AVIS'04 Preliminary Version

Verification of Asynchronous Systems with Unbounded and Unordered Message Buffers

Prasanna Thati, and Mahesh Viswanathan

Department of Computer Science, UIUC, Urbana-Champaign, USA

Abstract

We present algorithms for verifying safety and liveness properties of a class of systems, executing and communicating asynchronously. These systems will be modeled by a variant of finite state machines with unbounded and unordered message buffers, and hence will have an infinite state space. We present algorithms for deciding two weak preorder relations over such systems, namely language containment and generalized divergence language containment. This is in contrast to previous results that either decide equivalences between infinite state systems [25,7,29] or preorders between an infinite state system and a *finite* state system [2,1,6,13]. We also establish EXPSPACE lower bounds for the verification problems we investigate, and we show that our algorithms can be applied to decide the may testing equivalence on such systems.

Key words: asynchrony, verification, safety, liveness, may testing.

1 Introduction

Asynchrony is a common feature of distributed systems, where not only the execution of different system components but also the communication between them is asynchronous. Specifically, there is no assumption about the relative speed of execution of different system components, and messages can be buffered for arbitrarily long periods of time and be delivered to their target in arbitrary order. The state of such a system includes a message buffer containing the undelivered messages. Since there is no assumption about the order of message deliveries, this buffer is *unordered*. Further, since messages are subject to arbitrary delays, the message buffer can be of *unbounded* size. Note that this implies that such systems can have an *infinite* state space. The systems we consider are in addition *open*, i.e. they can asynchronously exchange messages with their environment.

We focus on verifying the class of asynchronous systems that can be modeled as a (finite) collection of asynchronously communicating finite state machines (FSMs). The FSMs send and receive messages from a shared message buffer which is a multiset of undelivered messages. The buffer is also allowed to input and output messages to the environment. Further, there is no assumption about the relative speed of execution of different FSMs. Such a system can be equivalently seen as consisting of a single FSM with a message buffer (that is open to interactions with the environment); the single FSM can be thought of as the product of several FSMs in the natural way. We call such a system as an asynchronous finite state machine (AFSM).

Our model of AFSMs is interesting for several reasons. First, while FSMs are exactly the processes expressible in regular CCS [23], AFSMs are the processes that can be expressed in the asynchronous extension of regular CCS [8]. Secondly, systems with unordered message buffers such as AFSMs can serve as convenient abstractions for verification of systems with ordered message deliveries [6]. For example, AFSMs can be used as abstractions for verifying properties of (lossy) FIFO-channel systems [3]. Finally, the simplicity of the AFSMs turns out to be useful for identifying the decidability boundaries for the problems we are interested in. We will see that enriching the model in even simple ways, leads to undecidability.

In this paper, we present algorithms to solve two problems. The first algorithm decides the language containment problem for AFSMs, which can be used to reason about safety properties [4]. Second, we present an algorithm to decide containment of the v-liveness language of AFSMs. The v-liveness language of an AFSM is the set of all traces after exhibiting which the AFSM can perform v repeatedly. Checking v-liveness language containment of AFSMs can be used to verify special kinds of liveness properties [21]. Although we do not have precise upper bounds on the running time or space requirements of these algorithms, we prove lower bounds for the resource requirements of these problems, demonstrating the computational difficulty of these problems.

We show that our algorithm for language containment can be used to decide the *may testing* equivalence between AFSMs. The may testing equivalence is a notion of process equivalence that is known to be useful for reasoning about safety properties [26]. So far, decision procedures for may testing were known for only the simple class of FSMs [19]; our results provide the first decision procedure over an interesting class of asynchronous infinite state systems. As one may expect, deciding may equivalence over AFSMs is computationally more complex than deciding it over FSMs. We also consider a generalized version of may equivalence which incorporates certain encapsulation mechanisms that constrain the interactions between a system and its environment. We show that the generalized equivalence is undecidable over AFSMs, but is decidable over FSMs.

Related Work: Many computational models for asynchrony have been studied before. The most popular ones include Basic Parallel Processes (BPP) [9], Multiset Automata (MSA) [7], Petri nets [27] and Vector Addition Systems [27]. BPPs are a special class of MSAs, which in turn are restrictions of Petri nets. Vector Addition Systems and Petri nets are equivalent in expressive power. The model of AFSMs, that we consider here, can be most directly seen as a special kind of MSA with τ-transitions (or ϵ-transitions), where the labels on the transitions are restricted in a particular way; more details can be found in Section 2. This restriction turns out to be crucial for decidability, because the problems that we consider [1] are known to be undecidable for BPPs (and hence for MSAs and Petri nets).

Decision procedures for verification problems such as reachability, bisimilarity and language containment are known for a general class of infinite state systems called the well-structured transition systems [1,13]. Since AFSMs are well structured transition systems these decision procedures apply to them as well. But these results do not subsume the problems we address in this paper. Specifically, the algorithms in [1,13] such as those for simulation and language containment only deal with comparing an infinite state system with a finite state system. For instance, there is no known procedure for deciding language containment between two arbitrary (possibly infinite) well-structured transition systems. In comparison, we address the problems of language containment and v-liveness language containment between two infinite state AFSMs.

AFSMs also relate to lossy channel systems investigated in [2,3,6]. Specifically, they can be viewed as finite control systems interacting with an unreliable (or noisy) buffer, where messages can be randomly lost to or received from the environment. But unlike in typical lossy channel systems where the message losses are invisible, message losses and additions are the only visible actions in AFSMs. The idea is that these transitions are viewed as interactions between the process and its environment, and we are only interested in the observable behavior of such open systems.

Lossy channel systems such as those in [2,6] are instances of well-structured transition systems, and even for these special instances there are no results for comparing two infinite state systems in the sense mentioned above. For example, [2] gives a decision procedure for only checking the language containment between a lossy channel system and a finite state system and [6] only considers model checking with respect to sub-logics of μ-calculus. Note that AFSMs can be used to specify properties not expressible in μ-calculus (see Section 2).

Overview: The technique we use for deciding language containment over AFSMs relates to the tree saturation method described in [13] for well structured transition systems. But in addition, since AFSMs are a special subclass of Petri nets, we are able to exploit Karp-Miller's algorithm for constructing coverability trees which is crucial for decision procedure. Karp-Miller's algorithm is not applicable to arbitrary well-structured transition systems [2]. We solve the v-liveness language-containment problem by reducing it to the language containment problem. The key idea in this reduction is to relate

[1] Bisimulation is known to be decidable for BPPs but not for the other models [24,9]

[2] General conditions on transition systems for which the Karp-Miller algorithm works are presented in [12,10]

the v-liveness at a state to the size of the message buffer in that state. We establish this relationship by generalizing Rackoff's lemma for unlabeled Petri nets (that gives an upperbound on the depth of coverability trees) to AFSMs that can be seen as a special class of Petri nets with labeled noisy transitions.

Following is the layout of the rest of the paper. In Section 2, we formally define AFSMs and prove some properties which will be useful in later sections. In Section 3, we present the algorithm for language containment problem, and in Section 4 we present the algorithm for v-liveness language containment. Section 5 applies our algorithms to decide may equivalence over AFSMs. Finally, in Section 6 we conclude with a few comments on possible directions for further work. Most of the proofs have been omitted due to space constraints, but can all be found in [31].

2 Asynchronous Finite State Machines

We assume disjoint infinite sets of names $\mathcal{N}$ and co-names $\overline{\mathcal{N}}$, and a bijection $\bar{\cdot} : \mathcal{N} \rightarrow \overline{\mathcal{N}}$. We let $\sum$ range over finite sets of names, and write $\overline{\sum}$ to denote the set $\{\overline{a} \mid a \in \sum\}$. Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ be an FSM with τ-moves. Specifically, Q is the finite set of states, $\sum \cup \overline{\sum}$ the finite alphabet set, $\rightarrow \subseteq Q \times (\sum \cup \overline{\sum} \cup \{\tau\}) \times Q$ the transition relation, q_0 the start state, and $F \subseteq Q$ the set of final states. We let p, q range over Q, and a, b, c over $\sum$. We call $\sum \cup \overline{\sum} \cup \{\tau\}$ the set of actions, and let α range over it. The actions in $\sum$ are called *input* actions, the actions in $\overline{\sum}$ the *output* actions, and τ the *internal* action. We write $\hat{\alpha}$ to mean α if $\alpha \neq \tau$, and ϵ otherwise. The set $\sum \cup \overline{\sum}$ is the set of visible actions, and we let β range over it.

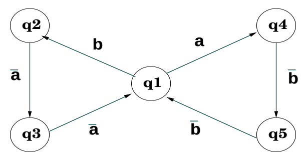
We write $p \xrightarrow{\alpha} q$ instead of $(p, \alpha, q) \in \rightarrow$, $p \Longrightarrow q$ if $p \xrightarrow{\tau}{}^{*} q$, and $p \overset{\alpha}{\Longrightarrow} q$ if $p \Longrightarrow \xrightarrow{\alpha} \Longrightarrow q$. We call $(\sum \cup \overline{\sum})^{*}$ the set of traces, and let r, s, t range over it. For $s = \epsilon$ we write $p \xrightarrow{s} q$ if $p = q$, and $p \overset{s}{\Longrightarrow} q$ if $p \Longrightarrow q$. For $s = \beta.s'$ we write $p \xrightarrow{s} q$ if $p \xrightarrow{\beta}\xrightarrow{s'} q$, and $p \overset{s}{\Longrightarrow} q$ if $p \overset{\beta}{\Longrightarrow}\overset{s'}{\Longrightarrow} q$. We define $L(p) = \{s \mid p \overset{s}{\Longrightarrow} q,\ q \in F\}$, and $L(M) = L(q_0)$. For a set S, we write $\mathcal{P}(S)$ to denote the powerset of S, and $\{|S|\}$ to denote the set of all (possibly infinite) multisets of S. We let B range over $\{|\sum|\}$.

Definition 2.1 [AFSM] An AFSM is an FSM with a message buffer that is open to asynchronous interactions with the environment. Specifically, the set of configurations (states) of an AFSM with an underlying FSM $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ is $Q \times \{|\sum|\}$, its initial configuration is $(q_0, \{\})$, its final configurations are $F \times \{|\sum|\}$, and its (asynchronous) transition relation $\longrightarrow_A \subseteq (Q \times \{|\sum|\}) \times (\sum \cup \overline{\sum} \cup \tau) \times (Q \times \{|\sum|\})$ is defined by the following rules

(i) $(q, B) \xrightarrow{a}_A (q, B \cup \{a\})$

(ii) $(q, B) \xrightarrow{\overline{a}}_A (q, B \setminus \{a\})$ if $a \in B$

(iii) $(q, B) \xrightarrow{\tau}_A (q', B')$ if any of the following is true



$$\begin{aligned}(q_1, \{\}) &\xrightarrow{a}_A (q_1, \{a\}) \xrightarrow{\tau}_A (q_4, \{\})\\ &\xrightarrow{\tau}_A (q_5, \{b\}) \xrightarrow{\tau}_A (q_1, \{b, b\})\\ &\xrightarrow{\bar{b}}_A (q_1, \{b\}) \xrightarrow{\tau}_A (q_2, \{\})\\ &\xrightarrow{\tau}_A (q_3, \{a\}) \xrightarrow{\tau}_A (q_1, \{a, a\})\end{aligned}$$

Fig. 1. An asynchronous transition sequence of an example AFSM.

(a) $q \xrightarrow{\tau} q'$, $B' = B$
(b) $q \xrightarrow{a} q'$, $a \in B$ and $B' = B \setminus \{a\}$
(c) $q \xrightarrow{\bar{a}} q'$, $B' = B \cup \{a\}$

The binary relations $\xrightarrow{s}_A, \Longrightarrow_A, \overset{s}{\Longrightarrow}_A$ on configurations are defined as expected. We define the language of a configuration as $L_A(q, B) = \{s \mid (q, B) \overset{s}{\Longrightarrow}_A (q', B'), q' \in F\}$. We write $L_A(q)$ as a shorthand for $L_A(q, \emptyset)$. An AFSM with the underlying FSM M will also be denoted by M, but it should be clear from the context as to what is being referred to. We define the asynchronous language of an AFSM M as $L_A(M) = L_A(q_0)$.

Thus, an AFSM's state is composed of the control state of the underlying FSM and a message buffer. The buffer contains inputs received from the environment and outputs produced by the underlying FSM, which have not yet been consumed. The first two transition rules of Definition 2.1 are for asynchronous exchange of messages between the buffer and the environment. Note that the control state does not change in these transitions. Rule 3 is for internal transitions where the underlying FSM sends or receives messages from the buffer. Note that only the transitions that involve interactions with the environment are labeled with visible actions. Figure 1 illustrates a transition sequence of an example AFSM. The terminology – AFSM – may be a bit misleading because the set of states of an AFSM is infinite. For the AFSM of Figure 1 we can show that

$$(q_1, \{\}) \overset{a}{\Longrightarrow}_A (q_1, \{a^n, b^m\}) \qquad \text{if } n, m \geq 0 \text{ and } n \equiv m + 1 \pmod 3$$

An alternate way of defining the acceptance condition would be to consider the final buffer state in addition to the control state. Specifically, the set of final states F could be a subset of $Q \times \{| \sum |\}$ and

$$L_A(M) = \{s \mid (q_0, \emptyset) \overset{s}{\Longrightarrow}_A (q, B),\ \text{for some } (q, B') \in F \text{ such that } B' \subseteq B\}$$

Such an acceptance condition is familiar in the Petri net literature and is known to be equivalent to the acceptance condition used in Definition 2.1.

Following is some notation and simple facts that will be useful later on. Let $\#(a, B)$ denote the number of times a occurs in the multiset B. For a sequence of multisets B_i, we define $\sqcup_i B_i$ as the multiset which satisfies for all a, $\#(a, \sqcup_i B_i) = \max_i \#(a, B_i)$.

Lemma 2.2 *(1) If $B \subseteq B'$ then $L_A(q, B) \subseteq L_A(q, B')$. (2) If $B_1 \subseteq B_2 \subseteq \ldots$ and $B = \sqcup_i B_i$, then $L_A(q, B) = \cup_i L_A(q, B_i)$.* □

Now, we establish the relationship between $L(M)$ and $L_A(M)$; namely that $L_A(M)$ is the smallest set that contains $L(M)$ and that is closed under the relation $\rhd$ defined below. This characterization will be very useful in later sections.

Definition 2.3 For a set of names $\sum$, let $\rhd$ be the smallest reflexive transitive relation on $(\sum \cup \overline{\sum})^*$ that is closed under the following rules

1. $s_1.s_2 \ \rhd \ s_1.a.s_2$	2. $s_1.\beta.a.s_2 \ \rhd \ s_1.a.\beta.s_2$
3. $s_1.s_2 \ \rhd \ s_1.a.\overline{a}.s_2$	4. $s_1.\overline{a}.s_2 \ \rhd \ s_1.s_2$
5. $s_1.\overline{a}.\beta.s_2 \ \rhd \ s_1.\beta.\overline{a}.s_2$	6. $s_1.\overline{a}.a.s_2 \ \rhd \ s_1.s_2$

We lift $\rhd$ to sets of traces as $R \rhd S$ if for every $s \in S$ there is $r \in R$ such that $r \rhd s$. We define the closure of S under the relation $\rhd$, denoted $[S]_\rhd$, as the smallest set that contains S and that is closed under $\rhd$.

Strictly speaking, in Definition 2.3, we have defined a family of relations indexed by the set $\sum$, and hence $\rhd$ has to be annotated with $\sum$. But to keep the notation simple, we ignore this detail, and instead ensure that $\sum$ is clear from context. The six rules above succinctly capture the asynchrony in message exchanges between an AFSM and its environment; the idea being that if $s \ \rhd \ r$ and $s \in L_A(M)$, then $r \in L_A(M)$. Rule 1 captures the fact that an AFSM is always input enabled, while rule 2 says that an AFSM can perform inputs in any order. Rule 3 states that an input followed by a complementary output can always be performed; the input received can be buffered and output back to the environment in the next step. Rules 4-6 are duals of the first 3 rules. Rule 5 states that outputs can be buffered and emitted to the environment later, while rule 4 accounts for the case where an output is buffered and not yet emitted. A buffered output can also be internally consumed instead of being emitted, and this is reflected in rule 6.

Theorem 2.4 *For an FSM M, $L_A(M) = [L(M)]_\rhd$.*

Remark: Thus, $L(M) \subseteq L_A(M)$, and $L_A(M)$ is closed under $\rhd$.

AFSMs are a special class of MSAs with τ-transitions [7]. The central difference between an MSA and an AFSM is that in an AFSM the labels on the transitions are intimately linked to the operations on the multiset buffer. The labels on the transitions of an AFSM, uniquely determine the changes to the multiset buffer. Due to this important restriction, the verification problems we consider in Section 3 and 4 are decidable, quite unlike the case of MSAs for which they are known to be undecidable.

Note that using AFSMs as a specification language, we can express properties that are not regular and hence not expressible in modal logics like μ-calculus and LTL. One such example is the language of an AFSM with a singleton alphabet, one state which is both an initial and a final state, and no transitions. The language of this machine is the set of all traces in which every prefix has at least as many inputs as outputs, which is not regular. Thus results on model checking of infinite state systems with respect to sub-logics of μ-calculus [6,7] do not subsume the results presented here.

3 Verifying Safety Properties

In this section, we will present an algorithm that allows us to verify safety properties of AFSMs. A safety property can be interpreted as a prefix-closed set of traces S [4]. We are interested in safety properties S which can be represented by an AFSM M_2 such that $S = L_A(M_2)$. An AFSM M_1 is said to satisfy a safety property S if $L_A(M_1) \subseteq S$. Thus, verifying if M_1 satisfies a safety property represented by M_2, corresponds to deciding if $L_A(M_1) \subseteq L_A(M_2)$. Note that, since S is prefix closed, every control state M_2 is also a final state. But in the following, we will present a decision procedure for the more general problem of deciding $L_A(M_1) \subseteq L_A(M_2)$ for arbitrary M_2. The reader may note that the language containment problem over the more general class of MSA (with τ-transitions) is undecidable [16].

Note that deciding $L_A(M_1) \subseteq L_A(M_2)$ involves comparing two infinite state systems. The following lemma, which is an easy consequence of Theorem 2.4, provides a handle to deal with this problem.

Lemma 3.1 *Let M_1 and M_2 have alphabet $\sum_1$ and $\sum_2$ respectively, and let $\sum_1 \subseteq \sum_2$. Then $L_A(M_1) \subseteq L_A(M_2)$ if and only if $L(M_1) \subseteq L_A(M_2)$.* $\Box$

For the case $\sum_1 \not\subseteq \sum_2$, it is easy to show that $L_A(M_1) \subseteq L_A(M_2)$ if and only if $L_A(M_1) = \emptyset$. Now, checking for emptiness of $L_A(M_1) \neq \emptyset$ is the same as checking for emptiness of $L(M_1)$, and decision procedures for this are well known. So, from now on we may assume that $\sum_1 \subseteq \sum_2$.

As a consequence of Lemma 3.1, in order to decide $L_A(M_1) \subseteq L_A(M_2)$, we only need to compare the (synchronous) transitions of a finite state system with (asynchronous) transitions of an infinite state system. Figure 2 shows a naive attempt at a decision procedure that exploits this simplification. The arguments to procedure contained are a control state p of $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$ and a set of configurations C of $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. The idea is that the procedure returns true if and only if $L(p) \subseteq L_A(C)$, where $L_A(C) = \cup_{(q,B) \in C} L_A(q, B)$. Thus, to decide if $L_A(M_1) \subseteq L_A(M_2)$ the procedure is to be invoked with arguments $(q_1, M_1, \{(q_2, \emptyset)\}, M_2)$.

The procedure contained recursively matches the synchronous transitions of M_1 starting from p, with asynchronous transitions of M_2 starting from any configuration in C. Without loss of generality, we assume that M_1 does not have any τ transitions between its control states, because otherwise we can eliminate the τ actions by the usual τ-elimination procedure without changing $L(M_1)$. In line 8, we assume a subroutine reach such that for a trace s, $\mathsf{reach}(C, s, M_2) = \cup_{(p_2,B) \in C} \{(p_2', B') \mid (p_2, B) \overset{s}{\Longrightarrow}_A (p_2', B')\}$.

Figure 2 does not provide a decision procedure since the procedure contained need not terminate due to two reasons. First, the recursion in lines 6

```
1   contained($p, M_1, C, M_2$)
2       if $p \in F_1$ and $\epsilon \notin L_A(C)$ then return false
3       for all $a \in \sum_1$, $p' \in Q_1$
4           if $p \xrightarrow{a}_1 p'$ then
5               $C' := \{(p_2, B \cup \{a\}) \mid (p_2, B) \in C\}$
6               if not contained($p', M_1, C', M_2$) then return false
7           if $p \xrightarrow{\overline{a}}_1 p'$ then
8               $C' :=$ reach($C, \overline{a}, M_2$)
9               if not contained($p', M_1, C', M_2,$) then return false
10      end for
11      return true
12  end contained
```

Fig. 2. A naive attempt at deciding the asynchronous language containment problem

and 9 is in general unbounded. Second, the set $\mathsf{reach}(C, \overline{a}, M)$ may not be finite. For instance for the AFSM M of Figure 1, we have

$$\mathsf{reach}(\{(q_1, \{a\})\}, \overline{a}, M) = \left\{ \begin{array}{ll|l} (q_1, \{a^n, b^m\}) & (q_2, \{a^{n+1}, b^m\}) & n, m \geq 0 \\ (q_3, \{a^{n+2}, b^m\}) & (q_4, \{a^n, b^{m+1}\}) & \\ (q_5, \{a^n, b^{m+2}\}) & & n \equiv m \pmod 3 \end{array} \right\}$$

Finally, we also have to provide a procedure to check if $\epsilon \in L_A(C)$ in line 2.

We use the following idea to bound the number of recursive calls. We define $C' \preceq C$ if for every $(q, B') \in C'$ there is $(q, B) \in C$ such that $B' \subseteq B$. Note that as a consequence of Lemma 2.2.1, $C' \preceq C$ implies $L_A(C') \subseteq L_A(C)$. But this implies that an invocation $\mathsf{contained}(p, M_1, C, M_2)$ is redundant if there was a previous invocation $\mathsf{contained}(p, M_1, C', M_2)$ such that $C' \preceq C$. This is because, if the previous invocation $\mathsf{contained}(p, M_1, C', M_2)$ returned true, then $L(p) \subseteq L_A(C') \subseteq L_A(C)$, and hence we know that $\mathsf{contained}(p, M_1, C, M_2)$ should return true. On the other hand, if $\mathsf{contained}(q, M_1, C', M_2)$ returned false, then the procedure would already have terminated by returning false.

The following lemma states a useful property of the relation $\preceq$.

Lemma 3.2 *Given a sequence $C_1, C_2, \ldots,$ where $C_i \in \mathcal{P}(Q \times \{|\sum|\})$ are finite sets, there exist m, n such that $m < n$ and $C_m \preceq C_n$.* □

Note that finiteness of C_i does not preclude C_i from containing configurations (q, B) where B is infinite.

To avoid computation of the possibly infinite set $\mathsf{reach}\ (C, \overline{a}, M)$, we compute a finite set of configurations C' such that $L_A(C') = L_A(\mathsf{reach}(C, \overline{a}, M))$, and use C' instead in line 8 of Figure 2.

Definition 3.3 For sets of configurations C_1 and C_2, we say C_2 *covers* C_1 if (1) $C_1 \preceq C_2$, and (2) $(q, B) \in C_2$ implies there are $B_1 \subseteq B_2 \subseteq \ldots$ such that $B = \sqcup_i B_i$ and $(q, B_i) \in C_1$.

For instance, for the AFSM in Figure 1 and the set $C' = \{ (q_i, \{a^\omega, b^\omega\}) \mid 1 \leq i \leq 5 \}$, we have C' *covers* $\mathsf{reach}(\{(q_1, \{a\})\}, \overline{a}, M)$. We write $a^\omega \in B$ to denote that B contains infinitely many a's, and adapt the usual multiset operations and relations accordingly. For instance, $\{|a, b^\omega|\} \cup \{|a, b|\} = \{|a, a, b^\omega|\}$, and $\{|a, b^\omega|\} \setminus \{|a, b|\} = \{|b^\omega|\}$. The following lemma is an easy consequence of Lemma 2.2.

Lemma 3.4 *If C_2 covers C_1 then $L_A(C_1) = L_A(C_2)$.* □

Our plan is to compute a finite set of configurations C' such that C' *covers* $\mathsf{reach}(C, \overline{a}, M)$. We first consider the case where the set C contains a single configuration. We use Karp and Miller's algorithm for computing the coverability tree of Petri Nets [20], which applies to AFSMs since they are a special class of Petri nets. This is the subroutine cover shown in Figure A.1 in the appendix. We recall from [20] that the procedure cover terminates for any input $((q, B), M)$, and returns a *finite* set of configurations such that $\mathsf{cover}(\{(q, B)\}, M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$. For a given $\overline{a}$, we then extract a set C' from $\mathsf{cover}(\{(q, B)\}, M)$ such that C' *covers* $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$.

Lemma 3.5 *For $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$, $q \in Q$, and $B \in \{| \sum |\}$, the following statements are true.*

(i) *$\epsilon \in L_A(q, B)$ if and only if $(q', B') \in \mathsf{cover}((q, B), M)$ for some $q' \in F$.*

(ii) *For a given $a \in \sum$, let $C = \{(q', B' \setminus \{a\}) \mid (q', B') \in \mathsf{cover}((q, B), M),\ a \in B'\}$. Then*
C covers $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$. □

We are now ready to present the correct version of the procedure $\mathsf{contained}$.

Theorem 3.6 *There is an algorithm, which given M_1 and M_2, decides if $L_A(M_1) \subseteq L_A(M_2)$.*

Proof. Figure 3 shows the algorithm, which differs from the procedure in Figure 2 as follows.

In line 4, instead of checking if $\epsilon \in L_A(C)$, we use Lemma 3.5.1 and check for the equivalent condition that for some $p_2 \in F_2$ and B, $(p_2, B) \in C''$ where $C'' = \cup_{(p,B) \in C} \mathsf{cover}((p, B), M_2)$. In line 11, we exploit Lemmas 3.5.2 and 3.4 to use the set $\{(p, B \setminus \{a\}) | (p, B) \in C'', a \in B\}$ which is always finite (because the output of cover is finite), instead of $\mathsf{reach}(C, \overline{a}, M)$ which can in general be infinite.

To ensure termination, we use the variable L to remember all the inputs with which $\mathsf{contained}$ has been invoked so far, and we recursively call $\mathsf{contained}$ with input (p_1', C') (lines 9 and 13) only if it is not redundant. The variable L is initially set to $\emptyset$. We say (q, C) is *covered by* L if there is $(q, C') \in L$ such

```
contained($p_1, M_1, C, M_2$)
    $L := L \cup (p_1, C)$
    $C'' := \cup_{(p,B) \in C}$cover($(p, B), M_2$)
    if $p_1 \in F_1$ and for all $(p_2, B) \in C''$ $p_2 \notin F_2$ then return false
    for all $a \in \sum_1$, $p'_1 \in Q_1$
        if $p_1 \xrightarrow{a}_1 p'_1$ then
            $C' := \{(p_2, B \cup \{a\}) \mid (p_2, B) \in C\}$
            if $(p'_1, C')$ not covered by $L$ then
                if not contained($p'_1, M_1, C', M_2$) then return false
        if $p_1 \xrightarrow{\overline{a}}_1 p'_1$ then
            $C' := \{(p, B \setminus \{a\}) | (p, B) \in C'', a \in B\}$
            if $(p'_1, C')$ not covered by $L$ then
                if not contained($p'_1, M_1, C', M_2$) then return false
    end for
    return true
end contained
```

Fig. 3. An algorithm for deciding asynchronous language containment of AFSMs.

that $C' \preceq C$. Thus, an input (p'_1, C') is redundant if and only if it is covered by L.

We now show that contained terminates for an input (p, C) provided C is a finite set. The proof is by contradiction. Suppose contained doesn't terminate for an input (p, C), where C is a finite set. Then contained is called an infinite number of times with arguments, say (p_i, C_i), each of which is added to L. The sequence (p_i, C_i) has a subsequence (p_k, C_{k_i}) for some k, since $|Q_1|$ is finite. Since C is finite it follows that each C_{k_i} is finite. Then by Lemma 3.2, there are m, n such that $m < n$ and $C_{k_m} \preceq C_{k_n}$. But this is impossible because when contained is called with arguments (p_k, C_{k_n}), the argument is already covered by L. Contradiction. □ □

Note that, given M_1, M_2 the above algorithm only decides if $L_A(q_1, \emptyset) \subseteq L_A(q_2, \emptyset)$. We can decide if $L_A(q_1, B_1) \subseteq L_A(q_2, B_2)$ for arbitrary B_1, B_2 as follows. Consider an FSM M'_1 (resp. M'_2) that first performs as many output transitions as messages in B_1 (resp. B_2) and then behaves like M_1 (resp. M_2). Clearly $L_A(M'_i) = L_A(q_i, B_i)$, and hence $L_A(q_1, B_1) \subseteq L_A(q_1, B_2)$ if and only if $L_A(M'_1) \subseteq L_A(M'_2)$.

We now contrast our algorithm with the usual procedure for deciding synchronous language containment over FSMs. To decide $L(M_1) \subseteq L(M_2)$ the usual procedure is to first construct $\neg M_2$ such that $L(\neg M_2) = \overline{L(M_2)}$, then construct $M_1 \cap \neg M_2$ such that $L(M_1 \cap \neg M_2) = L(M_1) \cap L(\neg M_2)$, and finally check if $L(M_1 \cap \neg M_2) = \emptyset$. This procedure cannot be used for deciding $L_A(M_1) \subseteq L_A(M_2)$ because we cannot in general construct $\neg M_2$, i.e. the set of asynchronous languages of FSMs is not closed under complementation.

Although we do not have a clear upper bound on the running time of the algorithm `contained`, we know that the asynchronous language containment problem is EXPSPACE-hard.

Theorem 3.7 *The asynchronous language containment problem for FSMs is EXPSPACE-hard.* □

The reader may contrast Theorem 3.7 with the fact that deciding if $L(M_1) \subseteq L(M_2)$ is only PSPACE-complete.

4 Verifying Liveness Properties

In this section, we will present an algorithm that allows us to verify a special class of liveness properties of AFSMs. Liveness properties [21] guarantee that something "good" will eventually happen and are identified with sets that can be expressed as a countable intersection of sets of the form $U\Sigma^\omega$, where $U \subseteq \Sigma^*$ and Σ^ω is the collection of infinitely long strings over Σ. We will investigate verifying a special class of liveness properties, where the property is a countable intersection of the sets $U_n\Sigma^\omega$, where $U_n = Uv^n$ for some string v over Σ and $U \subseteq \Sigma^*$. If the liveness property is represented by an AFSM M_2, then verifying M_1 with respect to M_2 requires checking if

$$\{uv^\omega \mid (q_1, \emptyset) \overset{uv^\omega}{\Longrightarrow}_A\} \subseteq \{uv^\omega \mid (q_2, \emptyset) \overset{uv^\omega}{\Longrightarrow}_A\}$$

where q_1 and q_2 are the initial states of the underlying FSMs M_1 and M_2 respectively, and v is some fixed string.

Definition 4.1 [liveness language] For a sequence $v \in (\sum \cup \overline{\sum})^*$, we say v is (asynchronously) live at a configuration (q_1, B_1), written $(q_1, B_1) \uparrow^v_A$, if $(q_1, B_1) \overset{v^\omega}{\Longrightarrow}_A$, i.e. there is an infinite sequence of transitions

$$(q_1, B_1) \overset{v}{\Longrightarrow}_A (q_2, B_2) \overset{v}{\Longrightarrow}_A (q_3, B_3) \overset{v}{\Longrightarrow}_A \ldots$$

We also require that if $v = \epsilon$ then for every i the computation $(q_i, B_i) \overset{v}{\Longrightarrow}_A (q_{i+1}, B_{i+1})$ involves atleast one transition step. For $u \in (\sum \cup \overline{\sum})^*$, we say v is live at (q, B) after u, written $(q, B) \uparrow^v_A u$, if $(q, B) \overset{u}{\Longrightarrow}_A (q', B')$ for some (q', B') such that $(q', B') \uparrow^v_A$. We write $q \uparrow^v_A$ as a shorthand for $(q, \emptyset) \uparrow^v_A$, and similarly for $q \uparrow^v_A u$. We define the asynchronous v-liveness language $L^{\uparrow v}_A(M)$ of M as $L^{\uparrow v}_A(M) = \{u \mid u \in (\sum \cup \overline{\sum})^*,\ q_0 \uparrow^v_A u\}$.

Intuitively, $(q, B) \uparrow^v_A u$ means that the configuration (q, B), after exhibiting u, *may* reach a state at which v is "live" and so it can exhibit v^ω. Observe that, if $v = \epsilon$, $(q, B) \uparrow^\epsilon_A u$ means that (q, B) may diverge after exhibiting u. Thus, checking if a configuration diverges is a special case of checking if a sequence v is live at it.

Now, we present a decision procedure which given M_1, M_2, and v, checks if $L^{\uparrow v}_A(M_2) \subseteq L^{\uparrow v}_A(M_1)$. Our approach is to reduce this problem to the asynchronous language containment problem, for which we gave a decision procedure in Section 3. Decidability of this problem should be contrasted with the fact that even the special case of checking divergence language containment of MSA is undecidable. This is because the language containment problem for MSA, which is known to be undecidable [16], can be reduced to the problem of checking divergence language containment.

Definition 4.2 For $v \in (\sum \cup \overline{\sum})^*$, a sequence of transitions

$$(q_1, B_1) \overset{v}{\Longrightarrow}_A (q_2, B_2) \overset{v}{\Longrightarrow}_A \cdots \overset{v}{\Longrightarrow}_A (q_k, B_k) \overset{v}{\Longrightarrow}_A (q', B')$$

is called a v-self-covering path starting at (q_1, B_1) if there is an $1 \leq i \leq k$, such that $q_i = q'$, $B_i \subseteq B'$, and if $v = \epsilon$ then the computation $(q_i, B_i) \Longrightarrow_A (q', B')$ involves atleast one transition step.

It is an easy exercise to show that $(q_1, B_1) \uparrow_A^v$ if and only if there is a v-self-covering path starting at (q_1, B_1). We can, in fact, strengthen this characterization further by bounding the length of the v-self-covering path. This is formally stated in the following lemma.

Lemma 4.3 *Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$ be such that $|Q \cup \sum| = n$. Then for $v \in (\sum \cup \overline{\sum})^*$, $q_1 \in Q$, $B_1 \in \{| \sum |\}$, $(q_1, B_1) \uparrow_A^v$ if and only if there is a transition sequence*

$$(q_1, B_1) \overset{v}{\Longrightarrow}_A (q_2, B_2) \overset{v}{\Longrightarrow}_A \cdots \overset{v}{\Longrightarrow}_A (q_k, B_k) \overset{v}{\Longrightarrow}_A (q', B')$$

that is a v-self-covering path such that the total number of transition steps (input, output, and τ-transitions) $\ell \leq 2^{2^{cn \log n}}(m+1)^{cn^{n+1}}$, where $m = |v|$ and c is a constant independent of n, q, B_1, m and v. □

The above lemma is a generalization of Rackoff's result [28] for Petri nets; Rackoff proves the analogue of this lemma for the special case of $v = \epsilon$. The proof of the above lemma is essentially an adaptation of Rackoff's argument to handle arbitrary traces v; such an adaptation is possible because AFSMs are a special class of Petri nets.

Using Lemma 4.3, we can show that given a string v and an AFSM M, we can effectively construct another AFSM M' whose asynchronous language is the same as the asynchronous v-liveness language of M.

Lemma 4.4 *Given M and $v \in (\sum \cup \overline{\sum})^*$, we can effectively construct M' such that $L_A^{\uparrow v}(M) = L_A(M')$.* □

It follows from Lemma 4.4 and Theorem 3.6 that the problem of deciding containment of asynchronous v-liveness languages can be reduced to the problem of containment of asynchronous languages.

Theorem 4.5 *There is an algorithm, which given M_1 and M_2, and $v \in (\sum \cup \overline{\sum})^*$, decides if $L_A^{\uparrow v}(M_1) \subseteq L_A^{\uparrow v}(M_2)$.* □

Theorem 4.6 *The asynchronous v-liveness language containment problem for FSMs is EXPSPACE-hard.* □

5 The May Testing Equivalence

In this section, we show that the algorithm for language containment can be used to decide the may testing equivalence over AFSMs. The may testing equivalence is an instance of the general notion of behavioral equivalence where, roughly, two processes are said to be equivalent if they are indistinguishable in all contexts of use. Specifically, a context in may testing consists of an observing process that runs in parallel and interacts with the process being tested. The observer can in addition signal a success while interacting with the process being tested, and a process is said to pass the test proposed by the observer if there is *at least* one run in which the observer succeeds. Two process are said to be indistinguishable if they pass exactly the same set of tests.

In the following, we consider a generalized version of may testing that we first introduced in [30]. Specifically, we parameterize the equivalence with a set of names ρ which determine the set of observers that are used to decide the equivalence. The names in ρ are treated as being private to the processes being compared, and only the observers that do not interact at these names are used to decide the parameterized equivalence. The usual (unparameterized) may equivalence corresponds to the case where $\rho = \emptyset$.

We first define parameterized may testing over AFSMs.

Definition 5.1 [asynchronous experiment] An asynchronous experiment with M_1 and M_2 is of form $(p|q, B)$, where $p \in Q_1$, $q \in Q_2$, and $B \in \{|\sum_1 \cup \sum_2|\}$. We define a transition relation on asynchronous experiments as $(p|q, B) \longrightarrow_A (p'|q', B')$ if

(i) $a \in B, B' = B \setminus \{a\}$, and $p \xrightarrow{a} p', q = q'$ or $p = p', q \xrightarrow{a} q'$.

(ii) $B' = B \cup \{a\}$, and $p \xrightarrow{\overline{a}} p', q = q'$ or $p = p', q \xrightarrow{\overline{a}} q'$.

(iii) $B' = B$, and $p \xrightarrow{\tau} p', q = q'$ or $p = p', q \xrightarrow{\tau} q'$.

We define the relation, $\Longrightarrow_A$, on asynchronous experiments as the reflexive transitive closure of $\longrightarrow_A$.

Definition 5.2 [asynchronous may testing]
For a set of names ρ, we say that M *respects* the interface ρ if $\rho \cap \sum = \emptyset$. We say $M_1 \underline{\mathsf{may}}\ M_2$ if $(q_1|q_2, \emptyset) \Longrightarrow_A (p_1|p_2, B)$ for some $p_2 \in F_2$. We say $M_1 \sqsubseteq_\rho M_2$ if for every M that respects the interface ρ, we have $M_1 \underline{\mathsf{may}}\ M$ implies $M_2 \underline{\mathsf{may}}\ M$. We say $M_1 \simeq_\rho M_2$ if $M_1 \sqsubseteq_\rho M_2$ and $M_2 \sqsubseteq_\rho M_1$. Note that $\sqsubseteq_\rho$ is a preorder and $\simeq_\rho$ is an equivalence. We write $\sqsubseteq$ as a shorthand for $\sqsubseteq_\emptyset$.

By interpreting the observer reaching a success state as something "bad" happening, the may preorder can be used to reason about safety properties; $M_1 \sqsubseteq_\rho M_2$ can be interpreted as M_1 is a safe implementation of the specification M_2, because if the specification M_2 is guaranteed to not cause anything bad to happen in a context that respects the interface ρ, then the implementation M_1 would also not cause anything bad to happen in the same context.

Theorem 5.3 presents an alternate characterization of the parameterized may preorder, that does not involve a universal quantification over observers. We skip the proof as it is a simple adaptation of a similar characterization over the more general model of asynchronous CCS [5,8]. A few definitions are in order before the theorem. For a set of names ρ and a trace s, we write $s\lceil\rho$ for the trace obtained from s by deleting all the actions in $\rho \cup \overline{\rho}$. For a set of traces L, we define $L\lceil\rho$ to be the set of all traces s in L such that $s\lceil\rho = s$. Note that $L\lceil\rho$ is not the usual lifting of the function $\cdot\lceil\rho$ on traces, to sets of traces.

Theorem 5.3 (characterization of may testing)
Let $M_1 = (Q_1, \sum_1 \cup \overline{\sum_1}, \rightarrow_1, q_1, F_1)$, and $M_2 = (Q_2, \sum_2 \cup \overline{\sum_2}, \rightarrow_2, q_2, F_2)$. Let $\sum = \sum_1 \cup \sum_2$, and $M_2' = (Q_2, \sum \cup \overline{\sum}, \rightarrow_2, q_2, F_2)$. Then $M_1 \sqsubseteq_\rho M_2$ if and only if $L_A(M_1)\lceil\rho \subseteq L_A(M_2')\lceil\rho$. □

Thus, may testing is characterized by trace semantics, and only the traces that are consistent with the interface ρ are to be considered for deciding $\sqsubseteq_\rho$. Further, note that we consider consider M_2' instead of M_2 because if $\sum_1 \backslash (\sum_2 \cup \rho) \neq \emptyset$ then there is always an $s \in L_A(M_1)\lceil\rho$ but $s \notin L_A(M_2)\lceil\rho$. But on the other hand, since inputs of a process are not observable due to asynchrony, it is not necessary that $M_1 \not\sqsubseteq_\rho M_2$.

For the special case of $\rho = \emptyset$, Theorem 5.3 says that $M_1 \sqsubseteq M_2$ if and only if $L_A(M_1) \subseteq L_A(M_2)$. Then by Theorems 3.6 and 3.7 it follows that the unparameterized may preorder $\sqsubseteq$ is decidable over AFSMs and is EXPSPACE-hard. Further, since $\simeq = \sqsubseteq \cap \sqsubseteq^{-1}$, it follows that deciding $\simeq$ is also EXPSPACE-hard. The reader may compare this with the fact that $\sqsubseteq$ is only PSPACE-complete over FSMs [19].

For arbitrary ρ, the relation $\simeq_\rho$ (and hence $\sqsubseteq_\rho$) is undecidable over AFSMs. This can be shown by reducing the language equality problem for labeled Petri nets, which is known to be undecidable [14], to deciding $\simeq_\rho$ over AFSMs. In fact, from the fact that the language equality problem for labeled Petri nets with even two unbounded places [18] is undecidable, it follows that $\simeq_\rho$ is undecidable over AFSMs even for $|\rho| = 2$.

Theorem 5.4 *The relation $\simeq_\rho$ is undecidable over AFSMs.* □

On the other hand, $\sqsubseteq_\rho$ (and hence $\simeq_\rho$) is decidable in PSPACE over FSMs. This is a simple consequence of the following two facts. First, we can show that for FSMs M_1 and M_2, $M_1 \sqsubseteq_\rho M_2$ if and only if $L(M_1)\lceil\rho \subseteq L(M_2)\lceil\rho$. Second, $L(M)\lceil\rho = L(M\lceil\rho)$ where $M\lceil\rho$ is the FSM obtained by removing all the transitions $\xrightarrow{\alpha}$ in M with $\alpha \in \rho \cup \overline{\rho}$ (note that in contrast $L_A(M)\lceil\rho \neq L_A(M\lceil\rho))$.

6 Conclusion

In this paper, we have addressed the problem of verifying asynchronous systems with unbounded and unordered message buffers. We have focused on a simple model of computation, namely AFSM, which is an asynchronous variant of finite state machines. We have presented algorithms that can be used to verify safety and special kinds of liveness properties of AFSMs. AFSMs are an interesting class of infinite state systems for which many problems of interest such as the ones we consider are decidable but undecidable for MSAs [7], lossy channel systems [2,6] and well-structured transition systems in general [1,13].

Our investigations leave some problems open, of which the major ones are the following. First, although we have shown that deciding language containment for AFSMs is EXPSPACE-hard, we do not have a clear upper bound on its complexity. Further, the proof of Theorem 3.7 shows that even deciding the membership of a specific string in the language of an AFSM is EXPSPACE-hard. Thus, obtaining improved upper and lower bounds for our problems is an important future exercise. We have considered three problems for AFSMs: membership, language containment and v-liveness containment. Our reductions demonstrate that these problems are of increasing computational difficulty. In the absence of clear upper and lower bounds, even investigating the complexity of these problems relative to each other would be a useful next step. Another interesting direction to explore would be look at the model checking problem for AFSMs with respect to modal logics, in the vein of [7,6]. These problems may be amenable to tighter complexity analysis.

An alternative to may testing is the notion of *must* testing [26], which is known to be useful for reasoning about liveness properties. Recall that in may testing a process is said to pass a test proposed by an observer if there is at least one computation in which the observer reaches a success state. In contrast, in must testing a process is said to pass a test only if in *every* possible computation the observer reaches a success state. We have shown that the parameterized must equivalence is undecidable over AFSMs, even for parameter sets with two elements. The proof uses a technique that Jancar introduced to prove undecidability of bisimilarity over Petri nets [18]. We do not present the details here due to space constrains. However, the decidability of unparameterized must equivalence is still open.

References

[1] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Tsay Yih-Kuen. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160:109–127, 2000.

[2] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *IEEE International Symposium on Logic in Computer Science*, 1993.

[3] Parosh Aziz Abdulla and Bengt Jonsson. Channel representations in protocol verification. In *CONCUR*, pages 1–15, 2001.

[4] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[5] M. Boreale, R. De Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes.

[6] Ahmed Bouajjani and Richard Mayr. Model checking lossy vector addition systems. In *STACS*, pages 323–333, 1999.

[7] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification over Infinite States. In *Handbook of Process Algebra*, pages 545–623. Elsevier Publishing, 2001.

[8] I. Castellini and M. Hennesy. Testing theories for asynchronous languages. In *FSTTCS*, pages 90–101, 1998. LNCS 1530.

[9] S. Christensen. *Decidability and Decomposition in Process Algebras.* PhD thesis, Department of Computer Science, University of Edinburgh, 1993.

[10] A. Emerson and K. Namjoshi. On model checking for nondeterministic infinite state systems. In *IEEE Symposium on Logic in Computer Science*, 1998.

[11] J. Esparza. Decidability and Complexity of Petri Net problems — An Introduction. In *Advances in Petri Nets*, volume 1491 of *Lecture Notes inComputer Science*, pages 374–428. Springer, 1998.

[12] A. Finkel. A generalization of the procedure of karp miller to well structured transition systems. In *ICALP*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508, 1987.

[13] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001.

[14] M. Hack. Decision problems for Petri nets and vector addition systems. Technical Report MAC, Memo 53, MIT, 1975.

[15] G. H. Higman. Ordering by divisibility in abstract algebras. *Proceedings of London Mathematical Society*, 3:326–336, 1952.

[16] Y. Hirshfeld. Petri nets and the equivalence problem. In *Lecture Notes in Computer Science 832*, pages 165–174. Springer Verlag, 1993.

[17] J.E. Hopcroft and J.D. Ullman. *Introduction to automata theory, languages, and computation.* Addison Wesely, 1979.

[18] P. Jancar. Undecidability of bisimilarity for petri nets and some related problems. *Theoretical Computer Science*, 148:281–301, 1995.

[19] P.C. Kanellakis and S.A.Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):48–68, May 1990.

[20] R. Karp and R. Miller. Parallel program schemata. *Journal of Computing System Science*, 3:147–195, 1969.

[21] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer, 1985.

[22] R. Lipton. The Reachability Problem Requires Exponential Space. Technical Report 62, Yale University, 1976.

[23] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[24] F. Moller. Infinite Results. In *Proceedings of CONCUR*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer, 1996.

[25] F. Moller. A Taxonomy of Infinite State Processes. In *Electronic Notes in Computer Science*, volume 1998. Elsevier, `http://www.elsevier.nl/locate/entcs/volume18.html`, 1998.

[26] R. De Nicola and M. Hennesy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[27] J. L. Peterson. *Petri Net Theory and the Modelling of Systems*. Prentice-Hall, 1981.

[28] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[29] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 671–681, 1997.

[30] P. Thati, R. Ziaei, and G. Agha. A theory of may testing for asynchronous calculi with locality and no name matching. In *Proceedings of the* 9^{th} *International Conference on Algebraic Methodology and Software Technology*, pages 222–238. Springer Verlag, September 2002. LNCS.

[31] Prasanna Thati and Mahesh Viswanathan. Verification of asynchronous systems with unbounded and unordered message buffers. Technical Report UIUC DCS-R-2003-2397, Department of Computer Science, University of Illinois at Urbana Champaign, 2003.

A Appendix

We define the complementation function $\bar{\cdot}$ on visible actions so that the complement of an input is the corresponding output, and vice versa. For a trace r, we write $\{|r|\}_i$ to denote the multiset of all input actions in r, and $\{|r|\}_o$ for the multiset of all output actions in r. We define $\{|r|\} = \{|r|\}_i \cup \{|r|\}_o$. The complementation function is lifted from the set of visible actions to multisets of visible actions the obvious way.

Proof of Theorem 2.4: Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. First, $[L(M)]_\triangleright \subseteq L_A(M)$ is a consequence of the following two observations which are easy to prove: (i) $L(M) \subseteq L_A(M)$, and (ii) if $s \in L_A(M)$ and $s \triangleright r$ by a single (and hence arbitrarily many) application of rules in Definition 2.3, then $r \in L_A(M)$. Next, we show $L_A(M) \subseteq [L(M)]_\triangleright$. Suppose

$$(q_0, \emptyset) \xrightarrow{\alpha_1}_A (q_1, B_1) \xrightarrow{\alpha_2}_A \ldots \xrightarrow{\alpha_n}_A (q_n, B_n)$$

and $r = \hat{\alpha}_1.\cdots.\hat{\alpha}_n$. We prove the stronger statement that there is s such that $q \overset{s}{\Longrightarrow} q_n$, $s \triangleright r$, and $B_n = (\{|r|\}_i \cup \overline{\{|s|\}_o}) \setminus (\overline{\{|r|\}_o} \cup \{|s|\}_i)$. Intuitively, the message buffer B_n contains all the inputs from the environment and the outputs by M, that have neither been consumed by M nor output to the environment. From Definition 2.3, we see that the above expression for B_n encodes the number of times rules 1 and 4 are applied in any derivation of $s \triangleright r$; only for these rules does the expression evaluate to a non-empty multiset when r is set to the RHS and s to the LHS of the rule. Specifically, for all $b \in \mathcal{N}$, $\#(b, B_n)$ equals the number of applications of rules 1 or 4 of Definition 2.3 with the meta-variable a instantiated to b, in any derivation of $s \triangleright r$.

The proof is by induction on n. The base case $n = 0$ is trivial. For the induction step, we may assume there is s' such that $q \xrightarrow{s'} q_{n-1}$, $s' \triangleright r' = \hat{\alpha}_1.\cdots.\hat{\alpha}_{n-1}$, and $B_{n-1} = (\{|r'|\}_i \cup \overline{\{|s'|\}_o}) \setminus (\overline{\{|r'|\}_o} \cup \{|s'|\}_i)$. We now only consider the case where $\alpha_n = \overline{a}$; the other cases are similar. Then $a \in B_{n-1}$, and therefore for a given derivation of $s \triangleright r$, there is an application of rule 1 or 4 of Definition 2.3 in the derivation. Let $s = s'$. We have two subcases:

- The derivation of $s' \triangleright r'$ contains an instance of rule 1. Then we can derive $s \triangleright r'.\overline{a} = r$ from a derivation of $s' \triangleright r'$, by replacing an instance of rule 1 with an instance of rule 3, and delaying the output introduced to the very end by repeated application of rule 5.
- The derivation of $s' \triangleright r'$ contains an instance of rule 4. Then we can derive $s \triangleright r'.\overline{a} = r$ from a derivation of $s' \triangleright r'$, by replacing an instance of rule 4 with repeating applications of rule 5 that delay the output $\overline{a}$ to the very end.

Note that in both cases $B_n = B_{n-1} \setminus \{a\} = (\{|r|\}_i \cup \overline{\{|s|\}_o}) \setminus (\overline{\{|r|\}_o} \cup \{|s|\}_i)$. Thus the induction hypothesis also holds for n. □

Proof of Lemma 3.1: This is an easy consequence of Theorem 2.4. Since $L(M_1) \subseteq L_A(M_1)$, we have $L_A(M_1) \subseteq L_A(M_2)$ implies $L(M_1) \subseteq L_A(M_2)$. Conversely, suppose $L(M_1) \subseteq L_A(M_2)$. Let $\triangleright_1$ be the relation as defined in Definition 2.3 with respect to the alphabet $\sum_1$, and similarly $\triangleright_2$ with respect to the alphabet $\sum_2$. Then $L(M_1) \triangleright_1 L_A(M_1)$, and $L_A(M_2)$ is closed under $\triangleright_2$. Since $\sum_1 \subseteq \sum_2$, $L_A(M_2)$ is also closed under $\triangleright_1$. Then $L(M_1) \subseteq L_A(M_2)$ implies $L_A(M_1) \subseteq L_A(M_2)$. □

Proof of Lemma 3.2: A simple consequence of Higman's lemma [15] for well-quasi-orders (applied to natural numbers extended with ω). □

Proof of Lemma 3.5:

(i) Suppose $\epsilon \in L_A(q, B)$. Then $(q, B) \Longrightarrow_A (q', B')$ for some $q' \in F$. Then, since C *covers* $\mathsf{reach}((q, B), \epsilon, M)$, we have $(q', B'') \in C$ for some $B'' \supseteq B'$. Conversely, suppose $(q', B') \in C$ for some $q' \in F$. Then again since C *covers* $\mathsf{reach}((q, B), \epsilon, M)$, we have $(q, B) \Longrightarrow_A (q', B'')$ for some $B'' \subseteq B'$, which implies $\epsilon \in L_A(q, B)$.

(ii) Suppose $(q', B') \in \mathsf{reach}(\{(q, B)\}, \overline{a}, M)$. Then, $(q, B) \overset{\overline{a}}{\Longrightarrow}_A (q', B')$, which implies $(q, B) \Longrightarrow_A (q', B' \cup \{a\}) \xrightarrow{\overline{a}}_A (q', B')$. Then, $(q', B' \cup \{a\}) \in \mathsf{reach}(\{(q, B)\}, \epsilon, M)$, and since $\mathsf{cover}((q, B), M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$, we have $(q', B'') \in \mathsf{cover}((q, B), M)$ for some $B'' \supseteq B' \cup \{a\}$. But then $(q', B''\backslash\{a\}) \in C$ and $B''\backslash\{a\} \supseteq B'$. We have shown $\mathsf{reach}(\{(q, B)\}, \overline{a}, M) \preceq C$. Now, suppose that $(q', B') \in C$. Then $(q', B' \cup \{a\}) \in \mathsf{cover}((q, B), M)$. Since $\mathsf{cover}((q, B), M)$ *covers* $\mathsf{reach}(\{(q, B)\}, \epsilon, M)$, there are $B_1 \subseteq B_2 \subseteq \ldots$ such that $(q', B_i) \in \mathsf{reach}(\{(q, B)\}, \epsilon, M)$ and $\sqcup_i B_i = B' \cup \{a\}$. Then there is n such that $a \in B_i$ for all $i \geq n$. Then, we have $(q', B_i \setminus \{a\}) \in \mathsf{reach}(\{(q, B)\}, \overline{a}, M)$ for all $i \geq n$, and $\sqcup_{i \geq n}(B_i \setminus \{a\}) = B'$. Thus, we have shown that C *covers* $\mathsf{reach}(\{(q, B)\}, \overline{a}, M)$. □

The cover routine: Roughly, $\mathsf{cover}((q, B), M)$ in Figure A.1 returns a set of configurations that covers all the configurations that M can reach starting from (q, B) and by a sequence of asynchronous transitions labeled with τ. In line 3, we write B^ω to denote $\{|a^\omega \mid a \in B|\}$. Note that, we have assumed that M has no τ-transitions (between the control states); such transitions, if any, can be eliminated by the usual τ-elimination procedure. The expression $\mathsf{configs}(V)$ returns the set of all configurations that occur in the paths (sequences of configurations) in V.

Proof sketch of Theorem 3.7: We reduce the halting problem of counter machines of size n, whose counters are bounded by 2^{2^n}, to the given problem. The halting problem for such counter machines is known to be EXPSPACE-complete [17]. We use a construction first presented by Lipton [22]; for a more recent exposition, see [11]. Given a counter machine C, Lipton constructs a Petri net P of size polynomial in n, such that P reaches a marking with a token in a pre-designated place if and only C halts. Lipton's procedure can be adapted in a straightforward manner to construct an AFSM M such that the string $\overline{\mu} \in L_A(M)$ for a designated alphabet μ, if and only if C halts. Importantly, although AFSMs are a restricted class of Petri nets, it is possible to construct an M whose size is polynomial in n. Now, consider an AFSM M' whose synchronous language $L(M') = \{\overline{\mu}\}$. Then $L_A(M') \subseteq L_A(M)$ if and

```
append($v, (q, B)$)
    if $\exists i$ s.t. $v(i) = (q, B_i)$ and $B \subseteq B_i$ then return $v$
    if $\exists i$ s.t. $v(i) = (q, B_i)$ and $B_i \subseteq B$ then return $v.(q, B_i \cup (B \setminus B_i)^\omega)$
    return $v.(q, B)$
end append

cover($(q, B), (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$)
    $V := \{(q, B)\}$
    repeat
        $V' := V;\ V := \emptyset$
        for all $v \in V'$
            let $v = v'.(q', B')$
                for all $a \in \sum$, $q'' \in Q$
                    if $q' \xrightarrow{a} q''$ and $a \in B'$ then
                        $V := V \cup$ append($v, (q'', B' \setminus \{a\})$)
                    if $q' \xrightarrow{\overline{a}} q''$ then
                        $V := V \cup$ append($v, (q'', B' \cup \{a\})$)
                end for all
            end let
        end for all
    until $V = V'$
    return configs($V$)
end cover
```

Fig. A.1. (An adaptation of) Karp and Miller's algorithm for computing the coverability sets.

only if the given counter machine C halts. □

Proof sketch of Lemma 4.3: We present the main ideas in the proof, highlighting the points of departure from Rackoff's proof for vector addition systems.

The proof is most easily presented by viewing the AFSM M as a vector addition system, which are the same as Petri Nets. In the rest of the proof, we will assume that configurations of M are represented as vectors in $\mathbb{Z}^n$, for some canonical ordering of the states Q and names $\sum$. Each transition step in the AFSM can be viewed as the result of adding a vector in $\mathbb{Z}^n$ to the current configuration. For example, the transition from $q \xrightarrow{a} q'$ can correspond to adding the vector that has -1 in the position of q, 1 in the position of q', -1 in the position of a, with all other positions being 0. We will say $c \overset{v}{\Longrightarrow}_A c'$ if one can add a sequence of vectors, whose labels correspond to taking v interspersed with τ's, to c to get c'. Finally a path $c_1 \overset{v}{\Longrightarrow}_A c_2 \overset{v}{\Longrightarrow}_A \cdots \overset{v}{\Longrightarrow}_A c_k \overset{v}{\Longrightarrow}_A c'$ is v-self-covering if there is an $1 \leq i \leq k$ such that $c_i \leq c'$, and if $v = \epsilon$ then the computation $c_i \Longrightarrow_A c'$ involves atleast one transition step.

There are a couple of concepts that Rackoff defines, that we will find useful as well: an i-bounded, v-self covering path, and an i—r-bounded, v-self-covering path. An i-bounded, v-self covering path is a v-self-covering path such that each configuration in the path is a vector whose first i coordinates are ≥ 0. Observe that any legal sequence of transitions ensures that the configuration of the AFSM in each step can be represented as $c \in \mathbb{Z}^n$ such that each coordinate $c(k) \geq 0$; so i-bounded, v-self-covering paths need not correspond to legal transition sequences of the AFSM, since the coordinates $i+1$ to n of the configurations could be < 0. An i—r-bounded, v-self-covering path is a self covering path where the first i coordinates of each configuration on the path is between 0 and r; once again the coordinates between $i+1$ and n of configurations could be any integer.

The lemma can, therefore, be proved by showing an upper bound on the length of an n-bounded, v-self-covering path. This is proved by inductively obtaining bounds on the length of an i-bounded, v-self-covering paths. But in order to obtain bounds on i-bounded paths, we will need a lemma that will bound the length of i—r-bounded, v-self-covering paths. Since the ideas used to obtain bounds on n-bounded, v-self-covering paths from the bound on i—r-bounded, v-self-covering paths are identical to those of Rackoff, modulo some minor modifications, we skip that proof here and refer the reader to Rackoff's original paper [28].

We will now sketch the proof that bounds the length of an i—r-bounded, v-self-covering path. Let $\rho \equiv c_1 \overset{v}{\Longrightarrow}_A c_2 \overset{v}{\Longrightarrow}_A \cdots \overset{v}{\Longrightarrow}_A c_k \overset{v}{\Longrightarrow}_A d_1 \overset{v}{\Longrightarrow}_A \cdots \overset{v}{\Longrightarrow}_A d_\ell$ be a *minimal* i—r-bounded, v-self-covering path, such that c_1 is the vector representation of (q_1, B_1), and $d_1 \leq d_\ell$. For a vector $c \in \mathbb{Z}^n$, let $\Pi^i(c) \in \mathbb{Z}^i$ be the projection of c onto the first i coordinates. Since ρ is a minimal path, it must be the case that $\Pi^i(c_1), \Pi^i(c_2) \ldots \Pi^i(c_k)$ are all distinct; thus $k \leq r^i \leq r^n$. Further, for any p, q, the length of any continuous block of τ-steps (without visible actions in between) in the sequence $c_p \overset{v}{\Longrightarrow}_A c_q$ must once again be at most $r^i \leq r^n$, for similar reasons. Hence the length of the sequence $c_1 \overset{v^*}{\Longrightarrow}_A c^k$ is at most $r^n \cdot (m+1) \cdot r^n$, where $m = |v|$.

Let us now focus on bounding the length of $d_1, \ldots d_\ell$. Call a sequence of $x_1 \overset{u}{\Longrightarrow}_A x_2 \overset{u}{\Longrightarrow}_A \cdots \overset{u}{\Longrightarrow}_A x_t$ a simple u-loop if $\Pi^i(x_{j_1}) \neq \Pi^i(x_{j_2})$ for $j_1 \neq j_2$ and $\Pi^i(x_1) = \Pi^i(x_t)$, where $u \in (\sum \cup \overline{\sum})^*$. Now, suppose $x_1, \ldots x_t$ is a simple u-loop in $d_1, \ldots d_\ell$. Consider the sequence of transitions obtained by removing the sequence of transitions appearing in the loop $x_1 \ldots x_t$. The sequence of vectors obtained, starting from d_1, by taking this shortened sequence of transitions, will be called the path obtained after removing the u-loop $x_1 \ldots x_t$. Observe that removing $x_1 \ldots x_t$ results in a path, which when projected, will continue to be the projection of an i—r-bounded path, with two important differences: first, the resulting path will not necessarily be self-covering; second, if $u \neq \epsilon$ then the resulting path may not be one that has label v^h, for some h. Since Rackoff considers unlabeled sequences, he does not deal with the second problem. In order to deal with the first problem, Rackoff first removes a series of simple loops to get a much shorter sequence that is once again an i—r-bounded path, but is no longer self covering, while keeping track of the "vector weight" of the loops removed. Care is taken, when removing the loops, to ensure that all of the vectors in the set $\{\Pi^i(d_1), \ldots \Pi^i(d_\ell)\}$ appear in the shortened path; only the multiplicity of these vectors changes. This ensures that the original path can be obtained by adding a non-negative linear combination of loops to the shortened path. The desired path of bounded length is then obtained by searching for another linear combination of loops, whose total weight is equal to the weight of the loops removed. This new linear combination of loops obtained by solving a feasible linear program. The desired bounds follow from the bounds on the solution size of a feasible linear program.

In the presence of labels, the labels of the loops removed needs to be taken into account. The natural choices for loops that can be removed would be those that are labeled ϵ or v^h for some h. In order to bound ℓ we will need to remove loops labeled v^h. However, since the final solution will be obtained by solving a linear program, care must be taken to ensure that the loops that need to be added need not be "very long". This is a problem for loops labeled v^h which have arbitrarily many τ-steps in between. We circumvent this problem by removing loops in two stages: first we remove loops labeled ϵ to ensure that none of the contiguous sequences of τ-steps are too long, and then we remove those labeled v^h. The length of the path after the first stage would be at most $(m+1)(r^n+1)^2\ell$, and after the second stage, we will have ensured that the shortened path is of length at most $(m+1)(r^n+1)^2(r^n+1)^2$. This ensures that the original path can be obtained from the shortened path by adding a linear combination of loops of bounded length. Then once again the desired i—r-bounded, v-self-covering path is obtained by searching of a linear combination of loops to add to the shortened path, by solving a feasible linear program. □

Lemma A.1 *For $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$, and $v \in (\sum \cup \overline{\sum})^*$, there is a finite $B_0 \in \{|\sum|\}$ such that for all $q \in Q$, if $(q, B) \uparrow^v_A$ then $(q, B \cap B_0) \uparrow^v_A$.*

Proof. Let $n = |Q \cup \sum|$. We show the multiset $B_0 \in \{|\sum|\}$ such that $\#(a, B_0) = 2^{2^{cn \log n}}(m+1)^{cn^{n+1}}$ for all $a \in \sum$, where c is the constant in Lemma 4.3 and $m = |v|$, satisfies the property stated above. Now, suppose $(q, B) \uparrow^v_A$. Then by Lemma 4.3, there is a transition sequence $(q, B) \overset{v^*}{\Longrightarrow}_A (q', B_1) \overset{v^+}{\Longrightarrow}_A (q', B_2)$ of length $\leq 2^{2^{cn \log n}}(m+1)^{cn^{n+1}}$ for some $B_2 \supseteq B_1$. Let $B' = B \cap B_0$, and $B'' = (B' \cup (B_1 \setminus B)) \setminus (B \setminus B_1)$. Then clearly, $(q, B') \overset{v^*}{\Longrightarrow}_A (q', B'') \overset{v^+}{\Longrightarrow}_A (q', B'' \cup (B_2 \setminus B_1))$, the length of which is the same as that of $(q, B) \overset{v^+}{\Longrightarrow}_A (q', B_2)$. Then again by Lemma 4.3, $(q, B') \uparrow^v_A$. □ □

Proof of Lemma 4.4: Let $M = (Q, \sum \cup \overline{\sum}, \rightarrow, q_0, F)$. We construct $M' = (Q', \sum \cup \overline{\sum}, \rightarrow', q', F')$ that simulates M, and at any time can non-deterministically choose to examine the contents of its message buffer. If it finds the buffer to be large enough for M to be able to exhibit v^ω from its current state, then M' jumps to an accepting state.

Specifically, let $Q = \{q_1, \ldots, q_n\}$. For each q_i, define the set $\mathcal{B}_i$ as follows. If $(q_i, B) \stackrel{v^\omega}{\not\Longrightarrow}_A$ for every B then $\mathcal{B}_i = \emptyset$. Else $\mathcal{B}_i$ is the finite set $\{B_{i1}, \ldots, B_{ik}\}$ such that: (a) $(q_i, B) \uparrow_A^v$ if and only if $B \supseteq B_{ij}$ for some $B_{ij} \in \mathcal{B}_i$, and (b) $B_{il} \subseteq B_{im}$ implies $B_{il} = B_{im}$. As a consequence of Lemma A.1, we know that such a $\mathcal{B}_i$ exists. In fact, it can be computed as follows. Let B_0 be the multiset produced by Lemma A.1. Enumerate all $B \subseteq B_0$, and check for each if $(q_i, B) \uparrow_A^v$ (Lemma 4.3 gives us a procedure for this), and let $\mathcal{B}_i$ be the set of all such minimal B's for which $(q_i, B) \uparrow_A^v$.

Let $Q' = \{(q_i, B) \mid B = \emptyset, \textit{ or } B \subseteq B' \textit{ for some } B' \in \mathcal{B}_i\} \cup \{f\}$ (note that $\mathcal{B}_i$ may be empty), $q' = (q_0, \emptyset)$, and $F' = \{f\}$. The transition function $\rightarrow'$ is defined as

$$\begin{array}{lll}
(q_i, \emptyset) \xrightarrow{\alpha}' (q_j, \emptyset) & & \text{if } q_i \xrightarrow{\alpha} q_j \\
(q_i, B) \xrightarrow{a}' (q_i, B \cup \{a\}) & & \text{if } (B \cup \{a\}) \subseteq B' \text{ for some } B' \in \mathcal{B}_i \\
(q_i, B) \xrightarrow{\tau}' f & & \text{if } B = B' \text{ for some } B' \in \mathcal{B}_i
\end{array}$$

Note that the transition (sub)graph of M' with only the nodes $(q_i, \emptyset)$ is isomorphic to the transition graph of M. Thus M' can simulate M. But at any time, M' can non-deterministically choose to "examine" the message buffer contents, by using the second and third transition rules above. It is easy to check that $L_A(M') = L_A^{\uparrow v}(M)$. □

Proof of Theorem 4.5: An immediate consequence of Lemma 4.4 and Theorem 3.6. Note that the proof of Lemma 4.4 not only shows the existence of M', but also effectively constructs it. □

Proof of Theorem 4.6: The problem of asynchronous language containment can be reduced in polynomial time to the problem of v-liveness language containment; the theorem then follows from Theorem 3.7. Let M_1 and M_2 be two finite state machines over the alphabet $\sum$. Consider a $\mu \notin \sum$. For each final state q of M_1, add the transition $q \xrightarrow{\overline{\mu}}_A q$ to obtain machine M_1'. Observe that $L_A(M_1) = L_A^{\uparrow v}(M_1') \cap (\sum \cup \overline{\sum})^*$, where $v = \overline{\mu}$. Similarly, construct M_2' from M_2. Thus $L_A(M_1) \subseteq L_A(M_2)$ iff $L_A^{\uparrow v}(M_1) \subseteq L_A^{\uparrow v}(M_2)$. □

Proof of Theorem 5.4: We reduce the language equality problem for labeled Petri nets, which is known to be undecidable [14], to the given problem. Specifically, given Petri nets P_1 and P_2 we construct AFSMs M_1, M_2, and ρ such that $L(P_1) \subseteq L(P_2)$ if and only if $L_A(M_1)\lceil\rho \subseteq L_A(M_2)\lceil\rho$. We are then done by Theorem 5.3.

Suppose $P = (S, T, F, \lambda, \mu)$ is a Petri net, where S and T are disjoint sets of places and transitions, $F \subseteq (S \times T) \cup (T \times S)$ is the flow relation, $\lambda : T \to L$ is the labeling function, and $\mu : S \to \mathbb{N}$ is the initial marking. Without any loss of generality we may assume that μ leaves a single token at exactly one of the places, because otherwise one can always construct another Petri net that that satisfies this condition and has the same language. We can construct an AFSM that "simulates" the net as follows. Let

$$M = (\{r, w\} \times \mathcal{P}(S), \sum \cup \overline{\sum}, \to, (w, \{\mu\}), \{r, w\} \times \mathcal{P}(S))$$

where $\sum = S \cup L \cup \{A\}$ for some $A \notin S \cup L$, $\{\mu\}$ denotes the singleton $\{s\}$ such that $\mu(s) \neq 0$, $\mathcal{P}(S)$ denotes the powerset of S, and the transition relation $\to$ is defined by the following rules. For $X \subseteq \mathcal{P}(S)$, $s \in S$ and $t \in T$:

- $(r, X) \xrightarrow{s} (r, X \cup \{s\})$ if $s \notin X$.
- $(r, X) \xrightarrow{\lambda(t)} (w, (X \setminus {}^{\bullet}t) \cup t^{\bullet})$ if ${}^{\bullet}t \subseteq X$.
- $(w, X) \xrightarrow{\overline{s}} (w, X \setminus \{s\})$ if $s \in X$.
- $(w, \emptyset) \xrightarrow{\overline{A}} (r, \emptyset)$.

where ${}^{\bullet}t = \{s \mid s \in S,\ (s,t) \in F\}$ and $t^{\bullet} = \{s \mid s \in S,\ (t,s) \in F\}$ are the preset and postset of the transition t. A marking of places in the Petri net P is encoded as the message buffer of M. To simulate a transition of P, M reads its message buffer and non-deterministically makes a transition that is enabled, i.e. whose preset has all places with non-zero marking. The transition in M has the same label as the corresponding transition in P. After the transition, M performs output actions so that its message buffer corresponds to the new marking that P reaches after its transition.

Now, given Petri nets P_1 and P_2, let M_1 and M_2 be as constructed above. Let $\rho = S_1 \cup S_2$, and let $\mathcal{L}$ be the set used to label transitions in P_1 and P_2. First, suppose $L(P_1) \not\subseteq L(P_2)$. Then there is r such that $r \in L(P_1)$ and $r \notin L(P_2)$. Suppose $r = l_1.l_2.\cdots.l_n$. Let $r' = \overline{A}.l_1.\overline{A}.l_2.\overline{A}.\cdots.l_n.\overline{A}$. Then $r' \in L_A(M_1)\lceil\rho$, but $r' \notin L_A(M_2)\lceil\rho$, and hence $L_A(M_1)\lceil\rho \not\subseteq L_A(M_2)\lceil\rho$. Now, suppose $L(P_1) \subseteq L(P_2)$. Let

$$L_1 = \{\overline{A}.l_1.\overline{A}.\cdots.l_n.\overline{A} \mid l_1.\cdots.l_n \in L(P_1)\}$$
$$L_2 = \{\overline{A}.l_1.\overline{A}.\cdots.l_n.\overline{A} \mid l_1.\cdots.l_n \in L(P_2)\}$$

It is clear from the construction of M_1 and M_2 that $L_A(M_1)\lceil\rho = [L_1]_{\triangleright}$ and $L_A(M_2)\lceil\rho = [L_2]_{\triangleright}$, where the relation $\triangleright$ is defined with respect to the alphabet $\mathcal{L} \cup \{A\}$. Now, since $L(P_1) \subseteq L(P_2)$, we have $L_1 \subseteq L_2$, which implies $[L_1]_{\triangleright} \subseteq [L_2]_{\triangleright}$. But since we assume $\sum_1 \setminus (\sum_2 \cup \rho) = \emptyset$ (see Section 5), it follows that

$L_A(M_1)\lceil\rho \subseteq L_A(M_2)\lceil\rho$. $\square$

In the following, let $\sim$ denote strong bisimilarity.

Theorem A.2 *$(q_1, B_1) \sim (q_2, B_2)$ if and only if $B_1 = B_2$ and $q_1 \sim q_2$.*

Proof. (if) Suppose $q_1 \sim q_2$, and let G be a bisimulation with $(q_1, q_2) \in G$. It is easy to verify that $H = \{((p, B), (q, B)) \mid (p, q) \in G\}$ is a bisimulation.

(only if) Suppose $(q_1, B_1) \sim (q_2, B_2)$ and let H be a bisimulation with $((q_1, B_1), (q_1, B_2)) \in H$. Note that since the contents of message buffers can be (exactly) observed with a sequence of output transitions (without any τ transitions), it is clear that for any $((p, B), (q, B')) \in H$ we have $B = B'$. We show that $G = \{(p, q) \mid ((p, B), (q, B)) \in H\}$ is a bisimulation. For $(p, q) \in G$ we are to show that if $p \xrightarrow{\alpha} p'$ then $q \xrightarrow{\alpha} q'$ such that $(q, q') \in G$ and vice versa. We only consider the case where α is an input, the other cases are similar. Let $p \xrightarrow{a} p'$. Now, since $(p, q) \in G$ there is some $((p, B), (q, B)) \in H$. Then $(p, B) \xrightarrow{a} (p, B \cup \{a\}) \xrightarrow{\tau} (p', B)$, and since H is a bisimulation we have $(q, B) \xrightarrow{a} (q, B \cup \{a\}) \xrightarrow{\tau} (q', B')$ such that $((p', B), (q', B')) \in H$. But then $B = B'$ which in turn implies that $q \xrightarrow{a} q'$, and we have $(p', q') \in G$. The argument for $q \xrightarrow{a} q'$ is similar. So G is a bisimulation. $\square$ $\square$