

**Observation 2.2** $(BP; \oplus, \text{ZERO})$ is a commutative monoid.

Associativity of the composition operator makes composition particularly intuitive. Figure 11 shows a traditional family, composed of a father (who pays), a mother (who selects the beverage) and a child (who drinks). This family constitutes the **most liberal** user of the vending machine depicted in Figure 1: the feasible prefixes of all paths of all users of $V$ in Figure 2 (except the forgetful juice drinker) are also feasible prefixes of a path of the family. Furthermore, the family process exhibits a maximum of concurrency.
Fig. 10 Composition $\text{unfold}(V) \oplus \text{unfold}(C_1)$ of the unfoldings of the vending machine $V$ from Figure 1 and the tea client $C_1$ from Figure 2 (with obvious shorthands). This process apparently has four feasible prefixes.

As discussed above, each path $w$ of $L \oplus R$ is shuffled from paths $w_L$ and $w_R$ of $L$ and $R$, respectively. It is interesting to observe that the feasible prefix of $w$ may exceed the feasible prefixes of both $w_L$ and $w_R$. For example, in Figure 8 the feasible prefix of the path $w = a!a?b!$ of $L \oplus R$ is $w$ itself. $w$ is shuffled from $w_L = a!$ and $w_R = a?b!$. The feasible prefix of $w_L$ and $w_R$ is $a!$ and the empty sequence, respectively.

2.5 The Composition of Complete Buffered Processes and Process Schemata

The world of complete processes is particularly interesting:
Lemma 2.2 The composition \( L \oplus R \) of two complete processes \( L \) and \( R \) yields a complete process.

Fig. 12 The composition of the vending machine from Figure 1 and client \( C_1 \) from Figure 2.

The proof can be found in Section 7.3. The composition of \( V \) and \( C_5 \) in Figure 9 exemplifies this lemma.

Hence, the set \( \text{CP} \) of all complete processes is closed under composition; and the process ZERO in fact, is complete. In algebraic terms:

Observation 2.3 The structure \((\text{CP}; \oplus, \text{ZERO})\) is a submonoid of \((\text{BP}; \oplus, \text{ZERO})\).

The schema representation of complete processes is particularly useful in the context of composition: Schema composition coincides with the classical transition system product. The following three propositions prepare this result, stated in Theorem 2.2.

Proposition 2.1 Let \( L, R \) be complete buffered processes. Then the following holds:

\[
Q_{\text{schema}}(L \oplus R) = Q_{\text{schema}}(L) \times Q_{\text{schema}}(R).
\]

For the proof, see Section 7.4.

Proposition 2.2 Let \( L, R \) be complete buffered processes. Then each step of \( \text{schema}(L \oplus R) \) is also a step of \( \text{schema}(L) \times \text{schema}(R) \).

See Section 7.5 for the proof.

Proposition 2.3 Let \((\ell_0, [])\) and \((r_0, [])\) be the initial states of two complete buffered processes \( L \) and \( R \), respectively. Then \((\ell_0, r_0)\) is the initial state of both, \( \text{schema}(L \oplus R) \) and \( \text{schema}(L) \times \text{schema}(R) \).
Asynchronous Communication

For the proof, see [Section 7.6]. Process composition corresponds to the classical transition system product of their schemata:

**Theorem 2.2** Let $L$ and $R$ be complete buffered processes. Then

$$\text{schema}(L \oplus R) = \text{schema}(L) \times \text{schema}(R)$$

[Section 7.7] features the proof. In fact, this Theorem makes process schemata an ideal representation for complete processes.

### 2.6 Composition and Simulation

As processes are transition systems, the relation of simulation (see [Definition 6.5]) is well defined on the set $\text{BP}$ of all buffered processes. For example, in [Figure 2] $C_3$ simulates $C_1$, as well as $C_2$; furthermore, $C_4$ simulates $C_1$. Simulation induces the well-known simulation preorder $\leq$ on $\text{BP}$ (see [Proposition 6.1]). It is interesting to observe that simulation preserves the composition of complete buffered processes:

**Lemma 2.3** Let $L$, $R$, $S$ be buffered processes and let $L$ and $S$ be complete. If $R$ simulates $S$ then $L \oplus R$ simulates $L \oplus S$.

The proof can be found in [Section 7.8].

**Corollary 1** Let $L$, $L'$, $R$, $R'$ be complete buffered processes.

(i) If $L$ simulates $L'$ then $L \oplus R$ simulates $L' \oplus R$.

(ii) If $L$ simulates $L'$ and $R$ simulates $R'$, then $L \oplus R$ simulates $L' \oplus R'$.

(iii) Simulation is a precongruence on $(\text{CP}, \oplus)$.

**Proof.** (i) follows from [Lemma 2.3] and the fact that $\oplus$ is commutative. For (ii), observe that $L \oplus R$ simulates $L \oplus R'$ (by [Lemma 2.3]). Furthermore, $L \oplus R'$ simulates $L' \oplus R'$ (by (i)). Transitivity of simulation then yields (ii). (iii) follows from (ii). $\square$

Moreover, we observe that simulation also respects the restriction to the feasible part of a buffered process:

**Lemma 2.4** Let $R$, $S$ be buffered processes. Then

(i) $R$ simulates $\mathcal{F}(R)$.

(ii) If $R$ simulates $S$ then $\mathcal{F}(R)$ simulates $\mathcal{F}(S)$.

(iii) Simulation is a precongruence on $(\text{BP}, \mathcal{F})$.

[Section 7.9] features the proof.
3 Property-Asserting Partners

3.1 Linear Time Safety Properties

The semantics of a conventional program $P$ is a partial function $f$: for a given input $a$, the program $P$ outputs $f(a)$ upon termination. If the computation of $P$ fails to terminate, $f(a)$ remains undefined.

The semantics of a buffered process can not be conceived as a partial function. For instance, the composed process $V \oplus C_4$ of the vending machine and the thirsty client yields a non-terminating, but nevertheless sensible behavior. Consequently, the quest for a proper notion of the semantics of a buffered process is challenging (as it is for process algebras and other process models).

Based on the intended use of buffered processes, i.e., the composition of processes, we assume the semantics of a process $P$ to be covered in properties of composed processes $P \oplus U$. Typically, both processes $P$ and $U$ eventually achieve a joint goal. For example, the semantics of the vending machine $V$ is covered in its composition with its (infinitely many) clients. Together with client $C_1$ the composition $V \oplus C_1$ reaches its goal state $(A,H)$: The vending machine returns to its initial state, and $C_1$ obtains a beverage. This likewise applies to $C_2$, $C_3$, $\ldots$, $C_6$ with the corresponding goal states $(A,L)$, $(A,P)$, etc.

Apart from the chance or the guarantee of jointly reaching a goal state, a typical property of a composed process is the bound of the number of tokens that a buffer intermediately contains. For example, the coin buffer in $V \oplus C_1$ never contains more than one token (and no token in the goal state $(A,H)$). The same also holds for all buffers and all other clients from Figure 2, except the beverage buffer and the forgetful client, $C_5$.

Viewed in a general framework, for a property $\varphi$ of (composed) buffered processes, a process $R$ is a $\varphi$-partner of a process $L$ iff $R$ asserts $\varphi$ for the feasible part of $L\oplus R$. In this work, we consider complete buffered processes and their complete $\varphi$-partners.

Definition 3.1 ($\varphi$-Partner, Set of all $\varphi$-Partners) Let $L$ be a complete buffered process and let $\varphi$ be a property (i.e., a subset) of buffered processes.

(i) A complete buffered process $R$ is a $\varphi$-partner of $L$ iff

$$F(L \oplus R) \in \varphi.$$  

(ii) Let $\text{Proc}(L,\varphi)$ denote the set of all $\varphi$-partners of $L$.

As buffered processes are intended to communicate with others, $\varphi$-partners play a decisive role in a theory of processes. One might even define the semantics of a process $L$ as the collection of all sets $\text{Proc}(L,\varphi)$, for all properties $\varphi$. The following questions are particularly relevant:

- Does $L$ have $\varphi$-partners at all?
- Is a given buffered process $R$ a $\varphi$-partner of $L$?
- How can we characterize the—generally infinite—set $\text{Proc}(L,\varphi)$?
- May $L$ be replaced by a given $L'$ such that $\text{Proc}(L,\varphi) \subseteq \text{Proc}(L',\varphi)$?
These questions are not decidable in general. So it is reasonable to concentrate on a class $\Phi$ of properties which renders these questions decidable, while still covering really relevant properties. Motivated by case studies and by the following theorems and lemmata, we decide in favor of linear time safety properties (see Definition 6.11). As formalized in Assumption 3 in the appendix, we assume a set of atomic propositions valid in each state. We refine this assumption for buffered states and their composition. To this end, we introduce buffer equations as special atomic propositions.

**Definition 3.2** Let $b$ be a buffer and $n \in \mathbb{Z}$. Then, $b \upharpoonright n$ is a buffer equation.

The buffer contents of a buffered state determines validity of each buffer equation. For all other atomic propositions, we merely assume that composition preserves their validity.

**General Assumption 2** We assume a set $\text{AP}_{\text{BP}}$ of atomic propositions for all buffered states; that is, $[p] \subseteq \text{AP}_{\text{BP}}$ for each buffered state $p$. We assume each buffer equation to be an element of $\text{AP}_{\text{BP}}$. Let $x \in \text{AP}_{\text{BP}}$ be an atomic proposition.

(i) If $x$ is a buffer equation $b \upharpoonright n$, we assume for each buffered state $p = (\ell, C)$:

$$p \models x \iff C(b) = n.$$  

(ii) Otherwise, we assume for all buffered states $p = (\ell, C)$ and $q = (r, E)$:

(a) If $\ell = r$ and $p \models x$ then $q \models x$, and

(b) $x \in [p \oplus q]$ iff $x \in [p] \cup [q]$.

**Notation 3.1** Let $\Phi$ denote the set of all linear time safety properties with atomic propositions over $\text{AP}_{\text{BP}}$.

The following are examples for linear safety properties in the context of our vending machine:

- “In every reachable state, there is at most one beverage and one coin on the buffer.” This may for instance be written as an LTL-formula $\square((\text{beverage} \uparrow 0 \lor \text{beverage} \uparrow 1) \land (\text{coin} \uparrow 0 \lor \text{coin} \uparrow 1))$. The processes $C_1$, $C_2$, $C_3$, $C_4$ and $C_6$ assert this property. However, $C_5$ does not: $C_5$ does not consume the beverage before ordering a new one.

- “No beverage is served before a coin was received.” We observe that every buffered process asserts this property if it does not produce the beverage itself.

- Let us assume an atomic proposition satisfied. Let satisfied be valid in every state of a client after consuming the beverage, for instance, $C_3$ is satisfied in $T$. Let $C_3$ be additionally satisfied in state $O$. Let $C_5$ be satisfied in state $W$. Consider the property “Whenever $V$ is in state $A$ and the client is satisfied, all buffers are empty.” Then, $C_1$, $C_2$, $C_4$ and $C_6$ assert the property, but $C_3$ and $C_5$ do not. $C_3$ and $C_5$ fail, because they are each satisfied without having consumed the beverage.

A process $L$ either has no $\varphi$-partners at all, or $L$ has infinitely many $\varphi$-partners. In the latter case there exists a maximal partner. (In fact, there are infinitely many maximal partners that mutually simulate one another):

**Lemma 3.1** Let $L$ be a complete buffered process and let $\varphi \in \Phi$. The set $\text{Proc}(L, \varphi)$ is either empty or has a maximal element w.r.t. simulation.
The proof can be found in Section 7.10.

It is important to observe that simulation respects composition with $\varphi$-partners, i.e., that $\text{Proc}(L, \varphi)$ is downward closed in $\text{BP}$. Figure 13 outlines Lemma 3.1 and the forthcoming theorem:

**Theorem 3.1** Let $L, R, S$ be complete buffered processes and let $\varphi \in \Phi$. If $R \in \text{Proc}(L, \varphi)$ and $R$ simulates $S$, then $S \in \text{Proc}(L, \varphi)$.

**Proof.** By Lemma 2.3, $L \oplus R$ simulates $L \oplus S$. By Lemma 2.4, $F(L \oplus R)$ simulates $F(L \oplus S)$. By Definition 3.1, $F(L \oplus R)$ satisfies $\varphi$. By Proposition 6.4, simulation preserves $\varphi$, and thus, $F(L \oplus S)$ satisfies $\varphi$. Finally, by Definition 3.1, $S \in \text{Proc}(L, \varphi)$.

Together with the above Lemma 3.1, this theorem implies that simulation characterizes the set $\text{Proc}(L, \varphi)$ of all $\varphi$-partners of $L$:

**Corollary 2** Let $L, M, R$ be complete buffered processes and let $\varphi \in \Phi$ such that $M$ is maximal in $\text{Proc}(L, \varphi)$. Then $R \in \text{Proc}(L, \varphi)$ iff $M$ simulates $R$.

![Fig. 13](image)

*Fig. 13* Let $P \rightarrow Q$ iff $Q$ simulates $P$. For a given process $L$ and a property $\varphi \in \Phi$, $\text{Proc}(L, \varphi)$ is downward closed in $\text{BP}$ and has maximal elements w.r.t. the simulation preorder.

### 3.2 Regular and Bounded Safety Properties

**Corollary 2** allows a beautiful characterization of the set of all $\varphi$-partners of $L$, in case there exists a maximal such partner. Here we prove the existence of a finite maximal partner for all linear time safety properties $\varphi$ that additionally require regularity and bounded buffers. Regularity of safety properties is a well-known concept \[1\]. Intuitively, this means that the set of traces ending in a “bad” state is a regular language, and thus is representable by a finite automaton. Boundedness of buffers means that there exists a number $n \in \mathbb{N}$ such that $C \leq n \cdot |B|$ for each state $(\ell, C)$ of $F(L \oplus R)$.

**Definition 3.3** Let $k \in \mathbb{N}$. 

(i) A buffered process $P$ is $k$-bounded iff for each state $(p,C)$ of $\mathcal{F}(P)$ holds: $C \leq k \cdot |B|$. 

(ii) A property $\varphi$ is $k$-bounded iff each $P \in \varphi$ is $k$-bounded.

Let the schema of $L$ be finite, and $\varphi$ be $k$-bounded. Then, there exists a maximal $\varphi$-partner of $L$ with a finite schema.

**Theorem 3.2** Let $L$ be a complete buffered process. Let $\text{schema}(L)$ be finite. Moreover, let $k \in \mathbb{N}$ and let $\varphi$ be a regular $k$-bounded linear time safety property. Then there exists a maximal element $M \in \text{Proc}(L, \varphi)$, such that $\text{schema}(M)$ is finite.

The proof can be found in [Section 7.11](#).

Yet, it is useful to have effective decision procedures and construction methods for problems of $\varphi$-partners. Such procedures and methods require properties to be represented in the syntax of a logic. As we consider linear time safety properties, the logic LTL is an obvious choice.

The postulate of $k$-boundedness can easily be encoded in LTL as the formula

$$\square \bigwedge_{b \in B} \left( \bigvee_{j=0}^{k} b \doteq j \right).$$

using the finitely many buffer equations $b \doteq j$, for all $b \in B$ and all $0 \leq j \leq k$. It is not necessary to include $b \doteq j$ for $j < 0$ because such a buffer equation is not valid in any state of the feasible part of any buffered process.

This concludes the quest for the proper treatment of property-asserting partners of a buffered process $L$: For regular and bounded linear time safety properties there exists a canonical finite partner $R^*_\varphi\mathcal{L}$ which simulates all other partners. $R^*_\varphi\mathcal{L}$ can be constructed efficiently. A construction method is easily obtained from the proof of Theorem 3.2.

### 4 Further partner classes and problems

A central aspect of asynchronously communicating processes are property asserting partners: For a given buffered process $L$ and a property $\varphi$, the set $\text{Proc}(L, \varphi)$ contains a process $R$ iff the feasible part of $L \oplus R$ has the property $\varphi$. There are usually infinitely many such processes $R$, so it is not trivial to answer any question concerning $\text{Proc}(L, \varphi)$. In [Section 3.2](#) we developed a technique to cope with $\text{Proc}(L, \varphi)$ for the special case of regular and bounded linear time safety properties.

Literature discusses other properties and further problems concerning $\text{Proc}(L, \varphi)$. Most publications follow principles similar to those adapted in this contribution: A distinguished partner $R^*$, usually called operating guideline [17], simulates all other $\varphi$-partners of $L$. For interesting properties $\varphi$, however, not each simulated buffered process $R$ belongs to $\text{Proc}(L, \varphi)$. Examples of such properties include deadlock-freedom [17] and weak termination [3].

To capture exactly the $\varphi$-partners of $L$, the outgoing arcs of each node of a simulated $R$ are required to meet a set of properties assigned to the simulating node of $R^*$.
Some properties are propositional combinations of more elementary ones. So it is interesting to study union, intersection and complement of sets $\text{Proc}(L, \varphi)$ for fixed $\varphi$ and varying $L$. Kaschner [10] introduces techniques for this kind of argument.

The rest of this section surveys so far published results on $\text{Proc}(L, \varphi)$. Most of them have been implemented in software tools. For more information, we kindly point the reader to http://service-technology.org.

4.1 Verification and Testing

In this subsection we introduce problems to assess whether a process can function properly, conforms to a specification or interacts correctly with another process.

4.1.1 $\varphi$-asserting Interaction

For given processes $L$ and $R$ and a given property $\varphi$ it is a fundamental problem to decide whether or not $R \in \text{Proc}(L, \varphi)$. With Def. 3.1 this reduces to the problem $F(L \oplus R) \in \varphi$. This problem can be attacked by a model checker. LoLA [29], a model checker for Petri nets, is a good fit because buffered processes expressed in industrial languages (e.g., BPEL [25]) can be translated into Petri nets (e.g., using BPEL2oWFN [16]), which LoLA can subsequently model-check. This approach, however, comes with two caveats: First, model checking is costly. Second, it only works if the provider of $L$ reveals the internals of $L$. An approach based on operating guidelines circumvents these limitations. Instead of publishing $L$, the provider publishes the operating guideline of $L$. Then $R$ is matched with $L$‘s operating guideline. [17] discusses the matching procedure in more detail and also discusses implementation details. The matching algorithm is implemented in Fiona [20]. Lohmann et al. propose a more compact representation of operating guidelines [19], replacing boolean formulas by a few bits. Besides saving space, this representation also leads to more efficient algorithms. The tool Cosme implements matching on this compact operating guideline.

There is another approach to matching in [24] based on fingerprints. A fingerprint abstractly describes the interaction behavior of a process in terms of systems of inequalities. The tool Linda computes the fingerprints of $L$ and $R$ and Yasmina composes them yielding a new fingerprint. If Yasmina detects that the composed fingerprint is not solvable, then $R$ does not match $L$. Otherwise, the result is inconclusive and matching must be checked via other means.

4.1.2 Controllability

A buffered process $L$ should be published only if there exists at least one $\varphi$-partner. This property is known as controllability. [36] contains decidability results about controllability in various settings. A more algorithmic approach to verifying controllability is presented in [38], which also features its implementation in Fiona. The underlying idea is to synthesize an arbitrary partner of $L$ if $L$ is controllable. Likewise, Wendy [18] (cf. Section 4.3.1) is able to check controllability of $L$.

In analogy to matching, there is also an approach for checking controllability with fingerprints [24]. Linda computes the fingerprint of $L$ and Yasmina tests
whether the fingerprint is solvable. If it is not solvable, $L$ is not controllable. Otherwise, Yasmina’s result is inconclusive. Hence the method based on fingerprints only yields a necessary condition for controllability.

4.1.3 Test Case Generation

Given a buffered process as specification and its implementation as a piece of software, we want to verify whether the implementation matches the specification, i.e., whether the implementation conforms to the specification. The usual method to find violations is to test the implementation systematically. Kaschner et al. show in [9] how to extract a finite set of test cases from the operating guideline derived from the specification.

4.1.4 Conformance Checking

Here we assume a specification and an implementation as in Section 4.1.3 augmented by a log, compiling the activities of the implementation. In [22] Müller et al. propose a method to check the conformance of an implementation to a specification using logs. Their idea is to generate a special process $L^*$, which matches the specification and has a decisive property: If $L^*$ could not have generated the log, then no process matching the specification could have generated the log. Consequently, the implementation cannot match the specification. DELAIN can compute processes with this property.

4.2 Substitution

This subsection introduces problems of replacing one process with another at either design-time (Sections 4.2.1 and 4.2.3) or run-time (Section 4.2.2).

4.2.1 Substitutability

After successfully deploying a process $L$, its provider may intend to update $L$, resulting in a process $L'$. The question is whether $L'$ can safely replace $L$. This means that every $\varphi$-partner of $L$ is also a $\varphi$-partner of $L'$. This relation is termed substitutability or accordance and can be decided using an operating guideline [30].

4.2.2 Migration

Substituting a process $L$ with another process $L'$ poses yet another challenge: Processes are often long-running (e.g., insurance processes). It may be unavoidable to migrate each running instance of $L$ to an equivalent instance of $L'$. So migration is substitution at run-time. [14] suggests constructing a migration process $M$ and substituting $L$ with $L \oplus M \oplus L'$. Thereby the state of every instance of $L$ is conserved in an instance of $L \oplus M \oplus L'$. The role of $M$ is to migrate the state of every instance of $L$ into an equivalent state of $L'$. With MIA there is a tool which computes $M$. 
4.2.3 Public View

A process of a company may contain trade secrets and algorithms, which the company wants to keep secret. Instead of publishing their original process \( L \), the company publishes a process \( L' \) which can be used instead of \( L \), but does not contain sensible information. This means that \( L \) and \( L' \) interact correctly with the same set of \( \varphi \)-partners. We term such a process \( L' \) the public view of \( L \) \cite{21}. Fiona implements an algorithm for computing the public view of a process.

4.3 Synthesis

This subsection introduces problems that amount to finding a \( \varphi \)-partner \cite{Section 4.3.1}. Sections 4.3.2 to 4.3.4 contain variants with additional assumptions.

4.3.1 Partner Synthesis

Partner synthesis is the problem of constructing a \( \varphi \)-partner \( R \) for a given process \( L \). The idea behind partner synthesis is to synthesize a maximal \( \varphi \)-partner \( R^* \) containing the behavior of every \( \varphi \)-partner. \( R^* \) serves as template for \( R \) to reduce costs and time to develop \( R \). Fiona and Wendy compute a canonical \( R^* \). As a consequence, they can also decide controllability. In addition, Fiona and Wendy can also synthesize smaller \( \varphi \)-partners. Assuming that message exchange causes costs, the tool Tara \cite{31} synthesizes cost-minimal partners.

4.3.2 Discovery

Given an implementation of a piece of software that correctly interacts with a buffered \( L \), one often wants to analyze the implementation, verify its correctness or understand it better. This is often impossible due to a missing formal model of the implementation. Even if there is one, it is frequently outdated or incomplete. Reengineering a formal model manually is tedious and error-prone. \cite{23} suggests an automated method to generate a formal model of the implementation using logs. This method was termed service discovery in analogy to process discovery \cite{33}. A genetic algorithm performing service discovery is available as a plugin for ProM\footnote{www.promtools.org/prom6} \cite{34}. The algorithm maximizes the quality of a population of processes in Proc(\( L, \varphi \)) w.r.t. fitness, simplicity, precision and generalizability (cf. \cite{23}). After the algorithm has terminated, the best candidate for the formal model is the \( \varphi \)-partner with the highest quality.

4.3.3 Adapter Synthesis

Sometimes processes \( L \) and \( R \) do not interact correctly. For instance, in a business setting, \( L \) may expect payments in dollar whereas \( R \) offers its payments in euro. A process \( M \) changing euro to dollar, would bridge this incompatibility. Hence \( L \) and \( R \) interact correctly together with \( M \). In this case \( M \) functions as an adapter \cite{6}. To ensure that an adapter does not add functionality, additional rules restrict what an
adapter is allowed to do. With these rules MARLENE can synthesize an adapter for $L$ and $R$. MARLENE can also take costs—associated with message exchange—into account to find cost-efficient adapters [5]. Gierds et al. showed in [7] that adapter synthesis can be reduced to partner synthesis.

4.3.4 Repair

As in case of adapter synthesis, let $L$ and $R$ be processes that do not interact correctly. In contrast to adapter synthesis, however, one may want to repair $R$. Repairing indicates that $R$ is altered as little as possible to achieve correct interaction with $L$. That is, the repaired process $R'$ resembles $R$ as much as possible. Resemblance for processes is defined by a simulation-based graph edit distance [15]. RACHEL finds the $R' \in \text{Proc}(L, \varphi)$ with the smallest edit distance from $R$ and, based on the edit distance, returns a list of corrections.

5 Related Work

This contribution stands in the tradition of reactive systems [8] and open systems [12], contrasting closed systems, which are entirely determined by internal states. In addition to being open, we consider processes that communicate asynchronously via unbounded buffered channels. The semantics of a buffered process $P$ is—as usual for transition systems—given by the reachable paths of $P$. A more abstract view can be gained in terms of stream processing relations:

A fixed buffer contents $C$ of a buffered process $P$ may be assumed in the initial state $s^0$. This renders a specific set of states and steps feasible. Computations along those states and steps then produce results on the buffers. Increasing the initial buffer contents then renders more states and steps feasible, hence yields more results. This is in line with the stream processing semantics of services, as suggested by Broy et al. [2]. It would be interesting to elaborate on this aspect.

We concentrate on partners of given processes, and showed means to cope with partners that assure regular safety properties. Of course, not every interesting property of partners is a regular safety property. There are also liveness and non-regular safety properties. An example of a liveness property is weak termination, which can be treated by techniques similar to those considered in Section 3. [3] presents details.

We represent the—in general infinite—set of processes $R$ with $\mathcal{F}(L \oplus R) \in \varphi$ by means of a distinguished finite $\varphi$-partner. Likewise, infinite sets of processes $R$ with $\mathcal{F}(R) \in \varphi$ can be represented by finite tree automata [32][13]. There is a substantial difference between both: The former depends on a process $L$ and a property $\varphi$ whereas the latter depends only on a property $\varphi$. Furthermore, some decision problems for tree automata are $\text{EXPTIME}$-complete (e.g. non-emptiness of intersection [4]). Therefore we believe that our approach is more efficient in our setting.

The central idea of our approach is the notion of a $\varphi$-partner $R^*$ of $L$ which is maximal w.r.t. the simulation preorder. Similarly, an operating guideline uses a $\varphi$-partner which is maximal w.r.t. a special simulation preorder. Temporal synthesis [11] is concerned with synthesizing a process that is a $\varphi$-partner for all processes.
For our approach, a $\varphi$-partner of $L$ suffices. Existing approaches consider synchronous \cite{11} and unbuffered, asynchronous \cite{26,28} communication. In contrast, we are concerned with asynchronous communication via unbounded buffers.

6 Appendix A (Transition Systems and Linear Time Properties)

In this appendix we recall necessary standard notions as well as results from literature. These are mostly adapted from \cite{1}.

6.1 General Multisets

**Definition 6.1 (General multiset, multiset)** Let $S$ be a set and let $f: S \rightarrow \mathbb{Z}$.

(i) $f$ is called a *general multiset over* $S$. $\text{Bags}(S)$ denotes all such multisets.
(ii) $f$ is called a *conventional multiset*, if $f(a) \geq 0$ for all $a \in S$.
(iii) $f$ is *finite* if $f(a) \neq 0$ for finitely many $a \in S$ only. In this case, $f$ can be written $[x_1, x_2, \ldots, x_n]$, where
- $x_i = a$ for $f(a)$ indices $i$, in case $f(a) > 0$
- $x_i = -a$ for $f(a)$ indices $i$, in case $f(a) < 0$

6.2 Transition systems

**Definition 6.2 (Transition system)** Let $Q$ and $L$ be disjoint sets. Let $p^0 \in Q$ and $\rightarrow \subseteq Q \times L \times Q$. Then the directed graph $T = \text{def} (Q, \rightarrow, p^0)$ is a *transition system*. The elements of $Q$ are *states*, the relation $\rightarrow$ is the *transition relation*, and its elements called *steps*. The state $p^0$ is the *initial state*, and $L$ contains the *transition labels* of $T$.

**General Assumption 3** As usual, we may assume a set $\text{AP}$ of atomic propositions, and a labeling function $\mathcal{L}$, which assigns to each state $p$ of a transition system the set $\mathcal{L}(p) \subseteq \text{AP}$ of atomic propositions, we assume to be valid in $p$. As usual, we write $p \models x$ for $x \in \mathcal{L}(p)$.

**Notation 6.1** As usual, we write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$. Furthermore, the components of a transition system $T$ are frequently indexed by $T$ (i.e., $Q_T$, $\rightarrow_T$, etc. instead of $Q$, $\rightarrow$, etc.).

**Definition 6.3 (Product of transition systems)** Let $L = (Q_L, \rightarrow_L, \ell^0)$ and $R = (Q_R, \rightarrow_R, v^0)$ be transition systems. The *product* of $L$ and $R$ is a transition system $L \times R$ such that

(i) $Q_{L \times R} = Q_L \times Q_R$
(ii) $(\ell, r) \xrightarrow{a} (\ell', r') \in \rightarrow_{L \times R}$ if
- $\ell \xrightarrow{a} \ell' \in \rightarrow_L$ and $r = r'$
- or $r \xrightarrow{a} r' \in \rightarrow_R$ and $\ell = \ell'$
Asynchronous Communication

\[ p^0_{L \times R} = (\ell^0, r^0). \]

**Definition 6.4 (Subsequent Step, Path, Run, Maximal Path)** Let \( T \) be a transition system. Two steps \( p \xrightarrow{a} q \) and \( p' \xrightarrow{a'} q' \) of \( T \) are called **subsequent**, denoted \( p \xrightarrow{a} q \xrightarrow{a'} p' \), if \( p' = q \). Let \( \pi = p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} \ldots \) be a finite or infinite sequence of subsequent steps of \( T \). Then, \( \pi \) is a **path** of \( T \).

1. \( \pi \) **starts** in a state \( p \), if the first step in the sequence is of shape \( p \xrightarrow{a} q \) for some \( a \in L \) and \( q \in Q \).
2. \( \pi \) **results** in a state \( q \), if its step sequence is finite and the last step is of the form \( p \xrightarrow{a} q \) for some \( a \in L \) and \( p \in Q \).
3. If \( \pi \) starts in the initial state of \( T \), then \( \pi \) is a **run** of \( T \).
4. A path \( \pi \) of \( T \) is **maximal**, if it is either infinite or it is finite and there is no subsequent step in \( T \) to extend \( \pi \).
5. A step \( p \xrightarrow{a} q \) **occurs** in \( \pi \), if it is part of the sequence of adjacent steps which defines \( \pi \).

**Notation 6.2** Let \( T \) be a transition system and let \( p \in Q_T \). Then the set set of all maximal paths starting in \( p \) is denoted as:

\[ \text{Paths}_T(p) = \{ \pi \mid \pi \text{ is a path in } T, \pi \text{ starts in } p, \pi \text{ is maximal} \} \]

We drop the subscript whenever the transition system is clear from the context. We write \( \text{Paths}(T) \) to denote all maximal paths starting in the initial state of \( T \), formally:

\[ \text{Paths}(T) = \text{Paths}_T(p^0_T) \]

### 6.3 Process equivalences

**Definition 6.5 (Simulation)** Let \( T \) and \( U \) be transition systems with state sets \( Q_T \) and \( Q_U \) and let \( W \subseteq Q_T \times Q_U \) be a binary relation. Then \( W \) is a **simulation relation** from \( T \) to \( U \), if each \( (p, p') \in W \) satisfies the following properties:

(i) \( [p] = [p'] \)
(ii) If there is a step \( u = p \xrightarrow{a} q \) in \( T \), then there is a step \( v = p' \xrightarrow{a} q' \) in \( U \) and \( (q, q') \in W \).

As known, the union of two simulation relations yields a simulation relation. Thus, the notion of the **maximal simulation relation** is valid.

**Definition 6.6 (Initial Simulation)** Let \( P \) and \( U \) be transition systems with initial states \( p^0_P, p^0_U \), respectively. Let \( W \) be a simulation relation from \( T \) to \( U \). Then, \( W \) is **initial** \( \text{iff} \ (p^0_P, p^0_U) \in W \).

**Definition 6.7 (Simulate)** Let \( T \) and \( U \) be transition systems. Then \( U \) **simulates** \( T \), \( \text{iff} \) the maximal simulation relation from \( T \) to \( U \) is initial.

**Proposition 6.1 (Simulation Induces a Preorder on Transition Systems)** For a fixed set \( AP \) of atomic propositions, the relation \( \leq \) is **reflexive** and **transitive**.
6.4 Traces

**Notation 6.3** Let $S$ be a set.

1. The set of finite words over $S$ is denoted by $S^*$. 
2. The set of infinite words over $S$ is denoted by $S^\omega$.

In the context of words, the elements of $S$ are called symbols. Let $w$ be a finite or infinite word over $S$.

1. We write $w = w_0w_1w_2 \ldots$, using $w_i, i \in \mathbb{N} \cup \{0\}$ to denote the $i$-th symbol of $w$.
2. For $i,j \in \mathbb{N} \cup \{0\}, j > i$ the notation $w[i \ldots j]$ refers to the segment of $w$ between symbols $w_i$ and $w_j$ (inclusively). We assume the indices to be valid (i.e. in range).
3. For $w \in S^\omega$ and $i \in \mathbb{N} \cup \{0\}$, $w[i \ldots]$ denotes the infinite segment of $w$ which starts at symbol $w_i$.
4. For a finite word $v$ over $S$, $vw$ denotes the concatenation of $v$ and $w$.

**Definition 6.8 (Trace of a Path)** Let $T$ be a transition system, $\pi$ be a path of $T$. Then the trace $t$ of $\pi$ is defined as follows:

1. If $\pi$ is finite, we can write it as $\pi = p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} p_3 \ldots p_n \xrightarrow{a_n} p_{n+1}, n \in \mathbb{N}$. Then $t \in (\mathcal{P}(\mathcal{AP}))^*$ such that $t = [p_1][p_2][p_3] \ldots [p_{n+1}]$.
2. If $\pi$ is infinite, we can write it as $p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} p_3 \ldots$. Then $t \in (\mathcal{P}(\mathcal{AP}))^\omega$ such that $t = [p_1][p_2][p_3] \ldots$.

**Notation 6.4** Let $T$ be a transition system and let $p \in Q_T$.

1. The notation $\text{trace}_T(\pi) = t$ expresses the fact that $t$ is the trace of $\pi$.
2. We use $\text{Traces}_T(p)$ to denote the set of all maximal traces starting in state $p$, i.e.,

$$\text{Traces}_T(p) = \{ \text{trace}(\pi) \mid \pi \in \mathcal{P}(\mathcal{AP}^\omega), \text{trace}(\pi) = [p_{t_1}][p_{t_2}] \ldots \}.$$ 

The subscript $T$ is omitted whenever no confusion can arise. Furthermore, we introduce the following notation:

1. $\text{Traces}(T) = \text{Traces}_T(p^0_T)$,
2. $\text{FiniteTraces}(T) = \{ \hat{t} \in (\mathcal{P}(\mathcal{AP}))^* \mid t \in \text{Traces}(T), \hat{t} \text{ is a finite prefix of } t \}$.

6.5 Properties

**Definition 6.9 (LT Property)** A linear time property is a set $\varphi$ such that

$$\varphi \subseteq (\mathcal{P}(\mathcal{AP}))^\omega.$$ 

**Definition 6.10 (Satisfaction Relation for LT Properties)** Let $T$ be a transition system, $p \in Q_T$, and let $\varphi$ be a linear time property. Then

1. $T$ satisfies $\varphi$ if $\text{Traces}(T) \subseteq \varphi$,
2. $p$ satisfies $\varphi$ if $\text{Traces}(p) \subseteq \varphi$. 

Notation 6.5 For a transition system $T$, a linear time property $\varphi$, and a state $p \in Q_T$ we write $T \models \varphi$ and $p \models \varphi$ to denote property satisfaction.

Definition 6.11 (Safety Properties, Bad Prefixes) Let $\varphi$ be a linear time property. $\varphi$ is safe if for all words $w \in ((\mathcal{P}(A))^\omega \setminus \varphi)$ there exists a finite prefix $\hat{w}$ of $w$ such that

$$\varphi \cap \{ w' \in (\mathcal{P}(AP))^\omega | \hat{w} \text{ is a finite prefix of } w' \} = \emptyset$$

Any such $\hat{w}$ is called a bad prefix for $\varphi$.

Notation 6.6 Let $\varphi$ be a linear time safety property. Then the set of bad prefixes for $\varphi$ is denoted as

$$\text{BadPref}(\varphi) = \{ \hat{w} \in (\mathcal{P}(AP))^* | \hat{w} \text{ is a bad prefix for } \varphi \}$$

Proposition 6.2 (Satisfaction Relation for LT Safety Properties) Let $T$ be a transition system and let $\varphi$ be a linear time safety property. Then

$$T \models \varphi \iff \text{FiniteTraces}(T) \cap \text{BadPref}(\varphi) = \emptyset$$

Theorem 6.1 (Finite Trace Inclusion and LT Safety Properties) Let $T, U$ be transition systems. Then the following statements are equivalent:

(i) $\text{FiniteTraces}(T) \subseteq \text{FiniteTraces}(U)$

(ii) For any linear time safety property $\varphi$ holds: $U \models \varphi$ implies $T \models \varphi$

Proposition 6.3 Let $T, U$ be transition systems. If $T$ simulates $U$ then $\text{FiniteTraces}(T) \supseteq \text{FiniteTraces}(U)$. If $T$ is deterministic, the reverse also holds.

Proposition 6.4 Let $R, S$ be buffered processes and let $\varphi \in \Phi$ (see Notation 3.1). If $R \in \varphi$ and $R$ simulates $S$, then $S \in \varphi$.

7 Appendix B (Proofs)

7.1 Proof of Lemma 2.1

Proof. We proceed in two steps, showing that the requirements of Definition 2.8 are fulfilled.

Ad (i). Let $(s,C)$ be a state of $P$

$$\Rightarrow$$

$s$ is a state of $\text{schema}(P)$ (by Definition 2.9)

$$\Rightarrow$$

For all buffer contents $E$, $(s,E)$ is a state of $\text{unfold}(\text{schema}(P))$ (by Definition 2.10)

$$\Rightarrow$$

Requirement (i) of Definition 2.8 holds. $\checkmark$
Ad (ii). Let \((s, C) \xrightarrow{a} (s', C')\) be a step of \(P\)

\[ s \xrightarrow{a} s' \] is a step of \(\text{schema}(P)\) (by Definition 2.9) \hspace{1cm} (1)

\[ \Rightarrow \]

For all buffer contents \(C, C'\) and for all actions \(a \in \text{ACT}\) holds: if \((s, C) \xrightarrow{a} (s', C')\) is a buffered step, then \((s, C) \xrightarrow{a} (s', C')\) is a step of \(\text{unfold}(\text{schema}(P))\) (by Definition 2.10)

\[ \Rightarrow \]

For all buffer contents \(E\) holds: if \((s, C) \xrightarrow{a} (s', C')\) is a step of \(\text{unfold}(\text{schema}(P))\), then \((s, C + E) \xrightarrow{a} (s', C' + E)\) is a step of \(\text{unfold}(\text{schema}(P))\) (by (1) and Definition 2.10 (ii))

\[ \Rightarrow \]

Requirement (ii) of Definition 2.8 holds \hspace{1cm} \checkmark

Steps (i) and (ii) yield the claim. \hspace{1cm} \square

7.2 Proof of Theorem 2.1

Proof. We show the claim in three steps.

(I) The state sets correspond.

(II) The transition relations correspond.

(III) The initial states correspond.

Ad (I). Let \((s, C)\) be a state of \(P\).

\[ \iff \]

\((s, C)\) is a state of \(\text{schema}(P)\) (by Definition 2.9) \hspace{1cm} \checkmark

Ad (II). Let \((s, C) \xrightarrow{a} (s', C')\) be a step of \(P\)

\[ \iff \]

\((s, C) \xrightarrow{a} (s', C')\) is a step of \(\text{schema}(P)\) (by Definition 2.9) \hspace{1cm} \checkmark
Ad (III). $(s,[])$ is the initial state of $P$

\[ \iff \]

\[ s \text{ is the initial state of } \text{schema}(P) \text{ (by Def. 2.9)} \]

\[ \iff \]

\[ (s,[]) \text{ is the initial state of } \text{unfold}(\text{schema}(P)) \]

Hence $\text{unfold}(\text{schema}(P)) = P$ (by (I), (II), and (III))

\[ \square \]

7.3 Proof of Lemma 2.2

**Proposition 7.1** Let $L, R$ be complete buffered processes, $((\ell, r), C) \xrightarrow{a} ((\ell', r'), C')$ be a step of $L \oplus R$, and let $E$ be any buffer contents. Then

\[ ((\ell, r), C + E) \xrightarrow{a} ((\ell', r'), C' + E) \]

is a step of $L \oplus R$.

**Proof.** There exist buffer contents $C, D, C', D'$ such that

\[ C = C + D \text{ and } C' = C' + D' \]

and

\[ ((\ell, C), (r, D)) \xrightarrow{a} ((\ell', C'), (r', D')) \]

is a step of $L \times R$.

Then either

\[ (\ell, C) \xrightarrow{a} (\ell', C') \]

or

\[ (r, D) \xrightarrow{a} (r', D') \]

Then the proposition follows by (2).

The case of (4) is analogous to (3).

\[ \square \]

We now prove Lemma 2.2

**Proof.** We have to show that the conditions of Def. 2.8 are fulfilled:

(i) Let $(p,q) \in Q_L \times Q_R$. Then $p$ and $q$ can be written as $p = (\ell, C)$ and $q = (r, D)$ for locations $\ell, r$ and buffer contents $C, D$ (by Def. 2.5).

The following holds:

1. For all buffer contents $E$, $(\ell, E)$ is a state of $L$ (by Def. 2.8 applied to $L$).
2. For all buffer contents $E'$, $(r, E')$ is a state of $R$ (by Def. 2.8 applied to $R$).
We know that \( p \oplus q = (\ell, C) \oplus (r, D) = ((\ell, r), C + D) \in Q_{L \oplus R} \) (by Definition 2.12).

Thus, for all buffer contents \( F \) there exist buffer contents \( E, E' \) such that
\[ F = E + E' \] and \( ((\ell, r), F) \in Q_{L \oplus R} \). Moreover, \( ((\ell, r), F) = (\ell, E) \oplus (r, E') \).

(ii). Follows directly from Proposition 7.1.

This concludes the proof. \( \square \)

7.4 Proof of Proposition 2.1

Proof. We show the claim in two steps

(I) \( Q_{\text{schema}(L \oplus R)} \subseteq Q_{\text{schema}(L) \times \text{schema}(R)} \)

(II) \( Q_{\text{schema}(L \oplus R)} \supseteq Q_{\text{schema}(L) \times \text{schema}(R)} \)

Ad (I). Let \( s \in Q_{\text{schema}(L \oplus R)} \).

\[ \exists E \in \text{Bags}(B) : (s, E) \in Q_{L \oplus R} \) (by Definition 2.9 (i)) \]

There exist locations \( \ell, r \) and buffer contents \( C, D \) such that \( s = (\ell, r), \)
\[ E = C + D, \] and \( ((\ell, r), C + D) \) (by Definition 2.12).

\[ \Rightarrow \]

\( ((\ell, r), (r, D)) \) is a state of \( L \times R \) (by Definitions 2.12 and 6.3).

\[ \Rightarrow \]

\( (\ell, C) \) is a state of \( L \) and \( (r, D) \) is a state of \( R \) (by Definition 6.3).

\[ \Rightarrow \]

\( \ell \) is a state of \( \text{schema}(L) \) and \( r \) is a state of \( \text{schema}(R) \) (by Definition 2.9).

\[ \Rightarrow \]

\( (\ell, r) \) is a state of \( \text{schema}(L) \times \text{schema}(R) \) (by Definition 2.9).

\[ \Rightarrow \]

\( s \) is a state of \( \text{schema}(L) \times \text{schema}(R) \) (by (5)).
Ad (II). Let \( s \in Q_{\text{schema}(L) \times \text{schema}(R)} \)

\[
\text{⇒} \\
\exists l \in Q_{\text{schema}(L)} \exists r \in Q_{\text{schema}(R)} : s = (l, r) \quad \text{(by Definition 6.3 (i))} \quad (6)
\]

\[
\text{⇒} \\
\exists C \in \text{Bags}(B) \exists D \in \text{Bags}(B) : ((l, C), (r, D)) \in Q_{L \oplus R} \quad \text{(by Definition 2.9)}
\]

\[
((l, r), C + D) \in Q_{L \oplus R} \quad \text{(by Definition 2.12)}
\]

\[
(l, r) \in Q_{\text{schema}(L \oplus R)} \quad \text{(by Definition 2.9 (i))}
\]

\[
s \in Q_{\text{schema}(L \oplus R)} \quad \text{(by (6))} \\
\checkmark
\]

Thus the proposition. \( \square \)

7.5 Proof of Proposition 2.2

Proof. We show the equivalence in two steps:

(I) \( \rightarrow_{\text{schema}(L \oplus R)} \subseteq \rightarrow_{\text{schema}(L) \times \text{schema}(R)} \)

(II) \( \rightarrow_{\text{schema}(L \oplus R)} \supseteq \rightarrow_{\text{schema}(L) \times \text{schema}(R)} \)

Ad (I). \( \sigma \) is a step of \( \text{schema}(L \oplus R) \)

\[
\Rightarrow \\
\exists C, C' \in \text{Bags}(B) : ((l, r), C) \xrightarrow{\alpha} (l', r', C') \quad \text{(by Definition 2.9 (ii))}
\]

\[
\Rightarrow \\
\text{either } (l, C) \xrightarrow{\alpha} (l', C') \text{ is a step of } L \text{ and } (r, D) = (r', D') \quad (7)
\]

or

\[
(r, D) \xrightarrow{\alpha} (r', D') \text{ is a step of } R \text{ and } (l, C) = (l', C') \quad \text{(by Definition 2.12)} \quad (8)
\]

Case 1 (7 holds). In this case, \( \ell \xrightarrow{\alpha} \ell' \) is a step of \( \text{schema}(L) \) and \( r = r' \) is a state of \( R \) (by Definition 2.9).

\[
\Rightarrow \\
(l, r) \xrightarrow{\alpha} (l', r') \text{ is a step of } \text{schema}(L) \times \text{schema}(R) \quad \text{(by Definition 6.3).}
\]
Case 2 ([8] holds). Here \( r \xrightarrow{a} r' \) is a step of \( \text{schema}(R) \) and \( \ell = \ell' \) is a state of \( L \) (by Definition 2.9).

\[
(\ell, r) \xrightarrow{a} (\ell', r') \text{ is a step of } \text{schema}(L) \times \text{schema}(R) \text{ (by Definition 6.3).}
\]

\[
\implies 
\sigma \text{ is a step of } \text{schema}(L) \times \text{schema}(R) \quad \checkmark
\]

\[
\text{Ad (II). } \sigma \text{ is a step of } \text{schema}(L) \times \text{schema}(R)
\]

\[
\implies
(\ell, r) \xrightarrow{a_{\text{schema}(L) \times \text{schema}(R)}} (\ell', r') \text{ (by Def. } \sigma) \quad \checkmark
\]

either \( \ell \xrightarrow{a} \ell' \) is a step of \( \text{schema}(L) \) and \( r = r' \) (9)

or

\[
\sigma \text{ is a step of } \text{schema}(L) \times \text{schema}(R)
\]

\[
\implies 
\sigma \text{ is a step of } \text{schema}(L \oplus R) \quad \checkmark
\]

This completes the proof. \( \Box \)
7.6 Proof of Proposition 2.3

Proof. The following holds

\[ \ell_0 \text{ is the initial state of } \text{schema}(L) \text{ (by Definition 2.9).} \]

(11)

and

\[ r_0 \text{ is the initial state of } \text{schema}(R) \text{ (by Definition 2.9).} \]

(12)

\[ \iff \]

\[ (\ell_0, r_0) \text{ is the initial state of the system } \text{schema}(L) \times \text{schema}(R) \text{ (by (11), (12), and Definition 6.3).} \]

(13)

On the other hand we have:

\[ ((\ell_0, []), (r_0, [])) \text{ is the initial step of } L \times R \text{ (by Definition 6.3)} \]

\[ \iff \]

\[ ((\ell_0, r_0), []) \text{ is the initial state of } L \oplus R \text{ (by Definition 2.12).} \]

\[ \iff \]

\[ (\ell_0, r_0) \text{ is the initial state of } \text{schema}(L \oplus R) \text{ (by Definition 2.9).} \]

(14)

The claim follows from (13) and (14). \(\square\)

7.7 Proof of Theorem 2.2

Proof. Let

\[ \text{schema}(L \oplus R) = (Q_{\text{schema}(L \oplus R)}, \rightarrow_{\text{schema}(L \oplus R)}, s_0) \]

(15)

and

\[ \text{schema}(L) \times \text{schema}(R) = (Q_{\text{schema}(L) \times \text{schema}(R)}, \rightarrow_{\text{schema}(L) \times \text{schema}(R)}, t_0) \]

(16)

Then we have

\[ Q_{\text{schema}(L \oplus R)} = Q_{\text{schema}(L) \times \text{schema}(R)} \text{ (by Proposition 2.1).} \]

(17)

as well as

\[ \rightarrow_{\text{schema}(L \oplus R)} = \rightarrow_{\text{schema}(L) \times \text{schema}(R)} \text{ (by Proposition 2.2).} \]

(18)

and

\[ s_0 = t_0 \text{ (by Proposition 2.3).} \]

(19)

Hence the claim holds. \(\square\)
7.8 Proof of Lemma 2.3

Proof.

By the Definition 6.7, we have to show that the maximal simulation relation from $L \oplus S$ to $L \oplus R$ is initial. To this end, we construct an initial simulation relation. Throughout this proof, we will make use of the following observation: For $i = 1, 2$, let $p_i = (t_i, C_i) \oplus (r_i, E_i)$ be a composite buffered state. If $p_1 = p_2$, then $t_1 = t_2$ and $r_1 = r_2$; that is, the location of $t_1$ is the same as the location of $t_2$. In general, the same does not hold for buffer contents. Furthermore, we conclude from the observation that for any three buffered states $p$, $q_1$ and $q_2$ with $[q_1] = [q_2]$ it holds $[p \oplus q_1] = [p \oplus q_2]$: For atomic propositions that are not buffer equations, the composite states satisfy the respective unions. For the buffered equations, we find that if $q_1$ and $q_2$ satisfy the same buffer equations, then $q_1$ and $q_2$ have the same buffer contents. Therefore, the buffer contents of $p \oplus q_1$ equals the buffer contents $p \oplus q_2$.

By assumption, $R$ simulates $S$. By the Definition 6.7 the maximal simulation relation from $S$ to $R$ is initial. Let $V$ be an initial simulation relation from $S$ to $R$. Let $W = \{(p \oplus x, p \oplus y) | p$ is a state of $L, (x, y) \in V\}$.

We show that $W$ is an initial simulation relation from $L \oplus S$ to $L \oplus R$. By the Definition 6.6, we have to show that (I) $W$ is a simulation relation from $L \oplus S$ to $L \oplus R$, and (II) $W$ is initial.

Part (I): $W$ is a simulation relation. Let $(x, y) \in V$ and $(p \oplus x, p \oplus y) \in W$. Let $p \oplus x \xrightarrow{a} p' \oplus x'$ be a step of $L \oplus S$. By the Definition 6.5, we have to show that (1) $[p \oplus x] = [p \oplus y]$, (2) there exists a step $p \oplus y \xrightarrow{a} p' \oplus y'$, and (3) $(p' \oplus x', p' \oplus y') \in W$.

By the Definition 6.5 and $(x, y) \in V$ we conclude $[x] = [y]$. By our above observation, $[p \oplus x] = [p \oplus y]$ holds. So we have proven part (1). We show part (2) and part (3). By the Definition 2.12 of $L \oplus S$, there exists a step $(p, x) \xrightarrow{a} (p', x')$ in $L \times S$, such that $p \oplus x = p \oplus x$ and $p' \oplus x' = p' \oplus x'$. We observe that the pairs $(p, p), (x, x'), (p', p')$, and $(x', x')$ each consist of two states with the same location. By the Definition 6.3 of $L \times S$, we have the following to cases: (i) $p \xrightarrow{a} p'$ is a step of $L$ and $x = x'$, and (ii) $x \xrightarrow{a} x'$ is a step of $S$ and $p = p'$.

Case (i): $L$ performs the step. Because $x$ and $p$ have the same location, and $L$ is complete, by the Definition 2.8 there exists a step $p \xrightarrow{a} p''$ in $L$, such that $p''$ has the same location as $p$ (and thus also as $p'$), and the buffer contents of $p''$ is the same buffer contents as of $p'$. Thus, $p \xrightarrow{a} p''$ is a step in $L$, by the Definition 6.3 $(p, x) \xrightarrow{a} (p', x)$ is a step of $L \times S$, and by the Definition 2.12 $p \oplus x \xrightarrow{a} p' \oplus x$ is a step of $L \oplus S$, which leads to the conclusion $x' = x$.

Likewise, $(p, y) \xrightarrow{a} (p', y)$ is a step of $L \times R$, and $p \oplus y \xrightarrow{a} p' \oplus y$ is a step of $L \oplus R$. Setting $y' = y$, we have shown part (2). Part (3) directly follows from $(x, y) \in V$ and $p'$ being a state of $L$.

Case (ii): $R$ performs the step. Because $x$ and $x'$ have the same location, and $S$ is complete, by the Definition 2.8 there exists a step $x \xrightarrow{a} x''$ in $S$, such that $x''$ has the same location as $x'$ (and thus also as $x'$), and the buffer contents of $x''$ is the same buffer contents as of $x'$. Thus, $x \xrightarrow{a} x''$ is a step in $S$. Because $V$ is a simulation relation from $S$ to $R$, and $(x, y) \in V$, by the Definition 6.3 there exists a step $y \xrightarrow{a} y'$ and $(x', y') \in V$. 

\[ \text{\hfill \qed} \]
By Definition 6.3, \((p, y) \xrightarrow{a} (p, y')\) is a step of \(L \times R\), and Definition 2.12 \(p \oplus y \xrightarrow{a} p \oplus y'\) is a step of \(L \oplus R\). Setting \(p' = p\), we have shown part (2). Part (3) directly follows from \((x', y') \in V\) and \(p'\) being a state of \(L\).

We conclude that \(W\) is a simulation relation from \(L \oplus S\) to \(L \oplus R\).

**Part (II):** \(W\) is initial. By Definition 6.6, we have to show: Let \(p, x, y\) be the initial states of \(L, S\) and \(R\), respectively. Then, \((p \oplus x, p \oplus y) \in W\). Because \(V\) is initial, \((x, y) \in V\). Then, (II) follows from \(p\) being a state of \(L\).

7.9 Proof of Lemma 2.4

**Proof.**

\(Ad\, (i).\) First, we make the following observation:

**Observation 7.1** 1. Each state of \(F(R)\) is also a state of \(R\) (by Definition 2.7).
2. Each step of \(F(R)\) is also a step of \(R\) (by Definition 2.7).

Consider the relation

\[ V = \{ (p, p) \mid p \in Q_{F(R)} \} \]

(\(*\))

Obviously, \(V \subseteq Q_{F(R)} \times Q_R\) (by Observation 7.1). It remains to be shown that:

(I) \(V\) is a simulation relation

(II) \(V\) is initial

\(Ad\, (I).\) We show the two parts of Definition 6.5

(i) This condition is trivially fulfilled since elements of \(V\) are of the form \((p, p)\) for \(p \in Q_{F(R)}\).

(ii) Let \((p, p) \in V\) and let \(u = p \xrightarrow{a} q\) be a step of \(F(R)\). Then \(u\) is also a step of \(R\) (by Observation 7.1) and \((q, q) \in V\) (by \((*)\)).

Hence, \(V\) is a simulation relation (by Definition 6.5).

\(Ad\, (II).\) We know that \(p^0_{F(R)} = p^0_R\) (by Definition 2.7). Hence \((p^0_{F(R)}, p^0_R) \in V\) (by \((*)\)).

Finally, let \(V_{\text{max}}\) be the maximal simulation relation from \(F(R)\) to \(R\). Then \(V_{\text{max}} \subseteq V\) and is therefore initial. We conclude that \(R\) simulates \(F(R)\) (by Definition 6.7).

\(Ad\, (ii).\) Consider an initial simulation relation \(V\) from \(S\) to \(R\). Let \(W\) be the restriction of \(V\) to pairs \((p, q)\) where \(p\) is a state of \(F(S)\) and \(q\) is a step of \(F(R)\). We show that \(W\) is a simulation relation from \(F(S)\) to \(F(R)\). By Definition 6.5 we have to show for each \((p, q) \in W\) and step \(u = p \xrightarrow{a} p'\) of \(F(S)\): (1) \([p] = [q]\), (2) there exists a step \(q \xrightarrow{a} q'\) in \(F(R)\), and (3) \((p', q') \in W\).

We can conclude part (1) directly from \(W \subseteq V\), \(V\) being a simulation relation and Definition 6.5. Because \((p, q) \in V \subseteq W\), there exists a step \(v =...\)
q \rightarrow q' in R and \((p', q') \in V\). By Definition 6.5, it holds \([p] = [q]\). From that, we may conclude that the buffer contents of \(p\) and \(q\) are equal. Because \(u\) is feasible, \(v\) is feasible as well. Hence, \(v\) is a feasible step of \(R\) starting at a state of \(F(R)\). By Definition 2.7, \(q'\) is a state and \(v\) is a step of \(F(R)\). Hence, part (2) is shown. As \(p'\) and \(q'\) are states of \(F(S)\) and \(F(R)\), respectively, and \((p', q') \in V\), we conclude \((p', q') \in W\), and have shown part (3).

It remains to be shown that \(W\) is initial. By Definition 6.6, we have to show that the respective initial states are in relation \(W\). This follows directly form \(V\) being an initial simulation relation, and the initial states of buffered processes always being the initial states of their feasible parts. √

Ad (iii). This claim follows from (ii). ✓

7.10 Proof of Lemma 3.1

Proof. We construct a maximal element of \(\text{Proc}(L, \phi)\).

Let \(T\) be the maximal, total, deterministic, and tree-shaped buffered process. That is, (1) \(T\) has an initial state \(p\) for each atomic proposition set \(X\) with \([p] = X\), and (2) for each state \(p\) of \(T\), action \(a\), and set \(X\) of atomic propositions, \(p\) has exactly one predecessor or is initial, and there is exactly one step \(p \rightarrow q\) with \([q] = X\). Let each state in the reachable part of \(T\) have a unique location.

We call a path \(\pi\) of \(F(L \oplus T)\) bad if trace(\(\pi\)) is a bad prefix. Otherwise, we call \(\pi\) good. We classify the locations of \(T\) as bad and good based on good and bad paths in \(F(L \oplus T)\): Let \(r\) be a location of \(T\). Then, we call each state \(((\ell, r), C)\) of \(F(L \oplus T)\) an \(r\)-state. Furthermore, we call \(r\) bad if there exists at least one bad path of \(F(L \oplus T)\) resulting in an \(r\)-state. Otherwise, we call \(r\) good. We call a state of \(T\) good or bad based on its location being good or bad. We observe that \(T\) contains at least one good initial state. Otherwise, any initial state would be good according to \(\varphi\), and thus \(\text{Proc}(L, \varphi)\) would be empty, contradicting our assumptions.

Let \(T^*_L\) be the restriction of \(T\) to good states. \(T^*_L\) is again a buffered process, because, as observed above, there remains at least one initial state. We show that \(T^*_L\) is a maximal element of \(\text{Proc}(L, \varphi)\) by showing (1) \(T^*_L \in \text{Proc}(L, \varphi)\), and (2) \(T^*_L\) is maximal in \(\text{Proc}(L, \varphi)\) w.r.t. simulation.

For (1), consider the process \(F(L \oplus T^*_L)\). Every finite path \(\pi\) of \(F(L \oplus T^*_L)\) is also a path of \(F(L \oplus T)\). Let \(\pi\) result in an \(r\)-state. Then, \(r\) is good, because otherwise it would have been removed from \(T\). Thus, the trace of \(\pi\) is not a bad prefix of \(\varphi\). Thus, \(F(L \oplus T^*_L)\) satisfies \(\varphi\) because none of its finite paths is a bad prefix of \(\varphi\).

For (2), consider a process \(R\) not simulated by \(T^*_L\); that is, \(R\) simulates \(T^*_R\), or \(R\) and \(T^*_L\) are incomparable. Because \(T^*_L\) is deterministic, there exists some finite path \(\pi\) of \(F(L \oplus R)\) such that there is no corresponding path of \(F(L \oplus T^*_L)\). Because \(T\) is maximal w.r.t. simulation in every set of processes, we find that there is a path \(\pi'\) corresponding to \(\pi\) in \(F(L \oplus T)\). As \(\pi'\) is a path of \(F(L \oplus T)\) but not of \(F(L \oplus T^*_L)\), \(\pi'\) results in an \(r\)-state and \(r\) is bad. Thus, trace(\(\pi'\)) = trace(\(\pi\)) is a bad prefix of \(\varphi\). Thus, \(R \not\in \text{Proc}(L, \varphi)\). Therefore, \(T^*_L\) simulates all processes in \(\text{Proc}(L, \varphi)\).

Thus we conclude that \(T^*_L\) is a maximal element of \(\text{Proc}(L, \varphi)\). √
7.11 Proof of Theorem 3.2

Proof. We assume the set of bad prefixes of \( \varphi \) given as a deterministic finite automaton \( A \).

Let \( T \) and \( T_\varphi^L \) be defined as in the proof of Lemma 3.1. For a path \( \pi \) of \( F(L \oplus T) \) let \( A(\pi) \) be the state of \( A \) results reading \( \text{trace}(\pi) \). Then, a location \( r \) of \( T \) is bad, if there exists a path \( \pi \) of \( F(L \oplus T) \) such that \( \pi \) results in \(( (\ell, r), C ) \) and \( A(\pi) \) is a final state of \( A \). Therefore, for all paths \( \pi \) of \( F(L \oplus T_\varphi^L) \), we conclude \( A(\pi) \) is not a final state of \( A \).

We inductively define the transition system \( (F(L \oplus T_\varphi^L), A) \):

1. Let \( p \) be an initial state of \( F(L \oplus T_\varphi^L) \). Then, \(( p, A(\varepsilon) ) \) is an initial state of \( (F(L \oplus T_\varphi^L), A) \).
2. Let \( p, q \) be a state of \( F(L \oplus T_\varphi^L), A) \). Let \( p \xrightarrow{a} p' \) be a step of \( F(L \oplus T_\varphi^L) \). Let \( q' \) be the state of \( A \) after reading \([p']\) in state \( q \). Then, \((p', q')\) is a state and \((p, q) \xrightarrow{a} (p', q') \) is a step of \( (F(L \oplus T_\varphi^L), A) \).

Then, for each path \( \pi \) of \( F(L \oplus T_\varphi^L) \), there exists a corresponding path \( \pi' \) in \( (F(L \oplus T_\varphi^L), A) \) and vice versa. Furthermore, \( \pi \) results in \( p \) iff \( \pi' \) results in \((p, A(\pi)) \).

Therefore, for all states \((( (\ell, r), C ), q ) \) of \( (F(L \oplus T_\varphi^L), A) \), \( q \) is not a final state of \( A \). Because \( \varphi \) is \( k \)-bounded, we can further conclude \( C(b) \leq k \) for each buffer \( b \).

However, \( (F(L \oplus T_\varphi^L), A) \) is still infinite, because there are infinitely many locations of \( T_\varphi^L \). For \( i = 1, 2 \), let \( \pi_i \) result in state \((( (\ell, r_i), C ), q ) \) in \( (F(L \oplus T_\varphi^L), A) \); that is, the two resulting states are equal up to the location of \( T_\varphi^L \). Let \( \pi_1 \pi_1^{-1} \) be a path of \( T_\varphi^L \) resulting in \((( (\ell', r'_1), C'), q' ) \). Then, there exists \( \pi_2 \), such that \( \pi_2 \pi_2^{-1} \) is a path of \( T_\varphi^L \) resulting in \((( (\ell', r'_2), C'), q' ) \). That is, the two resulting states again only differ in the location of \( T_\varphi^L \). Let \( \kappa(r) \) denote the set of all \(( (\ell, C), q ) \) such that \((( (\ell, r), C ), q ) \) is a state of \( (F(L \oplus T_\varphi^L), A) \). Then, we call \( r_1 \) and \( r_2 \) equivalent if \( \kappa(r_1) = \kappa(r_2) \).

We write \([r]_\kappa\) for the equivalence class of \( r \).

Now, we define the transition system \( G_\varphi^L \) inductively as follows:

1. Let \( ([r]) \) be an initial state of \( T_\varphi^L \). Then, \( ([r], [r]) \) is an initial state of \( G_\varphi^L \).
2. Let \( ([r], C) \) be a state of \( G_\varphi^L \) and \((r, C) \xrightarrow{a} (r', C') \) be a step of \( T_\varphi^L \). Then, \( ([r'], C') \) is a state and \( ([r], C) \xrightarrow{a} ([r'], C') \) is a step of \( G_\varphi^L \).

We observe that schema\( (G_\varphi^L) \) is finite, and \( G_\varphi^L \) and \( T_\varphi^L \) are similar. Thus, \( G_\varphi^L \) is a maximal element of \( \text{Proc}(L, \varphi) \) with a finite schema. \( \square \)

References


23. Richard Müller, Christian Stahl, WilM.F. Aalst, and Michael Westergaard. Service discovery from observed behavior while guaranteeing deadlock freedom in collaborations.


