Deciding Bisimulation Equivalences for a Class of Non-Finite-State Programs

Bengt Jonsson and Joachim Parrow

Swedish Institute of Computer Science, Kista, Sweden
and
Uppsala University, Sweden

Note to printer: Both affiliations apply to both authors.

August 1989 (Revised July 1991)

Footnote on first page:

This paper is a revised and extended version of a paper that has appeared under the same title in the Proceedings of the 6^{th} Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 1989, published as Lecture Notes in Computer Science Vol. 349, Springer Verlag, pp. 421-433.

Running title: Deciding Bisimulation Equivalences for Non-Finite-State Programs

Address for proofs: Bengt Jonsson, SICS, Box 1263, S-16428 Kista, Sweden.

Special symbols: Since this copy is mathematically type-set, only a few of the less obvious symbols are listed below.

$\mathbf{0}$	Boldface zero
$\emptyset$	Empty set
μ, τ	Greek letters
$\sim$, $\approx$, $\simeq$, $=$	Various equivalence symbols
$\mathcal{D}$, $\mathcal{S}$, $\mathcal{V}$, $\mathcal{U}$	Particular sets and functions (script style)
$\longrightarrow$, $\Longrightarrow$	Transition arrows
$\Rightarrow$, $\Leftarrow$, $\Longleftarrow$	Other arrows
$\bigcup_i$	Indexed union
$\sum_i$, $\sum_{i\in I}$, $\sum_{i=1}^{K}$	Indexed sums
$\cup$, $\cap$, $\times$, $\subseteq$, $\in$	Operations on sets etc.
$\{$, $\}$	Set brackets
$\widetilde{e}$, $\widetilde{x}$	Shorthand for $e_1, \ldots, e_n$, $x_1, \ldots, x_n$
$::=$	Syntax definition symbol
$\Box$	Used to signal end of a proof

Note that the transition arrow will come in subscripted and superscripted variants. The subscript should be placed in the normal position, but the superscript should be on top of the arrow, as in

$$\stackrel{\mu}{\Longrightarrow}_{\mathrm{M}}$$

where M is subscript and μ superscript.

Also note that punctuation marks such as "?", "!" and "." are used as mathematical symbols within some formulae. We therefore prefer not to punctuate displayed formulae.

Some displayed formulae are equivalent to program code and should *not* be in italics. We signify these with a "typewriter" font in our manuscript, for example

$$\mathtt{MemoryCell(x)} \Longleftarrow \mathtt{read!x\ .\ MemoryCell(x)} \quad + \quad \mathtt{write?y\ .\ MemoryCell(y)}$$

We suggest you use whatever font is normally used for program code here.

Finally, take care to distinguish "v" (the ordinary letter v), "v" (a mathematical variable) and "$\mathtt{v}$" (a particular constant). Only the last two occur in formulae and the may come with subscripts or primes; the constant "$\mathtt{v}$" can be typeset in any way which is easily distinguishable from v. Note also the difference between "V" (the ordinary letter) and $\mathcal{V}$ (a mathematical symbol in script style).

Abstract

Traditionally, many automatic program verification techniques are applicable only to finite-state programs. In this paper we extend some of these techniques to a class of infinite-state programs that, in addition to having a finite-state control component, may read, store, and write but not perform any other computations on data. Such programs are *data-independent* in the sense that their behavior does not depend on the actual data values supplied. We consider the problems of deciding strong equivalence and observation equivalence, defined by bisimulations (as in CCS), between such programs. These equivalences have major applications in verification of communication protocols. We present reductions of these problems to the problem of deciding strong equivalence and observation equivalence between finite-state programs, for which polynomial time algorithms exist. The equivalence problems on data-independent programs are shown to be NP-hard in the size of the programs.

1 Introduction

One of the advantages of producing formal specifications of software systems is the possibility to analyze and verify a system in a rigorous way. In this paper we focus on the particular verification problem of proving two programs equivalent. Equivalence is an often used criterion for a correct refinement of a specification, or for a correct optimization of a program. Our contribution is to extend the applicability of algorithms for deciding equivalence.

In general, most interesting verification problems are undecidable. Existing verification algorithms cover subclasses of programs; one prominent such subclass is the *finite-state* programs, consisting of programs which can only reach a finite number of distinct states during execution. Verification algorithms are usually based on *state space exploration*: first construct the set of states that the program can reach, and then analyze this set. Such algorithms exist for proving absence of deadlock, proving properties expressed in propositional temporal logic, proving trace equivalence and bisimulation equivalence etc. These verifications have been especially important for communication protocols and hardware structures.

Many important programs such as communication protocols, memories, bounded buffers, queues, and stacks are not finite-state. One reason for this is that they operate on data values from a potentially infinite domain: an input operation can result in an infinite number of different states corresponding to the infinitely many data values. The limitations of verification algorithms to finite-state programs has lead verifiers of communication protocols to verify only the control aspect (this is usually finite-state) and ignore the data aspect (which is not finite-state if the data domain is infinite).

The contribution of this paper is to extend the applicability of verification algorithms to a class of non-finite-state programs expressed in a dialect of CCS ([18]). The programs have finite-state control components and operate on data values from a potentially infinite set of data. They may read data values from input ports and subsequently write them on output ports; they may however not compute functions on the data values, nor base any decisions on tests on data values. Such programs are data-independent, in the sense that their behaviors are independent of the actual data domain on which they operate. It turns out that communication protocols, bounded buffers, queues, stacks, and memories are often data-independent.

We shall consider the problem of deciding observation equivalence, as defined in CCS. Observation equivalence has been widely used as a correctness criterion, e.g. in verification of communication protocols. Observation equivalence is strictly stronger than other equivalences which have been suggested as intuitively appealing, can be decided efficiently for finite-state programs, and has a simple theory.

Our main result is an algorithm that transforms an equivalence problem between two data-independent programs into an equivalence problem between two finite-state programs, for which there is a polynomial time decision procedure ([13]). We thus obtain an algorithm for deciding equivalence between data-independent programs. The main idea behind the transformation is that if a program is data-independent then it should be irrelevant that there is an infinite number of different data values, since all data values are treated uniformly by the program. Thus we can simulate the program on a finite set of data values. In the worst case, this results in an exponential blowup of the program size. In the paper, we also prove that the equivalence problem is NP-hard for the class of programs that we consider.

The rest of the paper is organized as follows. In Section 2 we comment on related work. Section 3 defines the syntax and semantics of programs, and the concept of bisimulation equivalence. In Section 4 we present algorithms for deciding strong equivalence and observation equivalence, and Section 5 contains a result on the complexity of these problems. In Section 6 we indicate an extension to parallel programs, and finally Section 7 contains a small example of a verification which is made possible with our algorithm. Some of the longer proofs are collected in an appendix.

2 Related Work

The problem of extending finite-state verification algorithms to non-finite-state programs has been considered by Wolper ([25]) in a different context: that of verifying that a program satisfies a temporal logic specification. Wolper shows how to reformulate this problem to a similar one over a smaller, finite data domain. The verification problem can then be handled by standard finite-state techniques (e.g., [7], [23]). Wolper gives an extensional definition of what it means for a program to be data-independent. This definition admits more programs than those that we consider, and (as Wolper remarks) makes the problem of checking whether a program is data-independent undecidable. We consider programs that from their syntactical definitions can easily be seen to be data-independent. Another difference between our work and that of Wolper is that we consider the problem of deciding equivalence between programs.

Our methods are related to ideas in symbolic execution ([4]): to execute the program on symbolic data values. Brand and Joyner have used this technique to verify communication protocols, not automatically, but with substantial mechanical support ([5]).

Observation equivalence has been widely used as a correctness criterion in verification of communication protocols, ([2, 14, 15, 21, 22, 24]), and other parallel systems ([10]). Other equivalences for nondeterministic programs have been explored in many papers; examples include [6, 19, 1, 3]. Algorithms for the equivalence problems in the finite-state case have been presented by Kanellakis and Smolka ([13]), by Paige and Tarjan ([20]), and by Cleaveland and Hennessy ([8]). Our work extends the applicability of their results.

A more remotely related area is the study of other equivalence problems for sequential programs and for program schemas. Jones and Muchnik ([12]), for instance, consider the problem of equivalence between generalized sequential machines that can test for equivalence between data values.

3 Preliminaries

3.1 Programs

We shall define a language for expressing nondeterministic and parallel programs. Programs in this language are data-independent in the sense that they do not perform computations or tests on data values and that any value output by a program must either have been previously received or stored in the program initially. The primitives of the language include input, output, nondeterminism, parallelism, and recursion. We defer the primitive for parallelism to Section 6 in order to simplify the presentation of our main ideas. Our notation is taken from CCS ([18]), with the modification that input is denoted by a question mark (?) and output by an exclamation mark (!).

We assume the following (pairwise disjoint) sets

- a set of *ports*, ranged over by $a, b, c, \ldots$.
- a set of *variables*, ranged over by $x, y, \ldots$.
- an infinite set $\mathcal{D}$ of *data values*, ranged over by $d, d_1, d_2, \ldots$.
- a finite set of *identifiers*, ranged over by Id. Each identifier has a nonnegative *arity*.

A *value expression* is either a variable or a data value. We will use $e, e_1, \ldots$ to range over value expressions. Normally, value expressions would represent the results of computations on data. Since we restrict attention to programs which do not perform computations, variables and data values will be the only relevant value expressions.

The set of *behavior expressions*, ranged over by $A, B, \ldots$, is defined recursively as follows:

$$A \quad ::= \quad a?x.A \quad | \quad a!e.A \quad | \quad \sum_{i \in I} A_i \quad | \quad Id(e_1, \ldots, e_n)$$

We assume in $Id(e_1, \ldots, e_n)$ that Id has arity n. Since we are only interested in syntactically finite programs we also assume that the index set I in $\sum_{i \in I} A_i$ is finite, and we will sometimes use the notation $A_1 + \cdots + A_n$ for $\sum_{i \in [1 \ldots n]} A_i$. For the special case where I is empty we use the symbol $\mathbf{0}$, i.e. $\mathbf{0} = \sum_{i \in \emptyset} A_i$.

An occurrence of a variable x is *bound* in A if it occurs within a subexpression of the form $a?x.A'$. Let $FV(A)$, the *free variables* of A, denote the set of variables in A that have an occurrence which is not bound. A *program* is a behavior expression with no free variables.

We assume that every identifier Id of arity n is *defined* by a clause

$$Id(x_1, \ldots, x_n) \Longleftarrow A$$

where A is a behavior expression with $FV(A) \subseteq \{x_1, \ldots, x_n\}$, and $x_1, \ldots, x_n$ are distinct variables. Note that this allows recursive definitions such as

$$\texttt{MemoryCell(x)} \Longleftarrow \texttt{read!x . MemoryCell(x)} \quad + \quad \texttt{write?y . MemoryCell(y)}$$

The intuition is that if Id is defined by $Id(x_1, \ldots, x_n) \Longleftarrow A$, then the program $Id(d_1, \ldots, d_n)$ behaves precisely as A with $x_1, \ldots, x_n$ replaced by $d_1, \ldots, d_n$.

3.2 Operational Semantics

An *action* (or communication event) is of the form $a?d$ or $a!d$. The action $a?d$ is an *input action*, and represents the reception of the data value d in a communication on port a. The action $a!d$ is an *output action*, and represents the transmission of the data value d in a communication on port a. We use μ to range over actions.

Intuitively, the program $a?x.A$ can first perform an action $a?d$ and thereafter behave like A where x is replaced by d. The program $a!d.A$ can first perform the action $a!d$ and thereafter behave like A. The program $\sum_{i \in I} A_i$ can behave like either of A_i, hence the program $\mathbf{0}$ cannot perform any actions.

Formally, behaviors are defined by a *transition relation.* In general a transition relation $\longrightarrow$ is a set of triples of the form $A \stackrel{\mu}{\longrightarrow} A'$ where A and A' are programs, and μ is an action. Intuitively, the transition $A \stackrel{\mu}{\longrightarrow} A'$ states that the program A can perform the action μ and thereby become the program A'. We define the operational semantics of programs by the particular transition relation $\longrightarrow_{\mathrm{M}}$ which is the least relation satisfying the following rules:

$$\textsc{out}_{\mathrm{M}}: \quad a!d.A \stackrel{a!d}{\longrightarrow}_{\mathrm{M}} A$$

$$\textsc{in}_{\mathrm{M}}: \quad a?x.A \stackrel{a?d}{\longrightarrow}_{\mathrm{M}} A[d/x] \qquad \text{for all } d \in \mathcal{D}$$

$$\textsc{sum}_{\mathrm{M}}: \quad \frac{A_j \stackrel{\mu}{\longrightarrow}_{\mathrm{M}} A'}{\sum_{i \in I} A_i \stackrel{\mu}{\longrightarrow}_{\mathrm{M}} A'} \qquad \text{for all } j \in I$$

$$\textsc{id}_{\mathrm{M}}: \quad \frac{A[d_1, \ldots, d_n/x_1, \ldots, x_n] \stackrel{\mu}{\longrightarrow}_{\mathrm{M}} A'}{Id(d_1, \ldots, d_n) \stackrel{\mu}{\longrightarrow}_{\mathrm{M}} A'} \qquad \text{if } Id(x_1, \ldots, x_n) \Longleftarrow A$$

Here and in the following, substitution of e for x in A, written $A[e/x]$, is defined in the usual way, using renaming of bound variables when necessary to avoid captures. Similarly, $A[e_1, \ldots, e_n/x_1, \ldots, x_n]$ means simultaneously substituting each x_i with e_i (here the x_i:s must be distinct). We use the subscript M in $\longrightarrow_{\mathrm{M}}$ to emphasize the fact that this transition relation is essentially the same as defined by Milner [18].

3.3 Bisimulations

In this paper, we shall give a method for deciding equivalences as defined by Milner ([17]). We will begin with the equivalence called "strong equivalence" in [18], and proceed to "observation equivalence" in Section 4.3. Both equivalences are defined in terms of *bisimulations*. Since we will treat many definitions of bisimulation which only differ in the transition relation that they use, it will be convenient to define bisimulation parametrized on a transition relation.

Let $\longrightarrow$ be a transition relation. A binary relation $\sim$ on the set of programs is a $\longrightarrow$-*bisimulation* if $A \sim B$ implies that for all μ:

- if $A \xrightarrow{\mu} A'$ then there is a B' such that $B \xrightarrow{\mu} B'$ and $A' \sim B'$.
- if $B \xrightarrow{\mu} B'$ then there is an A' such that $A \xrightarrow{\mu} A'$ and $A' \sim B'$.

Two programs A and B are $\longrightarrow$-*equivalent*, if there is a $\longrightarrow$-bisimulation $\sim$ such that $A \sim B$. In particular $\longrightarrow_{\mathrm{M}}$-equivalence, which we will also call M-equivalence and denote by $\sim_{\mathrm{M}}$, corresponds to "strong equivalence" in [18]. In Section 4.3, we will define observation equivalence by an analogous definition.

4 Deciding Bisimulation Equivalence

4.1 Schematic Names

Let $ST(A, \longrightarrow)$, the *states* of a program A with respect to a transition relation $\longrightarrow$, be the set of programs A' that can be reached from A via a sequence $A \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_k} A'$ of transitions. A program A is *finite-state* with respect to $\longrightarrow$ iff $ST(A, \longrightarrow)$ is finite. For the class of programs which are finite-state with respect to $\longrightarrow$ and for which $ST(A, \longrightarrow)$ can be computed from A, the problem of $\longrightarrow$-equivalence is decidable. Kannellakis and Smolka [13] have presented an algorithm which runs in polynomial time in the size of $ST(A, \longrightarrow)$.

Unfortunately, this result does not apply directly to $\longrightarrow_{\mathrm{M}}$ since our programs are not always finite-state. The reason for this is that programs of form $a?x.A$ may do infinitely many actions resulting in infinitely many different states, corresponding to the infinitely many data values that may be received.

In this paper, we shall present an alternative "finitary" transition relation $\longrightarrow_{\mathrm{F}}$ which makes programs finite-state by only allowing one action for each program of form $a?x.A$. The idea behind this transition relation is related to the idea of using schematic names for data values in analysis by symbolic execution ([4]). Our definition will ensure that a program $a?x.A$ can only perform one transition, of form $a?x.A \xrightarrow{a?v}_{\mathrm{F}} A[v/x]$, where v is a *schematic name*. The difficulty with this approach is to determine, for each program of form $a?x.A$, the appropriate schematic name v.

On the one hand, different constructs of the form $a?x$ in a program sometimes require different schematic names, otherwise information about the relationship between input and output actions may be lost. For instance, we must be able to distinguish the programs $A = a?x \,.\, a?y \,.\, B$ and $A' = a?y \,.\, a?x \,.\, B$ if the behavior expression B treats the variables x and y differently.

On the other hand, we cannot grant each input transition a unique schematic name; in that case even the simple agent Id defined by $Id \Longleftarrow a?x\,.\,a!x\,.\,Id$ would have an infinite state space (at each recurrence of Id a new schematic name would be needed). We cannot even use one schematic name per variable, since we want Id above to have the same transitions as Id' where $Id' \Longleftarrow a?y\,.\,a!y\,.\,Id'$ which only differs from Id in the name of the bound variable. A major technical idea in this paper is to choose the schematic names for input in an appropriate way.

In the following we assume a set $\mathcal{V} = \{\mathtt{v}_1, \mathtt{v}_2, \ldots\}$ of *schematic names* ranged over by v, v', ... For reasons which will become apparent later we assume $\mathcal{V}$ to be potentially infinite, and define an order $<$ on $\mathcal{V}$ by $\mathtt{v}_i < \mathtt{v}_j$ if $i < j$. As usual we will write $v \leq v'$ to mean that either $v = v'$ or $v < v'$. The schematic names will act as representatives of data values; they will for instance occur in actions. We will also use substitutions of value expressions for schematic names; in this respect the schematic names will behave as variables.

A *schematic action* is an action, possibly with a schematic name instead of a data value. The terms *schematic behavior expression* and *schematic program* are defined analogously. A formal definition of this terminology is obtained by replacing the set $\mathcal{D}$ with the set $\mathcal{D} \cup \mathcal{V}$ in the definitions of value expression, action, behavior expression, and program in Section 3.1. However, in an identifier definition $Id(x_1, \ldots, x_n) \Longleftarrow A$, we still require A to be a (non-schematic) behavior expression.

The definition of the transition relation $\longrightarrow_{\mathrm{M}}$ in Section 3.2 is extended to schematic programs and schematic actions: in the rules $\mathrm{OUT}_{\mathrm{M}}$ and IN_{M}, d ranges over $\mathcal{D} \cup \mathcal{V}$. It is not difficult to prove that this extension does not affect M-equivalence between ordinary programs. The point of all this is that we will derive a decision procedure for M-equivalence of schematic programs, and hence also for ordinary programs.

4.2 Strong Equivalence

In this section we will define the transition relation $\longrightarrow_{\mathrm{F}}$ which differs from $\longrightarrow_{\mathrm{M}}$ in that input transitions contain schematic names rather than data values. The following auxiliary definitions will be needed to achieve this:

- When A is a schematic program, the name v is *used* by A if there is a sequence of transitions

  $$A = A_1 \xrightarrow{\mu_1}_{\mathrm{M}} \quad \ldots \quad \xrightarrow{\mu_{k-1}}_{\mathrm{M}} A_k \xrightarrow{a!v}_{\mathrm{M}} A_{k+1}$$

  starting with A, in which the last action is $a!v$, and in which no input transition contains the name v, i.e. is of the form $A_j \xrightarrow{b?v}_{\mathrm{M}} A_{j+1}$.

- $nextname(A)$ is defined to be the least (w.r.t. $<$) schematic name which is not used by A.

It follows immediately that $nextname(A)$ is not used by A, and that $v < nextname(A)$ implies that v is used by A. The definition is naturally extended to schematic behavior expressions as follows: if A has x free, then v is *used* by A if v is used by $A[d/x]$ for some data value d (the choice of d does not affect the names used by $A[d/x]$).

Intuitively, v is used when it may occur in some future output action without first being received in an input action. It is easy to see that v must occur (syntactically) in A if v is used by A. On the other hand, v may well occur in A without being used: consider e.g. the following identifier definition:

$$Id(x,y) \Longleftarrow a!x \,.\, Id(x,y)$$

Here, v is the only name which is used in the program $Id(v,v')$, and consequently $nextname(Id(\mathtt{v}_1,\mathtt{v}_2)) = \mathtt{v}_2$ and $nextname(Id(\mathtt{v}_3,\mathtt{v}_1)) = \mathtt{v}_1$.

Since our transition relation $\longrightarrow_{\mathrm{F}}$ will depend on $nextname$ it is important to know that $nextname(A)$ is effectively computable from A:

Proposition 1 *$nextname(A)$ is computable in polynomial time.*

Proof: See the Appendix. □

The relation $\stackrel{\mu}{\longrightarrow}_{\mathrm{F}}$ on schematic programs is now defined as the least relation satisfying the following rules (here and in the following u ranges over $\mathcal{V} \cup \mathcal{D}$):

$$\textsc{out}_{\mathrm{F}}: \quad a!u.A \stackrel{a!u}{\longrightarrow}_{\mathrm{F}} A$$

$$\textsc{in}_{\mathrm{F}}: \quad a?x.A \stackrel{a?v}{\longrightarrow}_{\mathrm{F}} A[v/x] \qquad \text{for } v = nextname(A)$$

$$\textsc{sum}_{\mathrm{F}}: \quad \frac{A_j \stackrel{\mu}{\longrightarrow}_{\mathrm{F}} A'}{\sum_{i\in I} A_i \stackrel{\mu}{\longrightarrow}_{\mathrm{F}} A'} \qquad \text{if } j \in I$$

$$\textsc{id}_{\mathrm{F}}: \quad \frac{A[u_1,\ldots,u_n/x_1,\ldots,x_n] \stackrel{\mu}{\longrightarrow}_{\mathrm{F}} A'}{Id(u_1,\ldots,u_n) \stackrel{\mu}{\longrightarrow}_{\mathrm{F}} A'} \qquad \text{if } Id(x_1,\ldots,x_n) \Longleftarrow A$$

Let us as an example take the `MemoryCell` defined by

$$\mathtt{MemoryCell(x)} \Longleftarrow \mathtt{read!x} \,.\, \mathtt{MemoryCell(x)} \quad + \quad \mathtt{write?y} \,.\, \mathtt{MemoryCell(y)}$$

Here evidently $nextname(\mathtt{MemoryCell(d)}) = \mathtt{v}_1$, hence `MemoryCell(d)` will have the following transitions:

$$\mathtt{MemoryCell(d)} \stackrel{\mathtt{read!d}}{\longrightarrow}_{\mathrm{F}} \mathtt{MemoryCell(d)}$$
$$\mathtt{MemoryCell(d)} \stackrel{\mathtt{write?v_1}}{\longrightarrow}_{\mathrm{F}} \mathtt{MemoryCell(v_1)}$$

and $\mathtt{MemoryCell(v_1)}$ will similarly have the following transitions:

$$\texttt{MemoryCell}(\texttt{v}_1) \xrightarrow{\texttt{read!v}_1}_{\mathrm{F}} \texttt{MemoryCell}(\texttt{v}_1)$$
$$\texttt{MemoryCell}(\texttt{v}_1) \xrightarrow{\texttt{write?v}_1}_{\mathrm{F}} \texttt{MemoryCell}(\texttt{v}_1)$$

This example demonstrates that `MemoryCell(d)` is finite-state w.r.t. our new transition relation. In general, the following proposition holds:

Proposition 2 *For any schematic program A, the set $ST(A, \longrightarrow_{\mathrm{F}})$ is finite.*

Proof: In order to formulate how the size of $ST(A, \longrightarrow_{\mathrm{F}})$ depends on the size of A we make some definitions. For (schematic) behavior expressions extend the usual definition of "subterms" by including all subterms (in the standard sense) in identifier definitions as subterms of any term. So, for example, given the definition $Id(x) \Longleftarrow a!x\,.\,\mathbf{0}$, the program $P = b!1\,.\,Id(2)$ has the subterms

$$P,\ Id(2),\ a!x\,.\,\mathbf{0},\ \mathbf{0}$$

For a behavior expression A, define $size(A)$ to be the number of symbols in A and in all identifier definitions, and $vsize(A)$ to be the maximum number of free variables in any subterm of A. Note that $size(A)$ and $vsize(A)$ are finite, since there are only finitely many identifiers, and that $size(A) > vsize(A)$.

In order to prove Proposition 2, we first prove (by induction on the definition of $\longrightarrow_{\mathrm{F}}$) that if $A \longrightarrow_{\mathrm{F}} \cdots \longrightarrow_{\mathrm{F}} B$, then

1. B is an instance of a subterm B' of A (i.e., B is obtained by substituting schematic names for variables in B');
2. If v is a schematic name in B, then $v \in \{\mathtt{v}_1, \ldots, \mathtt{v}_{vsize(A)}\}$.

It follows that there are not more than $size(A) * (vsize(A))^{(vsize(A))}$ different such B. $\Box$

Recalling the definitions from Section 3.3, the transition relation $\longrightarrow_{\mathrm{F}}$ induces an equivalence relation on programs: $\longrightarrow_{\mathrm{F}}$-equivalence, which we will also call F-equivalence and write $\sim_{\mathrm{F}}$. Since $ST(A, \longrightarrow_{\mathrm{F}})$ is finite for each A, and since $\longrightarrow_{\mathrm{F}}$ is obviously computable from A (given that $nextname(A)$ is computable), there exists an algorithm which decides whether two schematic programs are F-equivalent. Our main result is that we can use F-equivalence instead of M-equivalence. Namely, we have:

Theorem 3 *Two schematic programs are* M*-equivalent precisely if they are* F*-equivalent.*

Proof: see the Appendix. In the proof, one key fact about $\longrightarrow_{\mathrm{F}}$ is that if two schematic programs $A[v/x]$ and $B[v/x]$ are M-equivalent for all v then they have exactly the same set of used schematic names, and hence $a?x.A$ and $a?x.B$ have the same initial transition $\xrightarrow{a?v}_{\mathrm{F}}$ (for $v = nextname(A) = nextname(B)$) leading to equivalent programs. $\Box$

Theorem 3 gives a decision procedure for $\sim_{\mathrm{M}}$. For instance, the algorithm by Kanellakis and Smolka mentioned in the introduction can be used. This algorithm takes polynomial time in the size of the state spaces of the programs to be compared. In Section 5 we further investigate the complexity of this algorithm in terms of the sizes of A and B.

4.3 Observation Equivalence

The programs as defined in Section 3.1 can not perform internal actions. This is a serious drawback if we want to consider systems consisting of several programs running in parallel, and where communications between these programs are internal to the system. We shall therefore (following [18]) extend the program notation by a new *unobservable* action τ. In the operational semantics a transition of the form $A \xrightarrow{\tau}_{\mathrm{M}} A'$ intuitively means that A can transform into the program A' without performing any externally observable action. We extend the syntax of (schematic) behavior expressions by allowing the construct $\tau.A$. The definition of the transition relations $\longrightarrow_{\mathrm{M}}$ and $\longrightarrow_{\mathrm{F}}$ is extended by

$$\text{TAU}_{\mathrm{M}}: \quad \tau.A \xrightarrow{\tau}_{\mathrm{M}} A \qquad\qquad \text{and} \qquad\qquad \text{TAU}_{\mathrm{F}}: \quad \tau.A \xrightarrow{\tau}_{\mathrm{F}} A$$

In the context of τ actions, M-equivalence is too strong to be interesting. Instead we use bisimulation equivalence based on the transition relation $\Longrightarrow_{\mathrm{M}}$ which disregards unobservable actions: define $A \overset{\mu}{\Longrightarrow}_{\mathrm{M}} A'$ to mean that there exists a sequence of transitions

$$A \xrightarrow{\tau}_{\mathrm{M}} \quad \cdots \quad \xrightarrow{\tau}_{\mathrm{M}} A_k \xrightarrow{\mu}_{\mathrm{M}} A_{k+1} \xrightarrow{\tau}_{\mathrm{M}} \quad \cdots \quad \xrightarrow{\tau}_{\mathrm{M}} A'$$

If μ is τ, then we also allow the case where $A = A'$ and the sequence is empty.

Two (schematic) programs are said to be *observation equivalent* iff they are $\Longrightarrow_{\mathrm{M}}$-equivalent.

In order to prove an analogue of Theorem 3, we must define a new transition relation, which has a function analogous to the relation $\longrightarrow_{\mathrm{F}}$. A first attempt is to define $A \overset{\mu}{\Longrightarrow}_{\mathrm{F}} A'$ analogously with $A \overset{\mu}{\Longrightarrow}_{\mathrm{M}} A'$, namely to mean that there exists a sequence of transitions

$$A \xrightarrow{\tau}_{\mathrm{F}} \quad \cdots \quad \xrightarrow{\tau}_{\mathrm{F}} A_k \xrightarrow{\mu}_{\mathrm{F}} A_{k+1} \xrightarrow{\tau}_{\mathrm{F}} \quad \cdots \quad \xrightarrow{\tau}_{\mathrm{F}} A'$$

of transitions, all but one of which is labeled by τ. However, with these definitions the analogue of Theorem 3 fails. The reason for this is that if μ is an action of the form $a?v$, then the choice of the schematic name v depends on the names used by A_{k+1} and not on the names used by A'. Consider as an example:

$$\begin{aligned} A &= a?x\,.\,(\tau\,.\,\mathbf{0} \;+\; c!\mathtt{v}_1\,.\,\mathbf{0}) \;+\; a?x\,.\,\mathbf{0} \\ B &= a?x\,.\,(\tau\,.\,\mathbf{0} \;+\; c!\mathtt{v}_1\,.\,\mathbf{0}) \end{aligned}$$

Here A and B are observation equivalent. Since $\mathbf{0}$ does not use any schematic names we have, by the second summand in A, that $A \xrightarrow{a?\mathtt{v}_1}_{\mathrm{F}} \mathbf{0}$. But the first transition from B must be $B \xrightarrow{a?\mathtt{v}_2}_{\mathrm{F}} (\tau\,.\,\mathbf{0} \;+\; c!\mathtt{v}_1\,.\,\mathbf{0})$, and hence B can not do $B \overset{a?\mathtt{v}_1}{\Longrightarrow}_{\mathrm{F}} B'$ for any B'.

To eliminate this type of discrepancy between observation equivalent programs, we define a transition relation $\Longrightarrow_{\mathrm{G}}$, which differs from $\Longrightarrow_{\mathrm{F}}$ only in the choice of schematic names in some input transitions:

$$\frac{A \overset{\mu}{\Longrightarrow}_{\mathrm{F}} A'}{A \overset{\mu}{\Longrightarrow}_{\mathrm{G}} A'} \quad \text{for } \mu = a!u \text{ or } \mu = \tau$$

$$\frac{A \overset{a?v}{\Longrightarrow}_{\mathrm{F}} A'}{A \overset{a?v'}{\Longrightarrow}_{\mathrm{G}} A'[v'/v]} \quad \text{for } v' = \mathit{nextname}(A'[x/v])$$

Note that $\mathit{nextname}(A'[x/v])$ is the minimum (w.r.t. $<$) of v and $\mathit{nextname}(A')$. We now have the following general result:

Theorem 4 *Two programs are observation equivalent precisely if they are* $\Longrightarrow_{\mathrm{G}}$*-equivalent.*

Proof: see the Appendix. The important property of $\Longrightarrow_{\mathrm{G}}$ (which is not shared by $\Longrightarrow_{\mathrm{F}}$) is that if $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} A'$ then v' is used by A' for all $v' < v$. □

Note that since $ST(A, \longrightarrow_{\mathrm{F}})$ is finite and computable from A, then so is $ST(A, \Longrightarrow_{\mathrm{F}})$ and hence also $ST(A, \Longrightarrow_{\mathrm{G}})$. We thus have a decision procedure for $\Longrightarrow_{\mathrm{G}}$-equivalence, and hence for $\Longrightarrow_{\mathrm{M}}$-equivalence by Theorem 4.

5 The equivalence problem is NP-hard

In this section, we discuss the complexity of the equivalence problem. We take $\mathit{size}(A) + \mathit{size}(B)$ to be the size of the problem of deciding equivalence between A and B. From the proof of Proposition 2 it follows that our method in Section 4 can take space exponential in the size of the problem, since $ST(A, \longrightarrow_{\mathrm{F}})$ and $ST(B, \longrightarrow_{\mathrm{F}})$ may be exponentially large. A careful examination of the proof of Proposition 2 shows that the equivalence problem is exponential only in $\mathit{vsize}(A) + \mathit{vsize}(B)$. Thus, if there is a given bound on the number of free variables in any subterm of any program, then the equivalence problem can be solved in polynomial time, since $ST(A, \longrightarrow_{\mathrm{F}})$ can then be computed in polynomial time from A.

In this section, we shall prove NP-hardness for the problem of determining strong equivalence. In fact, we prove it NP-hard even for programs without τ-actions. It immediately follows that the problem of determining observation equivalence is also NP-hard, since strong equivalence and observation equivalence coincide on programs without τ:s.

Theorem 5 *The problem of determining whether two programs are M-equivalent is NP-hard.*

Proof: We prove the theorem through a reduction from the clique problem, which is NP-complete (see e.g. problem GT19 of [11]). The clique problem is to determine, given an undirected graph G with nodes $n_1, \ldots, n_K$ and edges E (a set of pairs of the form $\langle n_i, n_j \rangle$), and a positive integer $J \leq K$, whether G contains a clique (i.e. a maximally connected subgraph) of size J.

We shall use a variant of the clique problem, which we call the *restricted clique problem*, obtained by restricting the clique problem to graphs G in which each node has at least J neighbours. We prove that the restricted clique problem is NP-complete by a polynomial time reduction from the clique problem as follows. Consider an instance (i.e., a graph G

and an integer J) of the clique problem. It can be checked in polynomial time whether G contains a clique of size J which contains a node with less than J neighbours, because (1) no clique can contain a node with less than $J-1$ neighbours, and (2) to check whether there is a clique containing a node with $J-1$ neighbours, just check whether any such node together with its $J-1$ neighbours forms a clique. Let G^* be the graph obtained from G by removing all nodes with less than J neighbours and their incident edges. It follows that G contains a clique of size J if and only if either G contains a clique containing a node with less than J neighbours, or G^* contains a clique of size J. Hence we have reduced the clique problem to the restricted clique problem in polynomial time.

An outline of the reduction from the restricted clique problem to the problem of M-equivalence is as follows. Given a graph G with nodes $n_1, \ldots, n_K$, and an integer $J \leq K$, we use the identifiers $A, A_1, \ldots, A_K$, each with arity $K+1$, and $B, B_0, \ldots, B_J$, each with arity $J+1$. Intuitively, $A_1, \ldots, A_K$ will correspond to the nodes $n_1, \ldots, n_K$, whereas $B_1, \ldots, B_J$ are intended to correspond to the nodes in a clique. We shall define the identifiers in such a way that the programs $A(d_0, d_1, \ldots, d_J, d_0, \ldots, d_0)$ and $B(d_0, \ldots, d_J)$ are M-equivalent precisely when the graph G contains a clique of size J. The main idea is to define $A_1, \ldots, A_K$ and $B_0, \ldots, B_J$ so that for a fixed permutation $d'_0, \ldots, d'_K$ of $d_0, d_1, \ldots, d_J, d_0, \ldots, d_0$ we have that $A_k(d'_0, \ldots, d'_K) \sim_{\mathrm{M}} B_i(d_0, \ldots, d_J)$ if and only if the permutation $d'_0, \ldots, d'_K$ moves $d_1, \ldots, d_J$ to positions corresponding to the nodes in a clique in G in such a ways that A_k and B_i correspond to the same node. This is achieved by letting the transitions between different A_k's correspond to the edges of G, and letting the transitions between different B_i's correspond to the edges in a fully connected graph. The extra identifier B_0 is intended to be equivalent to those A_k that correspond to a node in G outside the clique. We finally define A so that it can permute its data values arbitrarily and thereafter transform into any of the A_k's. Analogously, we define B so that it can transform into any of the B_i's. If G contains a clique then $A(d_0, d_1, \ldots, d_J, d_0, \ldots, d_0)$ can simulate a move from $B(d_0, \ldots, d_J)$ to $B_i(d_0, \ldots, d_J)$ by first guessing a clique, i.e., permuting its data values so that $d_0, \ldots, d_J$ are moved to positions corresponding to the nodes in the clique, and thereafter transforming to a corresponding A_k.

We now give a precise definition of the reduction. Define

$$
\begin{aligned}
A(x_0, x_1, \ldots, x_K) &\Longleftarrow \sum_{i=1}^{K-1} A(x_0, x_1, \ldots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \ldots, x_K) + \\
&\quad + \sum_{k=1}^{K} a!x_0.A_k(x_0, x_1, \ldots, x_K) \\
A_k(x_0, \ldots, x_K) &\Longleftarrow \sum_{\langle n_k, n_l \rangle \in E} a!x_k.A_l(x_0, \ldots, x_K) + \\
&\quad + \sum_{l=1}^{K} a!x_0.A_l(x_0, \ldots, x_K) \\
B(x_0, x_1, \ldots, x_J) &\Longleftarrow A(x_0, x_1, \ldots, x_J, x_0, \ldots, x_0) + \\
&\quad \sum_{i=0}^{J} a!x_0.B_i(x_0, x_1, \ldots, x_J) \\
B_i(x_0, \ldots, x_J) &\Longleftarrow \sum_{j=0}^{J} a!x_i.B_j(x_0, \ldots, x_J) + \\
&\quad + \sum_{j=0}^{J} a!x_0.B_j(x_0, \ldots, x_J)
\end{aligned}
$$

It is obvious that these definitions can be obtained in polynomial time from a representation of the graph G. The behavior of the programs can intuitively be described as follows: The program $A(d_0, \ldots, d_K)$ can either permute the last K data values arbitrarily, or perform the action $a!d_0$ while becoming a program $A_k(d_0, \ldots, d_K)$ for $k = 1, \ldots, K$. The program $B(d_0, \ldots, d_J)$ can either behave like $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$ or perform the action $a!d_0$ and become a program $B_i(d_0, \ldots, d_J)$ for $i = 0, \ldots, J$. The program $A_k(d_0, \ldots, d_K)$ is able to perform the action $a!d_k$ and become the program $A_l(d_0, \ldots, d_K)$ iff G has an edge between n_k and n_l. The program $B_i(d_0, \ldots, d_J)$ where $i \geq 0$ is always able to perform the action $a!d_i$ and become the program $B_j(d_0, \ldots, d_J)$. In addition, $A_k(d_0, \ldots, d_K)$ can always perform the action $a!d_0$ and become the program $A_l(d_0, \ldots, d_K)$ for $k, l > 0$, and $B_i(d_0, \ldots, d_J)$ can always perform the action $a!d_0$ and become the program $B_j(d_0, \ldots, d_J)$ for $i, j \geq 0$.

Next we make a definition of the correspondence between the indices of the B_i's and the nodes of G that are in a clique. Given a $J+1$-tuple $d_0, \ldots, d_J$ of data values, a $K+1$-tuple $d'_0, \ldots, d'_K$ of data values, and an injective function $h : [1, \ldots, J] \to [1, \ldots, K]$ we say that $d_0, \ldots, d_J$ *codes a clique in* $d'_0, \ldots, d'_K$ *via* h precisely if

1. $d'_{h(i)} = d_i$ for $i = 1, \ldots, J$;
2. $d'_k = d_0$ for $k \in [0, \ldots, K]$ which is not in the range of h;
3. the nodes $n_{h(1)}, \ldots, n_{h(J)}$ constitute a clique in the graph G.

Intuitively, the range of the function h contains indices of nodes in a clique in G, and the data values $d'_{h(1)}, \ldots, d'_{h(J)}$ are the same as the data values $d_1, \ldots, d_J$. The data values d'_k that "are not in the clique" are the same as d_0.

Let $d_0, \ldots, d_J$ be distinct data values. We now make the following claim.

Claim: The programs $A(d_0, d_1, \ldots, d_J, d_0, \ldots, d_0)$ and $B(d_0, \ldots, d_J)$ are M-equivalent precisely when the graph G contains a clique of size J.

To prove one direction, assume that G contains a clique of size J. Then there is a permutation $d'_0, \ldots, d'_K$ of $d_0, d_1, \ldots, d_J, d_0, \ldots, d_0$ and an an injective function $h : [1, \ldots, J] \to [1, \ldots, K]$ such that $d_0, \ldots, d_J$ codes a clique in $d'_0, \ldots, d'_K$ via h. From now on, we let $d'_0, \ldots, d'_K$ be one fixed such permutation. Define the relation $\sim$ to consist of

$$\begin{array}{lcl} A(d_0, \ldots, d_J, d_0, \ldots, d_0) & \sim & B(d_0, \ldots, d_J) \\ A_{h(i)}(d'_0, \ldots, d'_K) & \sim & B_i(d_0, \ldots, d_J) \text{ for } i \in [1, \ldots, J] \\ A_k(d'_0, \ldots, d'_K) & \sim & B_0(d_0, \ldots, d_J) \text{ for } k \notin \mathit{range}(h) \end{array}$$

We shall prove that $\sim \cup \mathcal{I}$ is a $\longrightarrow_{\mathrm{M}}$-bisimulation, where $\mathcal{I}$ is the identity relation. We consider the three cases of the above definition of $\sim$.

$A(d_0, \ldots, d_J, d_0, \ldots, d_0) \sim B(d_0, \ldots, d_J)$: Since $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$ is a summand of $B(d_0, \ldots, d_J)$, any transition from $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$ can be simulated by a transition from $B(d_0, \ldots, d_J)$. Also, any transition from $B(d_0, \ldots, d_J)$ which is derived from the summand $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$ can of course be simulated by a transition from $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$. Finally, for any i where $1 \leq i \leq J$, the transition $B(d_0, \ldots, d_J) \xrightarrow{a!d_0}_{\mathrm{M}} B_i(d_0, \ldots, d_J)$ can, according to the definitions, be simulated by the transition $A(d_0, \ldots, d_J, d_0, \ldots, d_0) \xrightarrow{a!d_0}_{\mathrm{M}} A_{h(i)}(d'_0, \ldots, d'_K)$. This transition is possible since $A(d_0, \ldots, d_J, d_0, \ldots, d_0)$ can permute its data values to become $A(d'_0, \ldots, d'_K)$.

$A_{h(i)}(d'_0, \ldots, d'_K) \sim B_i(d_0, \ldots, d_J)$ for $1 \leq i \leq J$: Here we consider two cases:

1. A transition $A_{h(i)}(d'_0, \ldots, d'_K) \xrightarrow{a!d'_{h(i)}}_{\mathrm{M}} A_{h(j)}(d'_0, \ldots, d'_K)$, where $1 \leq j \leq J$, bisimulates with the transition $B_i(d_0, \ldots, d_J) \xrightarrow{a!d_i}_{\mathrm{M}} B_j(d_0, \ldots, d_J)$ because $\langle n_{h(i)}, n_{h(j)} \rangle$ must be an edge in E since $n_{h(i)}$ and $n_{h(j)}$ are both in the clique. The same correspondence holds when $a!d'_{h(i)}$ and $a!d_i$ are both replaced by $a!d_0$.
2. A transition $A_{h(i)}(d'_0, \ldots, d'_K) \xrightarrow{a!d'_{h(i)}}_{\mathrm{M}} A_k(d'_0, \ldots, d'_K)$, where $k \notin \mathit{range}(h)$, is simulated by the transition $B_i(d_0, \ldots, d_J) \xrightarrow{a!d_i}_{\mathrm{M}} B_0(d_0, \ldots, d_J)$. Vice versa, the transition $B_i(d_0, \ldots, d_J) \xrightarrow{a!d_i}_{\mathrm{M}} B_0(d_0, \ldots, d_J)$ can always be simulated by a transition $A_{h(i)}(d'_0, \ldots, d'_K) \xrightarrow{a!d'_{h(i)}}_{\mathrm{M}} A_k(d'_0, \ldots, d'_K)$ where $k \notin \mathit{range}(h)$, because by the requirement that each node in G have at least J neighbours, the node $n_{h(i)}$ must have a neighbour n_k for which $k \notin \mathit{range}(h)$. As in the preceding case, the same correspondence holds when $a!d'_{h(i)}$ and $a!d_i$ are both replaced by $a!d_0$.

$A_k(d'_0, \ldots, d'_K) \sim B_0(d_0, \ldots, d_J)$ for $k \notin \mathit{range}(h)$: Any transition from these programs is labeled by $a!d_0$. In analogy with the previous case, we consider two cases

1. A transition of form $A_k(d'_0, \ldots, d'_K) \xrightarrow{a!d_0}_{\mathrm{M}} A_{h(j)}(d'_0, \ldots, d'_K)$ with $1 \leq j \leq J$ bisimulates with the transition $B_0(d_0, \ldots, d_J) \xrightarrow{a!d_0}_{\mathrm{M}} B_j(d_0, \ldots, d_J)$.

2. A transition of form $A_k(d'_0,\ldots,d'_K) \xrightarrow{a!d_0}_{\mathrm{M}} A_l(d'_0,\ldots,d'_K)$, with $l \notin \mathit{range}(h)$, of which there is one for each $l \notin \mathit{range}(h)$, bisimulates with the transition $B_0(d_0,\ldots,d_J) \xrightarrow{a!d_0}_{\mathrm{M}} B_0(d_0,\ldots,d_J)$.

To prove the claim in the other direction, assume that the programs $A(d_0,\ldots,d_J,d_0,\ldots,d_0)$ and $B(d_0,\ldots,d_J)$ are M-equivalent. We shall prove that G contains a clique of size J.

Consider an $i \in [1,\ldots,J]$. Since the program $B(d_0,\ldots,d_J)$ is able to perform the transition

$$B(d_0,\ldots,d_J) \xrightarrow{a!d_0}_{\mathrm{M}} B_i(d_0,\ldots,d_J) \ \ ,$$

there must be a $k \in [1,\ldots,K]$ such that the program $A(d_0,\ldots,d_J,d_0,\ldots,d_0)$ can simulate this transition by a transition of the form

$$A(d_0,\ldots,d_J,d_0,\ldots,d_0) \xrightarrow{a!d_0}_{\mathrm{M}} A_k(d'_0,\ldots,d'_K)$$

where $A_k(d'_0,\ldots,d'_K) \sim_{\mathrm{M}} B_i(d_0,\ldots,d_J)$ and where $d'_0,\ldots,d'_K$ is a permutation of $d_0,\ldots,d_J,d_0,\ldots,d_0$. Define the function $h : [1,\ldots,J] \to [1,\ldots,K]$ by letting $h(j)$ be the unique l such that $d'_l = d_j$. It follows from the distinctness of $d_0,\ldots,d_J$ that h is a well-defined function and injective. We shall demonstrate that $n_{h(1)},\ldots,n_{h(J)}$ constitute a clique in G. For $j \in [1,\ldots,J]$ we have that if $A_l(d'_0,\ldots,d'_K)$ can perform the action $a!d_j$ then $l = h(j)$. It follows that $A_l(d'_0,\ldots,d'_K) \sim_{\mathrm{M}} B_j(d_0,\ldots,d_J)$ implies $l = h(j)$; in particular we have $k = h(i)$. Now observe that since $A_{h(i)}(d'_0,\ldots,d'_K) \sim_{\mathrm{M}} B_i(d_0,\ldots,d_J)$ it must for each $j \in [1,\ldots,J]$ be possible to simulate the transition $B_i(d_0,\ldots,d_J) \xrightarrow{a!d_0}_{\mathrm{M}} B_j(d_0,\ldots,d_J)$ by a transition of the form $A_k(d'_0,\ldots,d'_K) \xrightarrow{a!d_0}_{\mathrm{M}} A_l(d'_0,\ldots,d'_K)$ with $B_j(d_0,\ldots,d_J) \sim_{\mathrm{M}} A_l(d'_0,\ldots,d'_K)$, implying that $l = h(j)$. Continuing in the same way, note that for each $j' \in [1,\ldots,J]$ it must be possible to simulate the transition $B_j(d_0,\ldots,d_J) \xrightarrow{a!d_j}_{\mathrm{M}} B_{j'}(d_0,\ldots,d_J)$ by the transition $A_l(d'_0,\ldots,d'_K) \xrightarrow{a!d_j}_{\mathrm{M}} A_{l'}(d'_0,\ldots,d'_K)$ with $B_{j'}(d_0,\ldots,d_J) \sim_{\mathrm{M}} A_{l'}(d'_0,\ldots,d'_K)$, implying that $l' = h(j')$. This implies that for all $j, j' \in [1,\ldots,J]$ there is an edge in G between $n_{h(j)}$ and $n_{h(j')}$. It follows that the nodes $n_{h(1)},\ldots,n_{h(J)}$ constitute a clique in G. $\square$

6 Parallelism

In this section we indicate how our results are extended to parallel programs. We follow [18] and define a binary operator " | " (parallel composition) together with a family of unary postfix operators "$\backslash a$" (restriction on the port a). The operational semantics in [18] amounts to the following extension of the definition of $\longrightarrow_{\mathrm{M}}$:

$$\mathrm{PAR_M}: \quad \frac{A \xrightarrow{\mu}_{\mathrm{M}} A'}{A \mid B \xrightarrow{\mu}_{\mathrm{M}} A' \mid B} \qquad\qquad \mathrm{COM_M}: \quad \frac{A \xrightarrow{a?u}_{\mathrm{M}} A' \quad B \xrightarrow{a!u}_{\mathrm{M}} B'}{A \mid B \xrightarrow{\tau}_{\mathrm{M}} A' \mid B'}$$

$$\mathrm{RES_M}: \quad \frac{A \xrightarrow{\mu}_{\mathrm{M}} A'}{A\backslash a \xrightarrow{\mu}_{\mathrm{M}} A'\backslash a} \quad \text{for } \mu \neq a?u,\ \mu \neq a!u$$

The rules for $\mid$ additionally have a symmetric form differing in that the operands of $\mid$ are interchanged, thus $A \mid B$ has the same transitions as $B \mid A$: either one of A and B performs a transition in isolation ($\textsc{par}_{\mathrm{M}}$), or they cooperate in a communication resulting in a τ-transition ($\textsc{com}_{\mathrm{M}}$).

We apply the same ideas as in Section 4 by extending the definition of $\longrightarrow_{\mathrm{F}}$ to the new operators. However, some extra care must be taken here for the results to carry over. In the $\textsc{com}_{\mathrm{M}}$ rule, the premises require identical data values or schematic names in the input and output transitions. For the definition of $\longrightarrow_{\mathrm{M}}$ this is sufficient since, according to the $\textsc{in}_{\mathrm{M}}$ rule, a program that can do an input transition with u can also do an input transition with any other data value or schematic name, and can thus cooperate with any possible output action on the correct port. But the corresponding rule $\textsc{in}_{\mathrm{F}}$ allows only one name. Clearly, the expected result of a communication will be that this name is substituted by whatever is output by the peer program, i.e. we have the rule

$$\textsc{com}_{\mathrm{F}}: \quad \frac{A \xrightarrow{a?v}_{\mathrm{F}} A' \quad B \xrightarrow{a!u}_{\mathrm{F}} B'}{A \mid B \xrightarrow{\tau}_{\mathrm{F}} A'[u/v] \mid B'}$$

Another modification is required for the $\textsc{par}$-rule where one program performs an input transition in isolation. A simple-minded attempt would be to define a $\textsc{par}_{\mathrm{F}}$ rule by which it is possible to derive $(A \mid B) \xrightarrow{a?v}_{\mathrm{F}} (A' \mid B)$ from $A \xrightarrow{a?v}_{\mathrm{F}} A'$. But then, the schematic name v that is introduced into A' may clash with an existing schematic name in B, and this v in B may later be substituted by a subsequent application of the $\textsc{com}_{\mathrm{F}}$ rule. Clearly, this is incorrect: the v in B is only accidentally related to the v in the input transition. To remedy this situation we adopt the following rule, which changes the schematic names of some input actions in order to avoid conflicts:

$$\textsc{pari}_{\mathrm{F}}: \quad \frac{A \xrightarrow{a?v}_{\mathrm{F}} A' \quad v' = \mathit{nextname}(A'[x/v] \mid B)}{A \mid B \xrightarrow{a?v'}_{\mathrm{F}} A'[v'/v] \mid B}$$

The schematic name v in the input action is here changed to v'. The point with choosing $v' = \mathit{nextname}(A'[x/v] \mid B)$ is to ensure that v' can not be confused with any used name in A' or B, except possibly with v if v does not occur in B.

The remaining rules for $\longrightarrow_{\mathrm{F}}$ are unproblematic:

$$\textsc{paro}_{\mathrm{F}}: \quad \frac{A \xrightarrow{\mu}_{\mathrm{F}} A'}{A \mid B \xrightarrow{\mu}_{\mathrm{F}} A' \mid B} \quad \text{for } \mu \neq a?v$$

$$\textsc{res}_{\mathrm{F}}: \quad \frac{A \xrightarrow{\mu}_{\mathrm{F}} A'}{A\backslash a \xrightarrow{\mu}_{\mathrm{F}} A'\backslash a} \quad \text{for } \mu \neq a?v,\ \mu \neq a!u$$

In addition, the rules for $\mid$ have a symmetric form. As in Section 4.3 we also define $\Longrightarrow_{\mathrm{F}}$ and $\Longrightarrow_{\mathrm{G}}$ in terms of the thus extended $\longrightarrow_{\mathrm{F}}$. With these definitions the results carry over to the language with parallel programs:

Theorem 6 *Theorems 3 and 4 still hold in the language extended by parallel composition and restriction.*

Proof: See the Appendix. □

It should be noted that Proposition 2 does not necessarily carry over. An infinite state-space may arise from the use of parallelism inside a recursive definition, resulting in a program that grows dynamically during execution. An example of this is the definition of an "unbounded bag":

$$\mathtt{Bag} \Longleftarrow \mathtt{in?x}\,.\,(\mathtt{out!x}\,.\,\mathbf{0} \mid \mathtt{Bag})$$

However, many commonly occurring programs have a static process structure and can be described without use of parallelism inside a recursive definition. For such programs Proposition 2 carries over: each state of $A|B$ is an instance of a parallel composition of a state of A with a state of B, and if $\mathtt{v}_i$ is a schematic name in this instance then $i \leq \mathit{vsize}(A) + \mathit{vsize}(B)$. Also, for such programs $\mathit{nextname}(A)$ and hence also $ST(A, \longrightarrow_{\mathrm{F}})$ is computable from A (although not in polynomial time; the size of $ST(A, \longrightarrow_{\mathrm{F}})$ is at worst exponential in the size of A). So the equivalence problems are decidable for this class of programs. The most straightforward way to compute $\mathit{nextname}(A)$ is to use an expansion law which removes top-level parallel compositions, and then proceed as in the Appendix — we will not elaborate on this here.

7 A Simple Application

In this section we indicate verifications which are possible in our formalism, and which would be impossible with existing fully automated techniques.

One major application of our techniques is in the area of verification of communication protocols. Consider a data transfer protocol, e.g. the Alternating-Bit Protocol, which should transfer messages from one user to another while preserving message order. Such a protocol can be proven correct by establishing an equivalence between the protocol and a (FIFO) buffer. Previous such verifications ([2, 14, 15, 21, 22, 24]) have either been performed manually, or have only considered the synchronization properties of the protocol, while ignoring the transfer of data. A verification which ignores data transfer does not determine whether the protocol transfers messages like a bag (not respecting their order) or like a buffer. With our techniques, it is possible to make such a distinction in an automated verification.

We shall as a simple example sketch how one could distinguish between a buffer and a bag, both of capacity two. We write $\approx$ for observation equivalence, and assume that $\mathtt{x}$ and $\mathtt{y}$ are variables. Define a *buffer* $\mathtt{B1}_{ab}$ of capacity one over two different ports a and b as:

$$\mathtt{B1}_{ab} \Longleftarrow a?\mathtt{x}\,.\,b!\mathtt{x}\,.\,\mathtt{B1}_{ab}$$

Correspondingly, a buffer $\mathtt{B2}_{ab}$ of capacity two is:

$$
\begin{array}{lcl}
\mathtt{B2}_{ab} & \Longleftarrow & a?\mathtt{x}\,.\,\mathtt{B2}'_{ab}(\mathtt{x}) \\
\mathtt{B2}'_{ab}(\mathtt{x}) & \Longleftarrow & b!\mathtt{x}\,.\,\mathtt{B2}_{ab} + a?\mathtt{y}\,.\,\mathtt{B2}''_{ab}(\mathtt{x},\mathtt{y}) \\
\mathtt{B2}''_{ab}(\mathtt{x},\mathtt{y}) & \Longleftarrow & b!\mathtt{x}\,.\,\mathtt{B2}'_{ab}(\mathtt{y})
\end{array}
$$

A *bag* $\mathtt{BG}_{ab}$ of capacity two is like a buffer but does not preserve the order of elements:

$$
\begin{array}{lcl}
\mathtt{BG}_{ab} & \Longleftarrow & a?\mathtt{x}\,.\,\mathtt{BG}'_{ab}(\mathtt{x}) \\
\mathtt{BG}'_{ab}(\mathtt{x}) & \Longleftarrow & b!\mathtt{x}\,.\,\mathtt{BG}_{ab} + a?\mathtt{y}\,.\,\mathtt{BG}''_{ab}(\mathtt{x},\mathtt{y}) \\
\mathtt{BG}''_{ab}(\mathtt{x},\mathtt{y}) & \Longleftarrow & b!\mathtt{x}\,.\,\mathtt{BG}'_{ab}(\mathtt{y}) + b!\mathtt{y}\,.\,\mathtt{BG}'_{ab}(\mathtt{x})
\end{array}
$$

It is now straightforward to prove that two buffers of capacity one can be connected to form a buffer of capacity two:

$$(\mathtt{B1}_{ac} \mid \mathtt{B1}_{cb})\backslash b \;\approx\; \mathtt{B2}_{ab}$$

and that connected in a different way they form a bag:

$$\mathtt{B1}_{ab} \mid \mathtt{B1}_{ab} \;\approx\; \mathtt{BG}_{ab}$$

These proofs can be performed mechanically using the alternative transition relation $\Longrightarrow_{\mathrm{G}}$, which makes all involved programs finite-state. For instance, $\mathtt{BG}_{ab}$ has five states:

$$\mathtt{BG} \;\;,\;\; \mathtt{BG}'(\mathtt{v}_1) \;\;,\;\; \mathtt{BG}''(\mathtt{v}_1,\mathtt{v}_2) \;\;,\;\; \mathtt{BG}'(\mathtt{v}_2) \;\;,\;\; \mathtt{BG}''(\mathtt{v}_2,\mathtt{v}_1)$$

Note that the programs are not finite-state with respect to $\Longrightarrow_{\mathrm{M}}$. Therefore, a traditional verification of similar programs would mechanically verify only the synchronization properties, disregarding the distinction between different data values. However, such a simplification makes it impossible to distinguish between a buffer and a bag.

Our algorithm has been implemented ([16]) as an experimental extension of the Concurrency Workbench ([9]). It has been successfully applied to the verification of a sublayer of the CSMA/CD-protocol.

8 Conclusion

We have presented an algorithm to determine bisimulation equivalences on nondeterministic and parallel programs, which operate on infinite data domains but which do not perform any computations on the data.

An alternative and perhaps more straightforward algorithm could be to let the programs operate on a finite but sufficiently large data domain. Such an algorithm would presumably be less efficient in practice, since each input construct then gives rise to many input transitions. For instance, when proving that two FIFO buffers, with a capacity to store n data items, are equivalent, the alternative algorithm would generate a state space whose size is exponential in n, whereas our algorithm would generate a state space which is quadratic in n.

Our main idea is related to symbolic execution: by using schematic names as representatives of data values received in input actions, the equivalence problems are reduced to equivalence of finite-state programs. We conjecture that the same idea would apply equally well to other equivalences such as testing-, failure- or trace equivalence. We also conjecture that a similar symbolic execution could be used to obtain decision procedures for restricted versions of first-order modal logics such as HML or CTL.

In this paper, we have assumed that the set $\mathcal{D}$ of data values is infinite. In fact, our results apply to a finite $\mathcal{D}$, as long as its size is larger than the maximum number of free variables in any subterm of the involved programs; if $\mathcal{D}$ is too small, one can always use the decision procedure for $\longrightarrow_{\mathrm{M}}$-equivalence directly, since $ST(A, \longrightarrow_{\mathrm{M}})$ then becomes finite.

It would be interesting to extend the result to programs where expressions with functions are allowed in output constructs. It does not seem possible to add even simple (interpreted) function symbols to the language without invalidating our results, but it seems plausible that an extension of the ideas in this paper would be able to handle a restricted class of function symbols.

Acknowledgments

We are grateful to Robin Milner for comments on an earlier version of the manuscript, and to Johan Håstad for discussions on the complexity problem in section 5. This research has been partly funded by Swedish Telecom under the Prometheus project, and by the Swedish Board of Technical Development (STU) within the Esprit Basic Research Actions CONCUR and SPEC.

References

[1] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2,3):225–241, 1987.

[2] J. Bergstra and J. Klop. Verification of an alternating bit protocol by means of process algebra. In Bibel and Jantke, editors, *Proc. Math. Methods of Spec. and Synthesis of Software Systems '85, Math. Research 31*, pages 9–23. Akademie-Verlag, 1986.

[3] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced: Preliminary report. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 229–239, 1988.

[4] D. Brand and W. H. Joyner. Verification of protocols using symbolic execution. *Computer Networks*, 2(4-5):351–360, 1978.

[5] D. Brand and W. H. Joyner. Verification of HDLC. *IEEE Transactions on Communications*, COM-30(5):1136–1142, 1982.

[6] S. Brookes, C.A.R. Hoare, and W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logics specification: A practical approach. In *Proc. 10th ACM PoPL*, pages 117–126, Austin, Texas, 1983.

[8] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. In Sifakis, editor, *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 11–23. Springer Verlag, 1989.

[9] R. Cleaveland, J. Parrow, and B. Steffen. A semantics-based verification tool for finite-state systems. In Brinksma, Scollo, and Vissers, editors, *Proc.* 9^{th} *IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification, 1989*, pages 287–302, Univ. of Twente, The Netherlands. North-Holland. To appear in *ACM Transactions on Programming Languages and Systems.*

[10] T. W. Doeppner and A. Giacalone. A formal description of the UNIX operating system. In *Proc.* 2^{nd} *ACM Symp. on Principles of Distributed Computing*, pages 241–253, 1983.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-completeness.* W.H. Freeman and Company, New York, 1979.

[12] N. D. Jones and S. S. Muchnick. Even simple programs are hard to analyse. *J. ACM*, 24(2):338–350, April 1977.

[13] P.C. Kanellakis and S.A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[14] C. Koomen. Algebraic specification and verification of protocols. *Science of Computer Programming*, 5(1):1–36, 1985.

[15] K. G. Larsen and R. Milner. Verifying a protocol using relativized bisimulation. In *Proc. ICALP '87, LNCS 267*, volume 267 of *Lecture Notes of Computer Science.* Springer Verlag, 1987.

[16] C.-H. Lee. Implementering av CCS med värdeöverföring (in Swedish). Technical Report T89002, SICS, 1989.

[17] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[19] R. de Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[20] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[21] J. Parrow. Verifying a CSMA/CD-Protocol with CCS. In *Protocol Specification, Testing, and Verification VIII (1984)*. North-Holland, 1988.

[22] S.A. Smolka, A.J. Frank, and S.K. Debray. Testing protocol robustness the CCS way. In *Protocol Specification, Testing, and Verification IV (1984)*, pages 93–108, 1985. North-Holland.

[23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[24] D. Vergamini. Verification by means of observation equivalence on automata. Technical Report 501, INRIA, Sophia Antipolis, 1986.

[25] P. Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. 13^{th} ACM Symp. on Principles of Programming Languages*, pages 184–193, Jan. 1986.

Appendix

We here give the proofs omitted from the main text.

Proposition 1 *$nextname(A)$ is computable in polynomial time.*

Proof: Let A be a schematic behaviour expression. For each subterm B of A define $Used(B)$ to be the least set of variables and schematic names which satisfy the following recursive equations:

$$
\begin{array}{lcl}
Used(a?x\,.\,C) & = & Used(C) - \{x\} \\
Used(a!x\,.\,C) & = & Used(C) \cup \{x\} \\
Used(a!v\,.\,C) & = & Used(C) \cup \{v\} \\
Used(a!d\,.\,C) & = & Used(C) \\
Used(\sum_i (C_i)) & = & \bigcup_i Used(C_i) \\
Used(Id(\tilde{e})) & = & (Used(C))[\tilde{e}/\tilde{x}] - \mathcal{D} \quad \text{when } Id(\tilde{x}) \Longleftarrow C
\end{array}
$$

We here write $\tilde{e}$ for $e_1, \ldots, e_n$ and $\tilde{x}$ for $x_1, \ldots, x_n$, and in the last equation $[\tilde{e}/\tilde{x}]$ means a substitution which simultaneously replaces each x_i with e_i in the set $Used(C)$. Recall that the x_i, as formal parameters to Id, are required to be distinct, and that e ranges over data values, schematic names, and variables. Since $Used(B)$ does not contain data values, the set of data values $\mathcal{D}$ is subtracted in the last equation.

We can write the recursive equations as $Used = \mathcal{U}(Used)$ where the functor $\mathcal{U}$ is as determined by the equations. Then $\mathcal{U}$ is monotonic and the least solution can thus be computed in the following standard way: begin with the approximation $U_0(B) = \emptyset$ for all B. Compute $U_i = \mathcal{U}(U_{i-1})$ until convergence, i.e. until $U_k = U_{k-1}$. Then $U_k = Used$.

We will now motivate that this calculation can be done in polynomial time. Each U_i is defined on the subterms of A, and the number of subterms is linearly bounded by the size of A. The computation of a particular $U_i(B)$ given U_{i-1} is, at worst, linearly bounded by the maximal length of the formal parameter lists, which again is linearly bounded by A. Thus, the computation of U_i from U_{i-1} is polynomial in A. The number of different possible U_i is also polynomial in A, since $\mathcal{U}$ is monotonic and each $U_i(B)$ can at most contain all names and variables in A. It follows that $Used$ can be calculated in polynomial time.

We finally prove that for a program A, it holds that v is used by A iff $v \in Used(A)$. In the course of this proof we will use the lemma that $Used(A[\tilde{e}/\tilde{x}]) = Used(A)[\tilde{e}/\tilde{x}] - \mathcal{D}$. (To prove this lemma, prove by induction on k that $U_k(A[\tilde{e}/\tilde{x}]) = U_k(A)[\tilde{e}/\tilde{x}] - \mathcal{D}$; the inductive step follows immediately from the definition of U_k.) For direction $\Rightarrow$, assume that v is used by A, i.e. that there is a sequence of transitions

$$A = A_1 \xrightarrow{\mu_1}_{\mathrm{M}} \quad \ldots \quad \xrightarrow{\mu_{k-1}}_{\mathrm{M}} A_k \xrightarrow{a!v}_{\mathrm{M}} A_{k+1}$$

starting with A, in which the last action is $a!v$, and in which no input transition contains the name v, i.e. is of the form $A_j \xrightarrow{b?v}_{\mathrm{M}} A_{j+1}$. We will proceed by induction on the length of the shortest inference of such a transition sequence. The base case is that only one step is needed; then $k = 1$ and $A = a!v\,.\,A'$, and this implies $v \in U_1(A) \subseteq \mathit{Used}(A)$. For the inductive step, consider the last step in the inference of $A_1 \xrightarrow{\mu_1}_{\mathrm{M}} A_2$. We get four cases from the four possible rules:

1. The last step was inferred with $\textsc{in}_{\mathrm{M}}$. Then $A_1 = a?x.A'$ and $A'[u/x] = A_2$ for some A' and $u \neq v$. Clearly v is used by A_2 and this can be demonstrated with a shorter inference, so by induction $v \in \mathit{Used}(A_2)$. Then, since $u \neq v$, it also holds that $v \in \mathit{Used}(A')$, which (since $x \neq v$) implies $v \in \mathit{Used}(A_1)$.

2. The last step was inferred with $\textsc{out}_{\mathrm{M}}$. This case is similar to (and simpler than) case 1 above.

3. The last step was inferred with $\textsc{sum}_{\mathrm{M}}$. Then $A_1 = \sum_k B_k$ and for some j it holds $B_j \xrightarrow{\mu_1}_{\mathrm{M}} A_2$. Clearly, by a shorter inference v is used by B_j so by induction $v \in \mathit{Used}(B_j)$, whence $v \in \mathit{Used}(A_1)$.

4. The last step was inferred with $\textsc{id}_{\mathrm{M}}$. Then $A_1 = \mathit{Id}(\tilde{e})$. Let $\mathit{Id}(\tilde{x}) \Longleftarrow B$. Then $B[\tilde{e}/\tilde{x}] \xrightarrow{\mu_1}_{\mathrm{M}} A_2$. Clearly then by a shorter inference v is used by $B[\tilde{e}/\tilde{x}]$ so by induction $v \in \mathit{Used}(B[\tilde{e}/\tilde{x}])$. But $\mathit{Used}(B[\tilde{e}/\tilde{x}]) = \mathit{Used}(B)[\tilde{e}/\tilde{x}] - \mathcal{D}$, which by the definition of Used is the same as $\mathit{Used}(\mathit{Id}(\tilde{e}))$, so $v \in \mathit{Used}(A_1)$.

For the direction $\Leftarrow$ assume that $v \in \mathit{Used}(A)$. Then, in the terminology of our calculation of Used above, $v \in U_k(A)$ for some k. We proceed by induction on k. The case $k = 0$ is vacuous. For the inductive step we get different cases depending on the form of A.

1. $A = a?x.C$. Then $v \in U_{k-1}(C)$. So by induction v is used by C and thus also by $C[d/x]$ for some d. Then, since $A \xrightarrow{a?d} C[d/x]$, it follows that v is used by A.

2. $A = a!v.C$. Then it directly follows that v is used by A.

3. $A = a!u.C$ for $u \neq v$. Then by an argument similar to case 1 above, v is used by A.

4. $A = \sum_j C_j$. Then $v \in U_{k-1}(C_j)$ for some j. By induction v is used by C_j, so (by the $\textsc{sum}_{\mathrm{M}}$ rule) v is used by A.

5. $A = \mathit{Id}(\tilde{e})$. Let $\mathit{Id}(\tilde{x}) \Longleftarrow C$. Then $v \in (U_{k-1}(C))[\tilde{e}/\tilde{x}]$ which implies $v \in U_{k-1}(C[\tilde{e}/\tilde{x}])$. By induction v is used by $C[\tilde{e}/\tilde{x}]$. So by the $\textsc{id}_{\mathrm{M}}$ rule v is also used by A.

In summary we have now proved that the set of used names of a program can be computed in polynomial time. It then follows directly from the definition of $\mathit{nextname}$ that $\mathit{nextname}(A)$ can also be computed in polynomial time. This concludes the proof of Proposition 1. $\square$

We now continue to prove Theorems 3, 4 and 6. In the course of these proofs it is convenient to use the following definitions: let $\mathit{next}_v(A)$ mean the minimum (w.r.t. $<$) of v and $\mathit{nextname}(A)$. Thus, in particular $v = \mathit{next}_v(A)$ holds if and only if all $v' < v$ are used by A. We will write $\mathcal{V}(A)$ for the set of names used by A.

Say that a name or data value u is *fresh* for A if it does not occur syntactically in A nor in any definitions of identifiers. In a proof we will sometimes say "Assume a fresh u ..." to mean that u is fresh for all programs considered so far in the proof. Since there are only finitely many identifier definitions, such fresh u:s always exist.

Define $A \simeq B$ to mean that there exist substitutions $[\widetilde{u}/\widetilde{v}]$ and $[\widetilde{u'}/\widetilde{v'}]$, where no name in $\widetilde{v}$ is used in A and no name in $\widetilde{v'}$ is used in B, such that $A[\widetilde{u}/\widetilde{v}] = B[\widetilde{u'}/\widetilde{v'}]$. In other words, $A \simeq B$ if A and B can be made syntactically identical by substitutions of names which are not used in A and B respectively. Thus, if $v \notin \mathcal{V}(A)$ we have e.g. $A \simeq A[u/v]$ and $A[v/v'][u/v] \simeq A[u/v']$. We will implicitly use the fact that $\simeq$ is preserved by all operators, for example that $A \simeq B$ implies $A|C \simeq B|C$ etc.

Relation composition is written in the ordinary way. For instance, $A \xrightarrow{\mu}_{\mathrm{M}} \circ \simeq C$ means that there exists a B such that $A \xrightarrow{\mu}_{\mathrm{M}} B$ and $B \simeq C$.

Proof of Theorem 3

The proof of Theorem 3 relies on the following lemmas:

Lemma 7 If $v' \geq v$ then $next_v(A) = next_{v'}(A[v'/v])$.

Lemma 8 If $A \sim_{\mathrm{M}} B$ then $\mathcal{V}(A) = \mathcal{V}(B)$.

Lemma 9 $\simeq$ is a $\longrightarrow_{\mathrm{M}}$-bisimulation and a $\longrightarrow_{\mathrm{F}}$-bisimulation; hence if $A \simeq B$ then $A \sim_{\mathrm{M}} B$ and $A \sim_{\mathrm{F}} B$.

Lemma 10

a) $A \xrightarrow{a!u}_{\mathrm{M}} A'$ iff $A \xrightarrow{a!u}_{\mathrm{F}} A'$.

b) If $A \xrightarrow{a?v}_{\mathrm{F}} A'$ then $v = next_v(A')$, and for all u it holds $A \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$.

c) If $u \in \mathcal{V}$ and u is fresh for A, and $A \xrightarrow{a?u}_{\mathrm{M}} A'$, then for $v = next_u(A')$ it holds $A \xrightarrow{a?v}_{\mathrm{F}} \circ \simeq A'[v/u]$.

d) If $A \xrightarrow{a?u}_{\mathrm{M}} A'$, then for some fresh v it holds $A \xrightarrow{a?v}_{\mathrm{M}} A''$ with $A' = A''[u/v]$.

Lemma 11 If $A \sim_{\mathrm{M}} B$ then $A[u/v] \sim_{\mathrm{M}} B[u/v]$, and if $A \sim_{\mathrm{F}} B$ then $A[u/v] \sim_{\mathrm{F}} B[u/v]$ (i.e. $\sim_{\mathrm{M}}$ and $\sim_{\mathrm{F}}$ are preserved by substitution).

Lemma 10 is the key lemma which relates $\longrightarrow_{\mathrm{M}}$ with $\longrightarrow_{\mathrm{F}}$. The main idea of the proof of Theorem 3 is to show that $\sim_{\mathrm{M}}$ is a $\longrightarrow_{\mathrm{F}}$-bisimulation and vice versa. It then follows immediately from the definitions that $\sim_{\mathrm{M}}$ and $\sim_{\mathrm{F}}$ coincide.

Theorem 3 *Two schematic programs are* M*-equivalent precisely if they are* F*-equivalent.*

Proof: We first prove that $A \sim_{\mathrm{M}} B$ implies $A \sim_{\mathrm{F}} B$, by proving that $\sim_{\mathrm{M}}$ is a $\longrightarrow_{\mathrm{F}}$-bisimulation. So assume that $A \sim_{\mathrm{M}} B$ and $A \xrightarrow{\mu}_{\mathrm{F}} A'$. We must prove that $B \xrightarrow{\mu}_{\mathrm{F}} B'$ for some B' such that $A' \sim_{\mathrm{M}} B'$. If μ is an output action (of the form $a!u$) then this follows immediately from Lemma 10a and the fact that $\sim_{\mathrm{M}}$ is a $\longrightarrow_{\mathrm{M}}$-bisimulation. So assume that $\mu = a?v$. By Lemma 10b we infer that $v = \mathit{next}_v(A')$ and that for all u it holds $A \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$. Choose a fresh name u such that $u > v$; in particular we have $A \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$ for this u. By $A \sim_{\mathrm{M}} B$ we infer that for some B'' it holds $B \xrightarrow{a?u}_{\mathrm{M}} B''$ with $A'[u/v] \simeq \circ \sim_{\mathrm{M}} B''$, which by Lemma 9 implies $A'[u/v] \sim_{\mathrm{M}} B''$. From Lemma 10c we then get that there is a program B' such that $B \xrightarrow{a?v'}_{\mathrm{F}} B' \simeq B''[v'/u]$ for $v' = \mathit{next}_u(B'')$. Now by Lemma 8 we get that $v' = \mathit{next}_u(A'[u/v])$, and by Lemma 7 conclude that $v' = \mathit{next}_v(A')$, i.e. $v' = v$. We have thus shown that $B \xrightarrow{\mu}_{\mathrm{F}} B'$ and it remains to show $A' \sim_{\mathrm{M}} B'$. Since u is a fresh name we have $A'[u/v][v/u] = A'$, and by Lemmas 9 and 11 and $B'' \sim_{\mathrm{M}} A'[u/v]$ we finally get $B' \simeq B''[v/u] \sim_{\mathrm{M}} A'[u/v][v/u] = A'$.

We next prove that $A \sim_{\mathrm{F}} B$ implies $A \sim_{\mathrm{M}} B$ in the same way, by proving that $\sim_{\mathrm{F}}$ is a $\longrightarrow_{\mathrm{M}}$-bisimulation. So assume that $A \sim_{\mathrm{F}} B$ and $A \xrightarrow{\mu}_{\mathrm{M}} A'$. We must prove that $B \xrightarrow{\mu}_{\mathrm{M}} B'$ for some B' such that $A' \sim_{\mathrm{F}} B'$. If μ is an output action then this follows immediately from Lemma 10a and the fact that $\sim_{\mathrm{F}}$ is a $\longrightarrow_{\mathrm{F}}$-bisimulation. So assume that $\mu = a?u$. By Lemma 10d we get that there is a fresh name v' and a program A'' such that $A \xrightarrow{a?v'}_{\mathrm{M}} A''$ and $A' = A''[u/v']$. Now by Lemma 10c we infer that $A \xrightarrow{a?v}_{\mathrm{F}} \circ \simeq A''[v/v']$ for $v = \mathit{next}_{v'}(A'')$. From $A \sim_{\mathrm{F}} B$ we then infer that there is a program B'' such that $B \xrightarrow{a?v}_{\mathrm{F}} B''$ and $B'' \sim_{\mathrm{F}} \circ \simeq A''[v/v']$, i.e. by Lemma 9 $B'' \sim_{\mathrm{F}} A''[v/v']$. By Lemma 10b we conclude that for some $B' \simeq B''[u/v]$ it holds $B \xrightarrow{a?u}_{\mathrm{M}} B'$. We have thus shown that $B \xrightarrow{\mu}_{\mathrm{M}} B'$ and it remains to show that $A' \sim_{\mathrm{F}} B'$. Since $v = \mathit{next}_{v'}(A'')$ we must have either $v = v'$ or $v = \mathit{nextname}(A'')$; in the latter case certainly $v \notin \mathcal{V}(A'')$, so in either case $A''[v/v'][u/v] \simeq A''[u/v']$. From Lemma 11 and $B'' \sim_{\mathrm{F}} A''[v/v']$ we then get $B' \simeq B''[u/v] \sim_{\mathrm{F}} A''[v/v'][u/v] \simeq A''[u/v'] = A'$, which with Lemma 9 implies $A' \sim_{\mathrm{F}} B'$ as required. □

We now sketch the proofs of the lemmas.

Proof of Lemma 7: If $v' = v$ the result is immediate, so assume $v' > v$ and let $u = \mathit{next}_v(A)$. This implies that $u \leq v$ and (1) for all $v'' < u$ it holds $v'' \in \mathcal{V}(A)$, and either of (2) $u = v$ or (3) $u \notin \mathcal{V}(A)$. Now, by definition of $\mathcal{V}(A)$ we obviously have $\mathcal{V}(A)[v'/v] = \mathcal{V}(A[v'/v])$. Since $u \leq v$ we have that a name $v'' < u$ cannot be equal to v, hence from (1) we get that (4) for all $v'' < u$ it holds $v'' \in \mathcal{V}(A[v'/v])$. Also, from $u \leq v < v'$ it follows $u \neq v'$ and we get from (2,3) that either (5) $u = v$ which means that $u \notin \mathcal{V}(A[v'/v])$ or (6) $u \notin \mathcal{V}(A)$ which also implies $u \notin \mathcal{V}(A[v'/v])$; hence in either case $u \notin \mathcal{V}(A[v'/v])$. So, by (4) we get $u = \mathit{nextname}(A[v'/v])$ which with $u < v'$ implies that $u = \mathit{next}_{v'}(A[v'/v])$. □

Proof idea for Lemma 8: Assume that $v \in \mathcal{V}(A)$, i.e. that there is a sequence of $\longrightarrow_{\mathrm{M}}$-transitions from A terminating with the action $a!v$, and which does not contain an input $b?v$. Then, from $A \sim_{\mathrm{M}} B$ and the definition of bisimulation we know that there is a similar sequence of transitions from B, i.e. $v \in \mathcal{V}(B)$. □

Proof idea for Lemma 9: It is straightforward to show that if $A \simeq B$ and $A \xrightarrow{\mu}_{\mathrm{M}} A'$, then for some B' it holds that $B \xrightarrow{\mu}_{\mathrm{M}} B' \simeq A'$. Hence $\simeq$ is a $\longrightarrow_{\mathrm{M}}$-bisimulation. The case for $\sim_{\mathrm{F}}$ is similar. We omit the details.

Proof of Lemma 10: The proof is through induction over the definitions of $\longrightarrow_{\mathrm{M}}$ and $\longrightarrow_{\mathrm{F}}$.

a) Direction $\Longrightarrow$: Assume that $A \xrightarrow{a!u}_{\mathrm{F}} A'$ by the $\textsc{out}_{\mathrm{F}}$ rule. Then $A = a!u.A'$, and hence $A \xrightarrow{a!u}_{\mathrm{M}} A'$ by the $\textsc{out}_{\mathrm{M}}$ rule. Similarly, if $A \xrightarrow{a!u}_{\mathrm{F}} A'$ by the $\textsc{sum}_{\mathrm{F}}$ or $\textsc{id}_{\mathrm{F}}$ rules, the result follows by induction and the $\textsc{sum}_{\mathrm{M}}$ or $\textsc{id}_{\mathrm{M}}$ rules respectively. Direction $\Longleftarrow$ is symmetric.

b) Assume that $A \xrightarrow{a?v}_{\mathrm{F}} A'$ by the $\textsc{in}_{\mathrm{F}}$ rule. Then $A = a?x.C$ and $A' = C[v/x]$, and $v = \mathit{nextname}(C)$. Hence for all $v' < v$ it holds $v' \in \mathcal{V}(C)$, thus also $v' \in \mathcal{V}(C[v/x]) = \mathcal{V}(A')$, whence by definition $v = \mathit{next}_v(A')$. Furthermore, by the $\textsc{in}_{\mathrm{M}}$ rule we get $A \xrightarrow{a?u}_{\mathrm{M}} C[u/x]$. Now since $v \notin \mathcal{V}(C)$ we have $C[v/x][u/v] \simeq C[u/x]$. Thus $A \xrightarrow{a?u}_{\mathrm{M}} C[u/x] \simeq C[v/x][u/v] = A'[u/v]$ as required.

Assume instead that $A \xrightarrow{a?v}_{\mathrm{F}} A'$ by the $\textsc{id}_{\mathrm{F}}$ rule. Then $A = Id(\tilde{u})$ where Id has the definition $Id(\tilde{x}) \Longleftarrow B$, and $B[\tilde{u}/\tilde{x}] \xrightarrow{a?v}_{\mathrm{F}} A'$. By induction we then know that $v = \mathit{next}_v(A')$, and that $B[\tilde{u}/\tilde{x}] \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$ holds for all u, which by the $\textsc{id}_{\mathrm{M}}$ rule implies that A $= Id(\tilde{u}) \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$ holds for all u.

Finally, if $A \xrightarrow{a?v}_{\mathrm{F}} A'$ by the $\textsc{sum}_{\mathrm{F}}$ rule, the result follows as above by induction and the $\textsc{sum}_{\mathrm{M}}$ rule.

c) Assume that $A \xrightarrow{a?u}_{\mathrm{M}} A'$ from the $\textsc{in}_{\mathrm{M}}$ rule. Then $A = a?x.C$ and $A' = C[u/x]$. By $\textsc{in}_{\mathrm{F}}$ it then holds that $A \xrightarrow{a?v}_{\mathrm{F}} C[v/x]$ for $v = \mathit{nextname}(C)$. Since u is a fresh name we have that $C[v/x] = C[u/x][v/u] = A'[v/u]$. Also, $v = \mathit{nextname}(C)$ means that for all $v' < v$ we have $v' \in \mathcal{V}(C)$, whence also $v' \in \mathcal{V}(C[u/x])$. Now $v = \mathit{nextname}(C)$ additionally means that $v \notin \mathcal{V}(C)$, so either $u = v$ or $v \notin \mathcal{V}(C[u/x])$; in either case $v = \mathit{next}_u(C[u/x]) = \mathit{next}_u(A')$.

If $A \xrightarrow{a?u}_{\mathrm{M}} A'$ from $\textsc{sum}_{\mathrm{M}}$ or $\textsc{id}_{\mathrm{M}}$ the result follows by induction in the same way as for Lemma 10b above.

d) Assume that $A \xrightarrow{a?u}_{\mathrm{M}} A'$ from the $\textsc{in}_{\mathrm{M}}$ rule. Then $A = a?x.C$ and $A' = C[u/x]$. Let v be a fresh name; we then by the $\textsc{in}_{\mathrm{M}}$ rule again have that $A \xrightarrow{a?v}_{\mathrm{M}} C[v/x]$. Choose $A'' = C[v/x]$, then $A''[u/v] = C[v/x][u/v] = C[u/x] = A'$ (since v is a fresh name).

If $A \xrightarrow{a?u}_{\mathrm{M}} A'$ from $\textsc{sum}_{\mathrm{M}}$ or $\textsc{id}_{\mathrm{M}}$ the result follows by induction in the same way as for Lemma 10b above. $\square$

Note that for Lemma 10c we have proved that $A \xrightarrow{a?v}_{\mathrm{F}} A'[v/u]$ (and not only that $A \xrightarrow{a?v}_{\mathrm{F}} \circ \simeq A'[v/u]$). In the next subsection we will prove this lemma for $\Longrightarrow_{\mathrm{G}}$, and then the stronger result will not hold. In order to get uniform proofs of Theorems 3 and 4 we use the weaker form of Lemma 10c in the proof of Theorem 3.

Proof idea for Lemma 11: It is straightforward, although a bit tedious, to prove that the relation $\mathcal{S}$ defined by

$$\mathcal{S} = \{(A[u/v], B[u/v]) : A \sim_{\mathrm{M}} B\}$$

is a $\longrightarrow_{\mathrm{M}}$-bisimulation. The proof relies on a lemma (similar to Lemma 10) which relates the transitions of A and $A[u/v]$; this lemma is proven by induction over the definition of $\longrightarrow_{\mathrm{M}}$. The same idea applies to $\sim_{\mathrm{F}}$. We omit the details.

For this lemma to hold, the requirement that schematic names do not occur in identifier definitions is essential. To see this, consider the illegal definition $Id \Longleftarrow a!v.Id$ where v is a name. Here $Id \sim_{\mathrm{M}} a!v.Id$, and if $u \neq v$ then $Id[u/v] = Id \not\sim_{\mathrm{M}} a!u.Id = (a!v.Id)[u/v]$. $\square$

Proof of Theorem 4

The proof of Theorem 4 is similar to the proof of Theorem 3. In the following we write $\approx_{\mathrm{M}}$ for $\Longrightarrow_{\mathrm{M}}$-equivalence, and $\approx_{\mathrm{G}}$ for $\Longrightarrow_{\mathrm{G}}$-equivalence. We first need a new part of Lemma 10 as follows:

Lemma 10 e) $A \xrightarrow{\tau}_{\mathrm{M}} A'$ iff $A \xrightarrow{\tau}_{\mathrm{F}} A'$

Proof: The proof is exactly as the proof of Lemma 10a, only with τ, $\textsc{tau}_{\mathrm{M}}$, and $\textsc{tau}_{\mathrm{F}}$ replacing $a!u$, $\textsc{out}_{\mathrm{M}}$, and $\textsc{out}_{\mathrm{F}}$, respectively. $\square$

We next need to prove that Lemmas 8–11 (including the new Lemma 10e) remain true when we consistently replace $\sim_{\mathrm{M}}$ by $\approx_{\mathrm{M}}$, $\sim_{\mathrm{F}}$ by $\approx_{\mathrm{G}}$, $\longrightarrow_{\mathrm{M}}$ by $\Longrightarrow_{\mathrm{M}}$, and $\longrightarrow_{\mathrm{F}}$ by $\Longrightarrow_{\mathrm{G}}$. The so obtained lemmas will be called Lemmas 8′–11′.

Theorem 4 *Two programs are observation equivalent precisely if they are $\Longrightarrow_{\mathrm{G}}$-equivalent.*

Proof: The proof follows closely that of Theorem 3; we here only indicate the differences. First, consistently replace $\sim_{\mathrm{M}}$ by $\approx_{\mathrm{M}}$, $\sim_{\mathrm{F}}$ by $\approx_{\mathrm{G}}$, $\longrightarrow_{\mathrm{M}}$ by $\Longrightarrow_{\mathrm{M}}$, and $\longrightarrow_{\mathrm{F}}$ by $\Longrightarrow_{\mathrm{G}}$, and replace all references to Lemmas 8–11 by references to Lemmas 8′–11′. Next, in each of the directions of the main proof we have a new case $\mu = \tau$. This case is handled by Lemma 10′e in the same way as the case $\mu = a!u$ is handled by Lemma 10′a. These changes yield a proof of Theorem 4. $\square$

We here only show the proof for Lemma 10′; the other lemmas are straightforward. We will need the following auxiliary lemma (which also is needed in order to prove Lemma 11′):

Lemma 12 If $A \xrightarrow{\tau}_{\mathrm{M}} A'$ then $A[u/v] \xrightarrow{\tau}_{\mathrm{M}} A'[u/v]$, and if $A \xrightarrow{\tau}_{\mathrm{F}} A'$ then $A[u/v] \xrightarrow{\tau}_{\mathrm{F}} A'[u/v]$.

Proof idea: The proof is by induction over the definitions of $\longrightarrow_{\mathrm{M}}$ and $\longrightarrow_{\mathrm{F}}$. If $A \xrightarrow{\tau}_{\mathrm{M}} A'$ by the $\textsc{tau}_{\mathrm{M}}$ rule, then $A = \tau.A'$, and $A[u/v] = \tau.(A'[u/v]) \xrightarrow{\tau}_{\mathrm{M}} A'[u/v]$. If $A \xrightarrow{\tau}_{\mathrm{M}} A'$ by the $\textsc{sum}_{\mathrm{M}}$ or $\textsc{id}_{\mathrm{M}}$ rule the result follows by induction. The case for $\longrightarrow_{\mathrm{F}}$ is similar. $\square$

Proof of Lemma $10'$:

a) We shall prove that $A \overset{a!u}{\Longrightarrow}_{\mathrm{M}} A'$ iff $A \overset{a!u}{\Longrightarrow}_{\mathrm{G}} A'$. For direction $\Longrightarrow$ assume $A \overset{a!u}{\Longrightarrow}_{\mathrm{M}} A'$, i.e. that $A \xrightarrow{\tau}_{\mathrm{M}} A_1 \xrightarrow{\tau}_{\mathrm{M}} \cdots \xrightarrow{\tau}_{\mathrm{M}} A_k \xrightarrow{a!u}_{\mathrm{M}} A_{k+1} \xrightarrow{\tau}_{\mathrm{M}} \cdots \xrightarrow{\tau}_{\mathrm{M}} A'$. Then by one application of Lemma 10a and repeated application of Lemma 10e we get that $A \xrightarrow{\tau}_{\mathrm{F}} A_1 \xrightarrow{\tau}_{\mathrm{F}} \cdots \xrightarrow{\tau}_{\mathrm{F}} A_k \xrightarrow{a!u}_{\mathrm{F}} A_{k+1} \xrightarrow{\tau}_{\mathrm{F}} \cdots \xrightarrow{\tau}_{\mathrm{F}} A'$, which implies $A \overset{a!u}{\Longrightarrow}_{\mathrm{F}} A'$, and by definition of $\Longrightarrow_{\mathrm{G}}$ this implies $A \overset{a!u}{\Longrightarrow}_{\mathrm{G}} A'$. Direction $\Longleftarrow$ is symmetric.

b) We must prove that if $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} A'$ then $v = \mathit{next}_v(A')$, and that for all u it holds that $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} \circ \simeq A'[u/v]$. So assume $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} A'$. Then $A \overset{a?v'}{\Longrightarrow}_{\mathrm{F}} A''$ with $A' = A''[v/v']$ and v is the minimum of v' and $\mathit{nextname}(A'')$. Thus for all $v'' < v$ we have that $v'' \in \mathcal{V}(A'')$ and hence (since $v \leq v'$) also $v'' \in \mathcal{V}(A''[v/v']) = \mathcal{V}(A')$, i.e. $v = \mathit{next}_v(A')$. By Lemmas 10b, 10e, 9, and 12 we then get that $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} \circ \simeq A''[u/v']$. Now either $v = v'$ or $v = \mathit{nextname}(A'')$; in the latter case $v \notin \mathcal{V}(A'')$, so in either case $A''[v/v'][u/v] \simeq A''[u/v']$. Since $A'[u/v] = A''[v/v'][u/v]$ this proves $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} \circ \simeq A'[u/v]$ as required.

c) We must prove that if the name u is fresh for A, and $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} A'$, then for $v = \mathit{next}_u(A')$ it holds $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} \circ \simeq A'[v/u]$. So assume $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} A'$. This means that we have $A \xrightarrow{\tau}_{\mathrm{M}} A_1 \xrightarrow{\tau}_{\mathrm{M}} \cdots \xrightarrow{\tau}_{\mathrm{M}} A_k \xrightarrow{a?u}_{\mathrm{M}} A_{k+1} \xrightarrow{\tau}_{\mathrm{M}} \cdots \xrightarrow{\tau}_{\mathrm{M}} A'$. By Lemmas 10c, 10e, 9, and 12 we get that $A \xrightarrow{\tau}_{\mathrm{F}} A_1 \xrightarrow{\tau}_{\mathrm{F}} \cdots \xrightarrow{\tau}_{\mathrm{F}} A_k \xrightarrow{a?v'}_{\mathrm{F}} \circ \simeq A_{k+1}[v'/u] \xrightarrow{\tau}_{\mathrm{F}} \cdots \xrightarrow{\tau}_{\mathrm{F}} A'[v'/u]$ for $v' = \mathit{next}_u(A_{k+1})$. Thus either $v' = u$ or $v' = \mathit{nextname}(A_{k+1})$; in the latter case $v' \notin \mathcal{V}(A_{k+1})$ which (since $A_{k+1} \xrightarrow{\tau}_{\mathrm{M}} \cdots \xrightarrow{\tau}_{\mathrm{M}} A'$) implies that $v' \notin \mathcal{V}(A')$. The $\longrightarrow_{\mathrm{F}}$-transitions give that $A \overset{a?v'}{\Longrightarrow}_{\mathrm{F}} \circ \simeq A'[v'/u]$. By definition of $\Longrightarrow_{\mathrm{G}}$ this implies $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} \circ \simeq A'[v'/u][v/v']$ for $v = \mathit{next}_{v'}(A'[v'/u])$. Since $v' = u$ or $v' \notin \mathcal{V}(A')$ we have that $A'[v'/u][v/v'] \simeq A'[v/u]$. This proves $A \overset{a?v}{\Longrightarrow}_{\mathrm{G}} \circ \simeq A'[v/u]$. Now $v' = \mathit{next}_u(A_{k+1})$ gives that $v' \leq u$. From $v = \mathit{next}_{v'}(A'[v'/u])$ and Lemma 7 we thus infer that $v = \mathit{next}_u(A'[v'/u][u/v'])$. Since $v' = u$ or $v' \notin \mathcal{V}(A')$ we have that $A'[v'/u][u/v'] \simeq A'$ whence $\mathcal{V}(A'[v'/u][u/v']) = \mathcal{V}(A')$ and thus $\mathit{next}_u(A'[v'/u][u/v']) = \mathit{next}_u(A')$, i.e. $v = \mathit{next}_u(A')$ as required.

d) We must prove that if $A \overset{a?u}{\Longrightarrow}_{\mathrm{M}} A'$, then for some name v not occurring in A it holds $A \overset{a?v}{\Longrightarrow}_{\mathrm{M}} A''$ with $A' = A''[u/v]$. The proof is directly from Lemmas 10d and 12.

e) We must prove that $A \overset{\tau}{\Longrightarrow}_{\mathrm{M}} A'$ iff $A \overset{\tau}{\Longrightarrow}_{\mathrm{G}} A'$. The proof is immediate by repeated application of Lemma 10e.

$\square$

Proof of Theorem 6

We finally prove Theorem 6 by demonstrating that the proofs of Theorems 3 and 4 carry over to the language extended with parallelism and restriction. Obviously, only the parts which explicitly refer to the structure of the language, namely the proofs of Lemmas 9–12, need reconsideration. There then emerges one small complication: Lemmas 10e and 12, and hence also Lemma 10′a and 10′e, only hold in the following slightly weaker form: $\longrightarrow_{\mathrm{M}}$ must be replaced by $\longrightarrow_{\mathrm{M}} \circ \simeq$ in consequents, and similarly $\longrightarrow_{\mathrm{F}}$ by $\longrightarrow_{\mathrm{F}} \circ \simeq$. For instance, Lemma 10e should be modified to "$A \xrightarrow{\tau}_{\mathrm{M}} A'$ implies $A \xrightarrow{\tau}_{\mathrm{F}} \circ \simeq A'$, and $A \xrightarrow{\tau}_{\mathrm{F}} A'$ implies $A \xrightarrow{\tau}_{\mathrm{M}} \circ \simeq A'$".[1] Fortunately, the main proof still holds if the symbols "$\circ \simeq$" are inserted in strategic places. We will not elaborate on the modification of the main proof; instead we indicate how some proofs of the critical lemmas are modified.

Lemma 10: The induction over the definitions of $\longrightarrow_{\mathrm{M}}$ and $\longrightarrow_{\mathrm{F}}$ must be extended to cover the new rules for parallel composition and restriction. This is straightforward for the PARO$_{\mathrm{F}}$, RES$_{\mathrm{F}}$, RES$_{\mathrm{M}}$ rules, and for the PAR$_{\mathrm{M}}$ rule applied to τ or output actions. We show here the remaining cases.

b) Assume that $A \xrightarrow{a?v}_{\mathrm{F}} A'$ by the PARI$_{\mathrm{F}}$ rule. Then $A = B|C$ where $B \xrightarrow{a?v'}_{\mathrm{F}} B'$ and $v = \mathit{nextname}(B'[x/v']|C)$ and $A' = B'[v/v']|C$. By induction $v' = \mathit{next}_{v'}(B')$, i.e. all $v'' < v'$ are used by B', and $B \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq B'[u/v']$ holds for all u. So by the PAR$_{\mathrm{M}}$ rule we infer that $A \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq B'[u/v']|C$. Now either $v = v'$ or $v \notin \mathcal{V}(B')$, and also $v \notin \mathcal{V}(C)$. Hence $A'[u/v] = (B'[v/v']|C)[u/v] \simeq B'[v/v'][u/v]|C \simeq B'[u/v']|C$. This proves $A \xrightarrow{a?u}_{\mathrm{M}} \circ \simeq A'[u/v]$. Also, from $v = \mathit{nextname}(B'[x/v']|C)$ we infer that all $v'' < v$ are used by $B'[x/v']|C$, and hence also used by $B'[v/v']|C = A'$, whence $v = \mathit{next}_v(A')$.

c) Assume that $A \xrightarrow{a?u}_{\mathrm{M}} A'$ by the PAR$_{\mathrm{M}}$ rule. Then $A = B|C$ where $B \xrightarrow{a?u}_{\mathrm{M}} B'$ and $A' = B'|C$. By induction $B \xrightarrow{a?v'}_{\mathrm{F}} B'[v'/u]$ for $v' = \mathit{next}_u(B')$. From PARI$_{\mathrm{F}}$ we infer that $B|C \xrightarrow{a?v}_{\mathrm{F}} B'[v'/u][v/v']$ for $v = \mathit{nextname}(B'[v'/u][x/v']|C)$. Now either $v' = u$ or $v' \notin \mathcal{V}(B')$; in either case $B'[v'/u][v/v'] \simeq B'[v/u]$. Since u is fresh for A it cannot occur in C, whence $C = C[v/u]$. We thus have $B'[v'/u][v/v']|C \simeq B'[v/u]|C[v/u] = (B'|C)[v/u] = A'[v/u]$. This proves $A \xrightarrow{a?v}_{\mathrm{F}} A'[v/u]$ and it remains to prove $v = \mathit{next}_u(A')$. Now $B'[v'/u][x/v'] \simeq B'[x/u]$, hence $v = \mathit{nextname}(B'[v'/u][x/v']|C)$ implies $v = \mathit{nextname}(B'[x/u]|C)$. Thus either $v = u$ or $v \notin \mathcal{V}(B'|C) = \mathcal{V}(A')$. Additionally, for all $v'' < v$ we get $v'' \in \mathcal{V}(B'[x/u]|C)$, which implies $v'' \in \mathcal{V}(B'|C) = \mathcal{V}(A')$. This proves $v = \mathit{next}_u(A')$ as required.

e) *direction* $\Longrightarrow$: Assume that $A \xrightarrow{\tau}_{\mathrm{M}} A'$ by the COM$_{\mathrm{M}}$ rule. Then $A = B|C$ and $B \xrightarrow{a!u}_{\mathrm{M}} B'$ and $C \xrightarrow{a?u}_{\mathrm{F}} C'$ for some u, and $A' = B'|C'$. We must prove that $A \xrightarrow{\tau}_{\mathrm{F}} \circ \simeq A'$. From Lemma 10d we infer that for some fresh v it holds $C \xrightarrow{a?v}_{\mathrm{M}} C''$ with $C' = C''[u/v]$. Lemma 10c then gives that for some C''' it holds that $C \xrightarrow{a?v'}_{\mathrm{F}} C''' \simeq C''[v'/v]$ for $v' = \mathit{next}_v(C'')$. Now $B \xrightarrow{a!u}_{\mathrm{M}} B'$ and Lemma 10a gives that $B \xrightarrow{a!u}_{\mathrm{F}} \circ \simeq B'$, so by the COM$_{\mathrm{F}}$ rule we conclude $A = B|C \xrightarrow{\tau}_{\mathrm{F}} \circ \simeq B'|C'''[u/v'] \simeq B'|C''[v'/v][u/v']$. Since $v' = v$ or $v' \notin \mathcal{V}(C'')$ we get that $C''[v'/v][u/v'] \simeq C''[u/v] = C'$. This proves $A \xrightarrow{\tau}_{\mathrm{F}} \circ \simeq B'|C' = A'$ as required.

[1]The reason for this is that if $A \xrightarrow{a?v}_{\mathrm{F}} A'$ there may be "useless" occurrences of v in A' which are not related to the derivation of the $\longrightarrow_{\mathrm{F}}$ transition. If this transition is used as a premise in the COM$_{\mathrm{F}}$ rule, all v:s (even the unrelated ones) will be substituted. In the corresponding derivation in the $\longrightarrow_{\mathrm{M}}$ semantics this anomaly is not present.

direction $\Longleftarrow$: Assume that $A \xrightarrow{\tau}_{\mathrm{F}} A'$ by the COM$_{\mathrm{F}}$ rule. Then $A = B|C$ and $B \xrightarrow{a!u}_{\mathrm{F}} B'$ and $C \xrightarrow{a?v}_{\mathrm{F}} C'$ for some u, v, and $A' = B'|C'[u/v]$. By Lemma 10a we get that $B \xrightarrow{a!u}_{\mathrm{M}} B'$, and by Lemma 10b that $C \xrightarrow{a?u}_{\mathrm{M}} C'' \simeq C'[u/v]$. Thus by the COM$_{\mathrm{M}}$ rule we conclude $A = B|C \xrightarrow{\tau}_{\mathrm{M}} B'|C'' \simeq B'|C'[u/v] = A'$ as required.

Note that because of the COM$_{\mathrm{F}}$ and COM$_{\mathrm{M}}$ rules, Lemma 10e now relies on Lemmas 10a–10d. □

Lemma 11: The proof idea is the same as before. The induction over the definitions of $\longrightarrow_{\mathrm{M}}$ and $\longrightarrow_{\mathrm{F}}$ is complicated in the same way as the proof of Lemma 10. We omit the details. □

Lemma 12: This lemma is actually one of the lemmas needed to prove Lemma 11. For example we now need to establish, by induction over the definition of $\longrightarrow_{\mathrm{M}}$, for arbitrary μ that if $A \xrightarrow{\mu}_{\mathrm{M}} A'$ then $A[u/v] \xrightarrow{\mu[u/v]}_{\mathrm{M}} A'[u/v]$. We omit the proof since all steps are straightforward. □

This concludes the proof of Theorem 6.