LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems

by

Rance Cleaveland
Joachim Parrow
Bernhard Steffen

ECS-LFCS-89-83

LFCS Report Series (also published as CSR-301-89)

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

August 1989

Copyright © 1989, LFCS

Copyright © 1989, Laboratory for Foundations of Computer Science,
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems *

Rance Cleaveland † Joachim Parrow ‡ Bernhard Steffen §

Abstract

The Concurrency Workbench is an automated tool that caters for the analysis of networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. Its key feature is its scope: a variety of different verification methods, including equivalence checking, preorder checking, and model checking, are supported for several different process semantics. One experience from our work is that a large number of interesting verification methods can be formulated as combinations of a small number of primitive algorithms. The Workbench has been applied to examples involving the verification of communications protocols and mutual exclusion algorithms and has proven a valuable aid in teaching and research.

1 Introduction

This paper describes the Concurrency Workbench [9, 10, 11], a prototype tool for the automatic verification of finite-state processes. Automated verification tools are practically motivated: the development of increasingly complex distributed computer systems requires increasingly complex verification techniques to guarantee correctness, and the increase in detail rapidly becomes unmanageable without computer assistance.

The analysis of concurrent systems relies heavily on the underlying formal semantics of processes, about which there is as yet no consensus. One goal in the design of the Workbench is to incorporate several different verification methods,

*Research supported by British Science and Engineering Research Council grant GC/D69464

†Computer Science Department, North Carolina State University, Box 8206, Raleigh, NC 27695, USA. The work described in this paper was performed while the author was a research associate in the Department of Computer Science at the University of Sussex, Brighton, UK.

‡Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, SWEDEN

§Computer Science Department, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, SCOTLAND

as well as process semantics, in a modular fashion. This means that each method may be applied to different semantic models, yielding a spectrum of techniques for reasoning about processes. The versatility of the Workbench has many advantages: it facilitates a comparison between many aspects of formal verification, it supports mixed verification strategies which use more than one method, and it makes the system easily extensible. This versatility contrasts with existing automated tools, which typically embody a particular semantics and a particular form of verification. Examples of such systems include Aldébaran [14], AUTO [3], XESAR [30], EMC [6], and Winston [25]. For a survey of systems for the verification of communication protocols see [2].

In order to achieve this flexibility the algorithms in the Workbench are partitioned into three *layers*. The first layer manages interaction with the user and also contains the basic definition of process semantics in terms of labeled transition graphs. The second layer provides transformation algorithms on transition graphs. These algorithms enable the user of the Workbench to change the semantic model of processes. The third layer features several basic analysis algorithms for establishing whether a process meets a specification. Depending on the algorithm used, a specification may either be another process (describing the desired behavior) or a formula in a modal logic expressing a relevant property.

The distinction between these three layers is one of the main achievements of the Workbench. In particular, the combination of various graph transformations with the analysis algorithms yields a wide spectrum of verification algorithms, and yet the underlying code is small and structured. As the layers are implemented independently of one another, the system is easy to maintain and extend.

The Workbench has been successfully applied to verification of communication protocols, notably the alternating bit protocol and the CSMA/CD protocol [29], and mutual exclusion algorithms [35]; it has also been used to debug the Edinburgh Computer Science Department's electronic mailing system. Furthermore, it is a valuable instrument in education, in industries as well as in universities, and is being investigated as a tool for analyzing communications protocols by Swedish Telecom and by Hewlett-Packard.

The remainder of the paper is organized as follows. In the next section we describe the conceptual structure of the workbench, and give an overview of the different verification methods. Section 1 describes the conceptual structure of the Workbench and Section 3 the model of processes used in the Workbench. Sections 4, 5 and 6 present the equivalence testing, preorder testing and model checking facilities in the Workbench, respectively, while Section 7 contains a brief account of some of the ways in which the Workbench is being extended. Section 8 contains our conclusions and directions for future work.

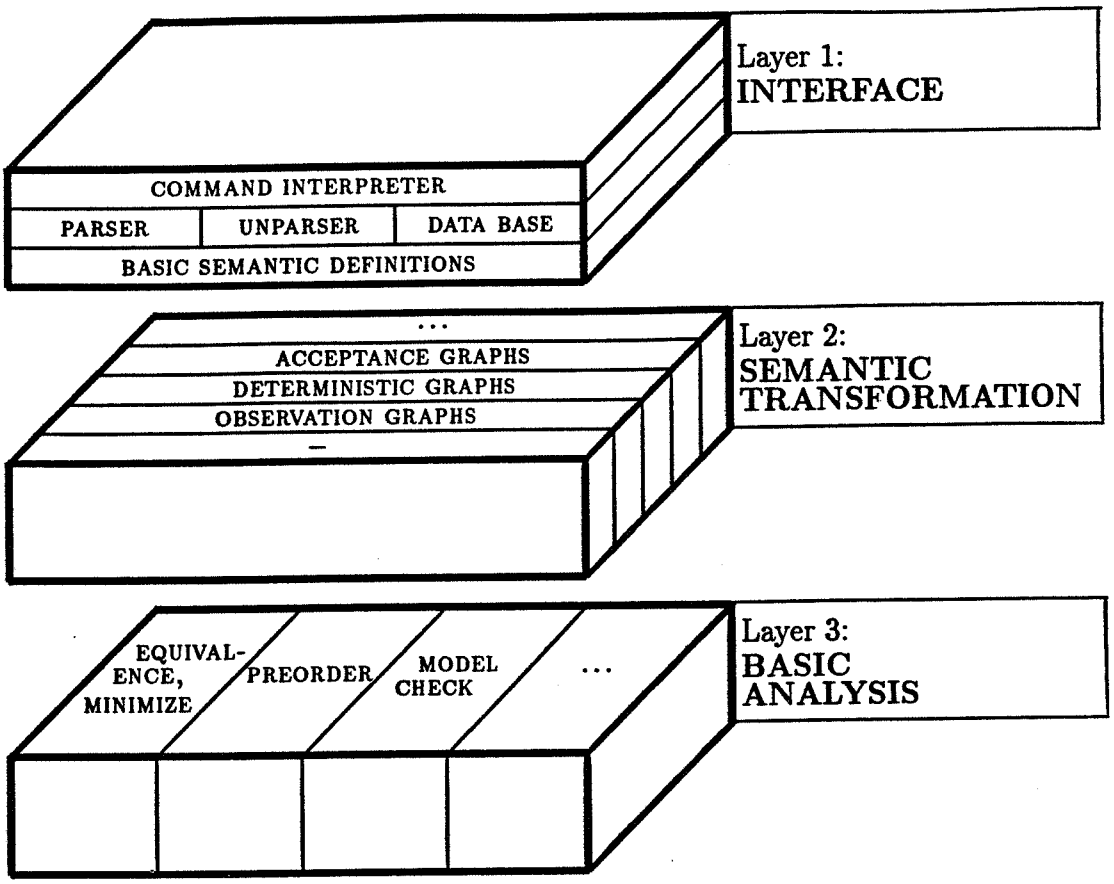


Figure 1: Overview of the Workbench

2 The Architecture of the Workbench

An overview of the Workbench is presented in Figure 1. In order to supply a wide variety of approaches for verification while maintaining a conceptually economical core, the system is highly modularized. Of particular importance is the separation between the three top-level layers.

2.1 The Interface Layer

The *interface layer* oversees the interaction between the Workbench and the user. Its key component is a *command interpreter*, which invokes the appropriate parts of the other layers and presents analysis results. Each verification method is implemented as one command, which may require parameters in the form of processes and modal formulas. *Parsers* transform the concrete syntax of such parameters into parse trees, which can be stored in a small *data base*. There is also a package defining the *basic semantics* of processes. Following [26], a process is interpreted

as labeled transition graph that describes the states a process may enter and the state transitions that result when a process executes an action. A description of the syntax and semantics of processes can be found in Section 3.

2.2 The Semantics Layer

The semantics layer consists of algorithms for transforming the transition graphs generated by the interface layer. For example, the *observation* transformation extends the transitions by absorbing internal computation into visible events; thus the resulting graphs do not record the timing of internal computations in the corresponding processes. The *deterministic* transformation makes the graphs deterministic in the usual sense of the word: the resulting graphs do not record nondeterministic choices. The *acceptance* transformation yields deterministic graphs augmented with information regarding divergence and nondeterminism (and hence deadlock) in the form of acceptance sets. These transformations, and some minor variations, are described in Section 3.3.

The semantics layer refers to processes represented as transition graphs rather than processes in the abstract syntax. Thus, future changes of the particular syntax will not require changes in this layer.

2.3 The Analysis Layer

The Workbench features three main verification methods for proving that processes meet their specifications, and the analysis layer contains the corresponding basic analysis algorithms. In the first method, specifications are themselves processes that describe precisely the high-level behavior required of an implementation. The corresponding algorithm determines whether two processes are *equivalent* in the sense of having the same behavior. This algorithm can also be used to *minimize* a process, i.e. compute an equivalent process with a minimal number of states. A definition of equivalence and a brief account of the algorithm can be found in Section 4.

The second method also uses processes as specifications, but these specifications are treated as minimal requirements to be met by implementations. In this approach specifications can be annotated with "holes" (or "don't care" points); an implementation satisfies one of these *partial* specifications if it supplies at least the behavior demanded by the specification while filling in some holes. The method relies on an ordering relation, or *preorder*, between processes: a process A is "more defined than" a process B if A has the same behavior as B except for the holes in B. The preorder algorithm determines if a process is more defined than its specification in this sense. A definition of the preorder and an account of the algorithm can be found in Section 5.

The third method involves the use of a modal logic, the propositional (modal) mu-calculus. Assertions formulated in this logic are viewed as specifications; ex-

amples of such assertions are "there are no deadlocks" or "every action of type a is always followed by an action of type b". The logic exhibits a considerable expressive power. As further described in Section 6, the *model checking* algorithm determines whether a process satisfies an assertion.

The basic analysis algorithms are polymorphic; that is, they work equally well on the different kinds of transitions graphs supplied by the semantics layer. For instance, the equivalence algorithm computes CCS strong equivalence on transition graphs. If applied to observation graphs then this equivalence corresponds to CCS observation equivalence on the original graphs; if the graphs have been made deterministic it corresponds to trace equivalence. Observation congruence and testing (failures) equivalence can also be computed by first choosing appropriate transformations and applying the general equivalence checking algorithm. Analogous results hold for the other basic analysis algorithms.

3 Representation of Processes

This section describes the syntax of the Calculus of Communicating Systems (CCS) which is used to define processes, or *agents*, used in the Workbench, and it shows how such agents are interpreted as transition graphs. Transformations of transition graphs are also introduced; these enable a change of the semantics under consideration. We assume the reader to have some familiarity with CCS.

3.1 Actions and Agents

CCS agents are built from a set of *actions* containing a distinguished unobservable (or silent) action τ. The observable actions, also called *communication events*, are divided into input events and output events. In the following $a, b, \ldots$ will range over input events, and $\overline{a}, \overline{b}, \ldots$ will range over output events. Input event a and output event $\overline{a}$ are said to be *complementary*, reflecting the fact that they represent input and output on the "port" a. We consider only communication events without value parameters. Agents are defined using the following standard operators from [26].

Nil	Termination
$\perp$	Divergence (or *bottom*)
$a.$	Prefixing by action a; unary prefix operator
$+$	Choice; binary infix operator
$\mid$	Parallel composition; binary infix operator
$\backslash L$	Restriction on (finite) set of actions L; unary postfix operator
$[f]$	Relabeling by f, which maps actions to actions; unary postfix operator

Relabeling functions f are required to satisfy two conditions: $f(\tau) = \tau$, and $f(\overline{a}) = \overline{f(a)}$. They are frequently written as a sequence of substitutions; for example $p[a_1/b_1, a_2/b_2]$ is the process p with with b_1, b_2, $\overline{b_1}$ and $\overline{b_2}$ replaced by a_1, a_2, $\overline{a_1}$ and $\overline{a_2}$, respectively.

We also assume a set of *agent identifiers*. An identifier A may be *bound* to an agent expression p that may itself contain A. This enables recursively defined processes.

Agents are given an operational semantics defined in terms of *transition relations*, $\xrightarrow{a}$, where a is an action. Intuitively, $p \xrightarrow{a} p'$ holds when p can evolve into p' by performing action a; in this case, p' is said to be an *a-derivative* of p. The transition relation is defined inductively on the basis of the constructors used to define an agent. Thus, $a.p \xrightarrow{a} p$ holds for any p, and $p + q \xrightarrow{a} p'$ if either $p \xrightarrow{a} p'$ or $q \xrightarrow{a} p'$. The agent $p|q$ behaves like the "interleaving" of p and q with the possibility of complementary actions synchronizing, yielding a τ action. $p \setminus L$ behaves like p with the exception that no actions in L are allowed, while $p[f]$ behaves like p with actions renamed by f. A formal account of the semantics may be found in the appendix at the end of the paper. Examples of agents defined in CCS appear in Figure 2.

3.2 Transition Graphs

The Workbench uses *transition graphs* to model processes. These graphs statically represent the operational behavior of agents; given an agent, the system generates the corresponding transition graph on the basis of the transitions available to the agent. A transition graph contains a set of *nodes* (corresponding to processes) with one distinguished node, the *root* node, and a set of edges which are labeled by actions (corresonding to transitions between processes). An edge labeled by a has source n and target n' iff $p \xrightarrow{a} p'$ holds of the corresponding processes. Figure 3 contains examples of transition graphs.

Each node additionally carries a polymorphic *information* field, the contents of which vary according to the computations being performed on the graph. For example, the algorithm for computing testing equivalence and the algorithm for computing preorders store *acceptance sets* and *divergence information*, respectively, in this field.

3.3 Graph Transformations

As we indicated previously, several transformations on transition graphs are used in conjunction with general algorithms to yield a variety of verification methods. We briefly describe some of these transformations here.

- BUF_n defines a buffer of capacity n.

$$\begin{aligned}
\mathrm{BUF}_n &= \mathrm{BUF}_n^0 \\
\mathrm{BUF}_n^0 &= in.\mathrm{BUF}_n^1 \\
\mathrm{BUF}_n^i &= in.\mathrm{BUF}_n^{i+1} + \overline{out}.\mathrm{BUF}_n^{i-1} \quad \text{for } i = 1, \ldots, n-1 \\
\mathrm{BUF}_n^n &= \overline{out}.\mathrm{BUF}_n^{n-1}
\end{aligned}$$

- CBUF_n defines a *compositional* buffer of capacity n.

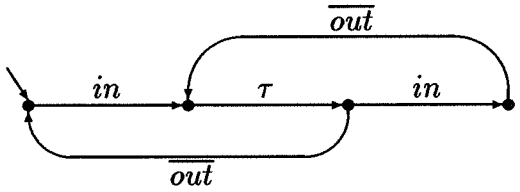
$$\mathrm{CBUF}_n = (\mathrm{BUF}_1[x_1/out] \mid \underbrace{\ldots \mid \mathrm{BUF}_1[x_i/in, x_{i+1}/out] \mid \ldots}_{i=1,\ldots,n-2} \mid \mathrm{BUF}_1[x_n/in]) \setminus \{x_1, ., x_n\}$$

- The *partial* of capacity n, PBUF_n, specifies agents that behave like buffers of capacity n, so long as no more than n elements are stored at once.

$$\begin{aligned}
\mathrm{PBUF}_n &= \mathrm{PBUF}_n^0 \\
\mathrm{PBUF}_n^0 &= in.\mathrm{PBUF}_n^{n-1} \\
\mathrm{PBUF}_n^i &= in.\mathrm{PBUF}_n^{i+1} + \overline{out}.\mathrm{PBUF}_n^{i-1} \quad \text{for } i = 1, \ldots, n-1 \\
\mathrm{PBUF}_n^n &= in.\bot + \overline{out}.\mathrm{PBUF}_n^{n-1}
\end{aligned}$$

Figure 2: Examples of Buffers Defined in CCS.

The transition graph for BUF_n, the buffer of capacity n.



The transition graph for CBUF_2, the compositional buffer of capacity 2.

Figure 3: Examples of Transition Graphs.

3.3.1 Observation Graphs

The transition graphs as described in Section 3.2 are synchronous in the sense that they faithfully represent τ events, and hence the "timing behavior", of agents. Many verification methods require this information; however, others do not, and to cater for these the Workbench includes a procedure for computing *observation* graphs.

Observation graphs are based on the notion of *observations.* These are defined as follows.

$$n \stackrel{\varepsilon}{\Longrightarrow} n' \text{ iff } n \stackrel{\tau}{\longrightarrow}^* n'$$
$$n \stackrel{a}{\Longrightarrow} n' \text{ iff } n \stackrel{\varepsilon}{\Longrightarrow} \stackrel{a}{\longrightarrow} \stackrel{\varepsilon}{\Longrightarrow} n'$$

So $\stackrel{\varepsilon}{\Longrightarrow}$ is defined as the transitive and reflexive closure of $\stackrel{\tau}{\longrightarrow}$, and $\stackrel{a}{\Longrightarrow}$ is defined in terms of relational products of $\stackrel{\varepsilon}{\Longrightarrow}$ and $\stackrel{a}{\longrightarrow}$. These relations allow τ events to be *absorbed* into visible events, so that information as to the precise amount of internal computation performed is obscured.

The observation graph transformation takes a graph and modifies the links to reflect the $\stackrel{a}{\Longrightarrow}$ and $\stackrel{\varepsilon}{\Longrightarrow}$ relations instead of the $\stackrel{a}{\longrightarrow}$ and $\stackrel{\tau}{\longrightarrow}$ relations. It uses well-known methods for computing the product of two relations and the transitive and reflexive closure of a relation. Figure 4 indicates the nature of the transformation (for clarity, we have omitted the ε-links resulting from the reflexive closure of $\stackrel{\tau}{\longrightarrow}$).

The transformation takes time that is polynomial in the size of the graph.

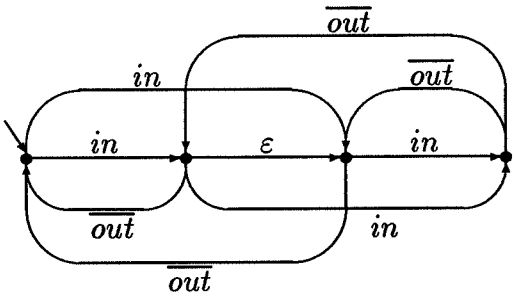


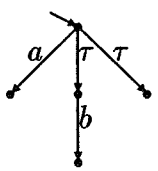
Figure 4: The Observation Graph for CBUF_2.

Surprisingly, in most cases the subroutine for transitive closure accounts for more than 80 per cent of the execution time when determining observation equivalence.

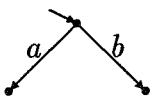
A variation of the observation transformation computes *congruence graphs*, which are used to check for observational *congruence* [26] and weak precongruence [34]. Intuitively, these graphs are observation graphs that record the possibility of initial τ-actions. To construct them, a copy of the root node is created; this new node becomes the root node of the congruence graph, and by construction it has no incoming edges. Subsequently, the observation transformation is applied as before, except that for the new root node, the transitive closure of $\xrightarrow{\tau}$ is applied, rather than the transitive and reflexive closure.

3.3.2 Deterministic Graphs

The strong and observation equivalences and preorders distinguish agents on the basis of the exact points during their executions where nondeterministic choices are made. Accordingly, the graphs mentioned in Sections 3.2 and 3.3.1 faithfully record each time a agent makes such a choice. However, other relations do not require such detailed accounts of the choice structure of agents; for example, the *may* preorder and equivalence [15] (which coincide with *trace* containment and equivalence, respectively) require only information about an agent's *language*, or the sequences of visible actions the agent may perform. In order to cater for these relations the Workbench includes an algorithm for transforming graphs into language-equivalent *deterministic* graphs (also called *Dgraphs* in [8]), i.e. graphs having no τ-derivatives and at most one a-derivative for any action a. As a simple example, the transition graph



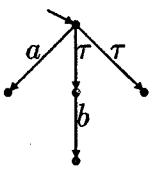
is transformed to



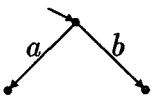
As another example, the deterministic graph of CBUF_2 is precisely the transition graph of BUF_2. The algorithm for computing deterministic graphs is well-known from automata theory (see e.g. [18]). In general, this transformation has an exponential complexity, owing to the fact that it is theoretically possible to have a node in the deterministic graph for each subset of nodes in the original graph. Our experience, however, indicates that the number of nodes is usually smaller than the number of nodes in the original graph, owing to the collapsing of τ-transitions.

3.3.3 Acceptance Graphs

In addition to the language of an agent, other relations, such as the *testing* and *failures* equivalences and preorders [15, 17], require information about an agent's *divergence potential*, or capability of engaging in an infinite internal computation, and *degree of nondeterminism*, as it attempts to "execute" a sequence of visible actions. The appropriate transition graphs for these relations are *acceptance* graphs (also called *Tgraphs*); these are deterministic graphs whose nodes additionally contain information regarding divergence and nondeterminism encoded as *acceptance sets*. The acceptance set $n.acc$ of a node n is a set of sets of actions: each member of $n.acc$ is a set of actions, representing an obligation to be able to continue from n with any action in the set. For example,



has the acceptance graph



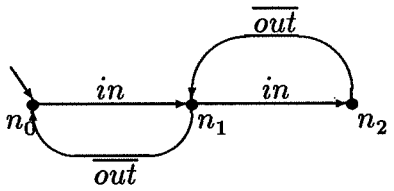
where the root node has acceptance set

$$\{\emptyset, \{b\}\}$$

and each leaf node has acceptance set

$$\{\emptyset\}.$$

Each element in the acceptance set corresponds to a stable state (a state from which no internal action is immediately possible) in the original graph; such an element contains precisely the actions that are immediately possible from the stable state. In the example, the two elements of the acceptance set correspond to the possibilities of the original graph to evolve unobservably along the right branch (where no further action is possible) or the middle branch (where only b is possible). Note that $\{a\}$ is not a member of the acceptance set of the start state set since the start state of the original graph is not stable and cannot evolve into a stable state where only an a is possible. As another example, Figure 5 shows the acceptance graph resulting from the transformation of CBUF_2.



$$
\begin{aligned}
n_0.acc &= \{\{in\}\} \\
n_1.acc &= \{\{in, \overline{out}\}\} \\
n_2.acc &= \{\{\overline{out}\}\}
\end{aligned}
$$

Figure 5: The acceptance graph for CBUF_2.

Acceptance sets may also be used to record divergence information. By convention, if an acceptance graph node has an empty acceptance node, then the node is *divergent*, meaning that during the "execution" of the sequence of actions leading from the start node to the present node, it is possible for the agent to engage in an infinite internal computation.

A closely related kind of graph, the *must* graph (also called *STgraph*), is appropriate for the must equivalence and preorder ([15]). Must graphs are like acceptance graphs, except that a node reachable from a globally divergent node has no outgoing edges. The algorithm for generating acceptance and must graphs is described in [8].

4 Equivalence Checking

The first basic analysis within the Workbench involves checking for equivalence between two agents. As indicated in Section 2, our approach is to convert the agents to transition graphs of the appropriate type and then apply a general equivalence algorithm.

4.1 Definition of the Equivalence

The Workbench uses a general notion of equivalence between transition graphs that is based on "node matching." Intuitively, two graphs will be deemed equivalent if it is possible to match up their nodes in such a way that

1. two matched nodes have "compatible" information fields (the specific notion of compatibility will depend on the equivalence being computed);

2. if two nodes are matched and one has an a-derivative, then the other must have a matching a-derivative; and

3. the start nodes of the two graphs are matched.

This intuitive account may be formalized as follows. Let G_1 and G_2 be transition graphs with node sets N_1 and N_2, respectively, let $N = N_1 \cup N_2$, and let $\mathcal{C} \subseteq N \times N$ be an equivalence relation reflecting a notion of "compatibility between information fields."

Definition 4.1 A *$\mathcal{C}$-bisimulation* on G_1 and G_2 is a relation $\mathcal{R} \subseteq N \times N$ such that $\langle m, n \rangle \in \mathcal{R}$ implies that:

1. if $m \xrightarrow{a} m'$ then $\exists n' : n \xrightarrow{a} n'$ and $\langle m', n' \rangle \in \mathcal{R}$, and
2. if $n \xrightarrow{a} n'$ then $\exists m' : m \xrightarrow{a} m'$ and $\langle m', n' \rangle \in \mathcal{R}$, and
3. $\langle m, n \rangle \in \mathcal{C}$.

A $\mathcal{C}$-bisimulation, then, is a "matching", as described above. Accordingly, two graphs are said to be $\mathcal{C}$-equivalent if there exists a $\mathcal{C}$-bisimulation relating the root nodes of the graphs.

4.2 Derived Equivalences

Many interesting equivalences turn out to be instances of $\mathcal{C}$-equivalence on appropriately transformed graphs. Let U denote the universal relation, i.e. $U = N \times N$. Then a U-bisimulation is a bisimulation in the sense of Milner [26] and U-equivalence is strong equivalence in CCS. Also, observation equivalence corresponds to U-equivalence on observation graphs, observation congruence to U-equivalence on congruence graphs, and trace (or may) equivalence to U-equivalence on deterministic graphs. For must and testing (failures) equivalence we define $\langle m, n \rangle \in \mathcal{A}$ to hold exactly when $m.acc$ and $n.acc$ are compatible, i.e. each element of $m.acc$ is a superset of an element of $n.acc$, and vice versa. Then two graphs are must equivalent if their associated must graphs are $\mathcal{A}$-equivalent, and they are testing (failures) equivalent if their associated acceptance graphs are $\mathcal{A}$-equivalent [8].

As an example, recall the definitions for BUF_n and CBUF_n (see Section 3).For any n, these two agents can be shown to be equivalent according to each of these equivalences, with the exception of strong equivalence.

4.3 The Algorithm

Our algorithm is adapted from one presented in [21]. It works by attempting to find a $\mathcal{C}$-bisimulation relating the root nodes of the graphs. To do so, it maintains a *partitioning* of the nodes in G_1 and G_2, the graphs under consideration. A partitioning is a set of *blocks*, where each block is a set of nodes such that each

node is contained in exactly one block. Such a partitioning naturally induces an equivalence relation on the nodes of the graphs: two nodes are related precisely when they are in the same block.

The algorithm starts with the partition containing only one block and successively refines this partition. It terminates when the roots of the two graphs end up in different blocks (in which case the graphs are not equivalent) or the induced equivalence relation on the nodes becomes a $\mathcal{C}$-bisimulation (in which case the graphs are $\mathcal{C}$-equivalent).

The time and space complexities of this algorithm are $O(k * \ell)$ and $O(k + \ell)$ respectively, where k is the number of states, and ℓ is the number of transitions, in the two graphs. A marginally more efficient algorithm appears in [27]; however, there is not yet enough evidence to suggest that this algorithm is appreciably faster in practice. In any event, this complexity is not a limiting factor; tests with the Workbench have shown that the time consumed by this algorithm is only a small fraction of the total time spent when computing observation equivalence. Most of the time is consumed in the graph transformations.

One final interesting point is that the algorithm can be trivially modified to determine the coarsest $\mathcal{C}$-bisimulation on the nodes of a single graph. This can be used to transform a graph into a $\mathcal{C}$-equivalent one which has a minimum number of states: first compute the coarsest $\mathcal{C}$-bisimulation and then collapse each block in the final partition into a single node.

5 Preorder Checking

The second basic analysis within the Workbench involves checking a preorder between two agents. This is done in a way similar to equivalence checking; after converting the agents to transition graphs we then apply a general preorder algorithm. The algorithm is based upon the following generalization of the notion of equivalence introduced in Section 4.1.

5.1 Definition of the Preorder

The Workbench uses a general notion of preorder between transition graphs. Like for the equivalence this notion is based on "node matching". The main idea behind the preorder definition is to generalize the notion of equivalence given in the last section by qualifying certain actions as to be unimportant under certain circumstances: if an action a is qualified to be unimportant for one of the processes in a certain state, this state has no to match the a-transition of its matching states. Furthermore, compatibility of the information fields is given by a preorder rather then an equivalence relation. This idea of generalization may be formalized as follows. Let G_1 and G_2 be transition graphs with node sets N_1 and N_2, respectively, let $N = N_1 \cup N_2$, and let $\mathcal{C} \subseteq N \times N$ be an equivalence relation reflecting a notion

of "compatibility between information fields."

Definition 5.1 A *C-bisimulation* on G_1 and G_2 is a relation $\mathcal{R} \subseteq N \times N$ such that $\langle m, n\rangle \in \mathcal{R}$ implies that:

Let G_1 and G_2 be transition graphs with (disjoint) node sets N_1 and N_2, let $N = N_1 \cup N_2$, and let $\mathcal{C} \subseteq N \times N$ be a preorder reflecting a notion of "ordering on information fields" (in general a preorder is a transitive and reflexive relation). Also let $\mathcal{P}_a \subseteq N$ and $\mathcal{Q}_a \subseteq N$ be predicates over N, where a ranges over the set of actions. Intuitively, $\mathcal{P}_a$ and $\mathcal{Q}_a$ determine the states from which a-transitions must be matched.

Definition 5.2 A *C-prebisimulation* between G_1 and G_2 is a relation $\mathcal{R} \subseteq N \times N$ such that $\langle m, n\rangle \in \mathcal{R}$ implies that:

1. if $n \in \mathcal{P}_a$ then [if $m \xrightarrow{a} m'$ then $\exists n' :\ n \xrightarrow{a} n'$ and $\langle m', n'\rangle \in \mathcal{R}$], and
2. if $m \in \mathcal{Q}_a$ then [if $n \xrightarrow{a} n'$ then $\exists m' :\ m \xrightarrow{a} m'$ and $\langle m', n'\rangle \in \mathcal{R}$], and
3. $\langle m, n\rangle \in \mathcal{C}$.

The $\mathcal{C}$-preorder is defined by: $G_1 \sqsubseteq_{\mathcal{C}} G_2$ if there exists a $\mathcal{C}$-prebisimulation relating the roots of the two graphs. Note that when $\mathcal{P}_a = \mathcal{Q}_a = N$ and $\mathcal{C}$ is an equivalence relation, then a $\mathcal{C}$-prebisimulation is just a $\mathcal{C}$-bisimulation.

5.2 Derived Preorders

Many interesting preorders turn out to be instances of $\mathcal{C}$-preorder on appropriately transformed graphs. Let U denote the universal relation on N and $\Downarrow a$ the *local convergence* predicate on a; $n \Downarrow a$ holds if n is not globally divergent and cannot be triggered by means of an a-action to reach a globally divergent state. For details of this predicate see [32, 34]. Now many interesting preorders can be constructed as instances of the $\mathcal{C}$-preorder:

- The bisimulation divergence preorder [32, 34] results by setting:

  $$\mathcal{P}_a = N, \mathcal{Q}_a = \{n | n \Downarrow a\} \quad \text{and} \quad \mathcal{C} = \{\langle m, n\rangle | \text{ for all } a{:}\, m \Downarrow a \Rightarrow n \Downarrow a\}.$$

  This defines the strong version of the divergence preorder. The weak version, $\sqsubseteq_{\approx}$, where τ-actions are not observable, can be obtained from the corresponding observation graphs that have the divergence information stored in their information fields.

- The may, must and testing preorders require the transformation of graphs into deterministic, must, and acceptance graphs, respectively. Then these relations are the following instances of the general preorder [8].

- The may preorder: $\mathcal{P}_a = N$, $\mathcal{Q}_a = \emptyset$, and $\mathcal{C} = U$.
- The must preorder: $\mathcal{P}_a = \emptyset$, $\mathcal{Q}_a = \{m | m.acc \neq \emptyset\}$, and $\langle m, n \rangle \in \mathcal{C}$ holds exactly when either $m.acc = \emptyset$, or both $m.acc$ and $n.acc$ are nonempty and each element in $n.acc$ is a superset of some element in $m.acc$.
- The testing preorder: $\mathcal{P}_a = N$, $\mathcal{Q}_a = \{m | m.acc \neq \emptyset\}$, and $\mathcal{C}$ is defined as for the must preorder.

The preorder can be regarded as a specification-implementation relation, i.e. we can interprete $P \precsim Q$ as "Q is closer to an implementation than P". This interpretation is based upon the idea of regarding divergence as a means of marking underspecified states. For example the totally divergent state $\perp$ can be seen as the totally unspecified state, allowing each process as a correct implementation. "Loose" or "partial" specifications can be used to establish that processes, although not being equivalent, may be used interchangeably in certain contexts. As an example, assume that we have a transport protocol with sender entity SENDER and receiver entity RECEIVER, interconnected through ports L with a medium MEDIUM. To verify this protocol, we might want to establish its observational equivalence to a complete service specification SERVICE:

$$\text{SERVICE} \quad \approx \quad (\text{SENDER} | \text{RECEIVER} | \text{MEDIUM}) \backslash L \tag{1}$$

Furthermore assume that SENDER and RECEIVER are already proved correct, and that we only have to check the implemention MEDIUM of the medium, whose specification might be given by PBUF_n. This assumption can be formalized by:

$$\text{SERVICE} \quad \precsim \quad (\text{SENDER} | \text{RECEIVER} | \text{PBUF}_n) \backslash L \tag{2}$$

Now, exploiting the fact that

$$\text{PBUF}_n \quad \precsim \quad \text{BUF}_m, \quad \text{for all } m > n$$

and that $\precsim$ preserves "$|$" contexts, we therefore obtain:

$$(\text{SENDER} | \text{RECEIVER} | \text{PBUF}_n) \backslash L \quad \precsim \quad (\text{SENDER} | \text{RECEIVER} | \text{BUF}_m) \backslash L \tag{3}$$

and together with (2), (3), and the transitivity of $\precsim$:

$$\text{SERVICE} \quad \precsim \quad (\text{SENDER} | \text{RECEIVER} | \text{BUF}_m) \backslash L \tag{4}$$

It is easy to see that the preorder $\precsim$ coincides with $\approx$ for complete specifications. In fact, whenever the left hand side process is completely specified then so is the right hand side process, and both processes are equivalent. Thus the completeness of SERVICE yields:

$$\text{SERVICE} \quad \approx \quad (\text{SENDER} | \text{RECEIVER} | \text{BUF}_m) \backslash L$$

and therefore establishes all buffers of capacity greater than n as correct implementations of the medium.

5.3 The Algorithm

The algorithm for computing the $\mathcal{C}$-preorder works by attempting to find a $\mathcal{C}$-prebisimulation relating the roots of the graphs. In contrast to Section 4.3, however, preorders cannot be represented by partitions. We obtain an appropriate representation by annotating every node n with a set of nodes considered to be "greater" than n.

In principle, the preorder algorithm proceeds in the same way as the equivalence algorithm. It starts by considering all states to be indistinguishable, i.e. every node is annotated with the set of all nodes N. Then it successively refines the annotation of each node until the root node of G_1 no longer is in the annotation of the root node of G_2 (in which case $G_1 \not\sqsubseteq_C G_2$) or the annotations determine a $\mathcal{C}$-prebisimulation (in which case $G_1 \sqsubseteq_C G_2$).

The time and space complexities of this algorithm are $O(k^4 * \ell)$ and $O(k^2 + \ell)$, respectively, where k is the number of states, and ℓ is the number of transitions, in the two graphs. The loss of efficiency compared with the equivalence algorithm is due to the fact that we cannot use the same compact representation of relations as in Section 4.3.

6 Model Checking

The Workbench also supports a verification method based on model checking [5, 6, 7, 13, 33], in which specifications are written in an expressive modal logic based on the *propositional (modal) mu-calculus*. The system can automatically check whether an agent meets such a specification.

The Workbench actually uses two logics, the *interface logic* and the *system logic*. The former is a "syntactically sugared" version of the latter that also provides for user-defined propositional constructors, called *macros*. The model checker establishes that a node in a graph enjoys a property in the interface logic by first translating the property into a formula in the system logic, which is simpler to analyze. We shall only described the interface logic here.

6.1 The Logic

The interface logic includes traditional propositional constants and connectives together with modal operators and mechanisms for defining recursive propositions. The formulas are described by the following grammar.

$$
\begin{aligned}
\Phi \;::=\;& \mathit{tt} \mid \mathit{ff} \mid X \\
\mid\;& \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \Phi \Rightarrow \Phi \\
\mid\;& \langle a \rangle \Phi \mid [a]\Phi \mid \langle . \rangle \Phi \mid [.]\Phi \\
\mid\;& B\ \textit{arg-list} \\
\mid\;& \nu X.\Phi \mid \mu X.\Phi
\end{aligned}
$$

In the above, X ranges over variables, a over actions, B over user-defined macro identifiers, and *arg-list* over lists of actions and/or formulas that B requires in order to produce a proposition. There is a restriction placed on Φ in $\nu X.\Phi$ and $\mu X.\Phi$ that requires any *free* occurrences of X to appear inside the scope of an even number of negations.

These formulas are interpreted with respect to nodes in transition graphs. *tt* and *ff* hold of every node and no node, respectively. X is interpreted with respect to an environment binding variables to propositions; n satisfies X if it satisfies the formula to which X is bound in the environment. $\neg\Phi$ holds of a node n if Φ does not hold of n, $\Phi_1 \vee \Phi_2$ holds of n if either of Φ_1 or Φ_2 does, $\Phi_1 \wedge \Phi_2$ holds of n if both Φ_1 and Φ_2 do, and $\Phi_1 \Rightarrow \Phi_2$ holds of n if, whenever Φ_1 holds of n, then Φ_2 does as well.

The modal constructors $\langle a\rangle$, $[a]$, $\langle . \rangle$ and $[.]$ make statements about the edges leaving a node. A node n satisfies $\langle a\rangle\Phi$ if it has an a-derivative n' with n' satisfying Φ, while n satisfies $[a]\Phi$ if *all* its a-derivatives satisfy Φ. In the case that n has no such derivatives, n trivially satisfies $[a]\Phi$. In $\langle . \rangle\Phi$ and $[.]\Phi$, the "." should be thought of as a "wild-card" action; n satisfies $\langle . \rangle\Phi$ if it satisfies $\langle a\rangle\Phi$ for some a, while it satisfies $[.]\Phi$ if it satisfies $[a]\Phi$ for all a.

A macro can be thought of as a "function" that accepts some number of arguments, which may be either actions or formulas, and returns a proposition. A formula *B arg-list* is then interpreted as the proposition returned by B in response to *arg-list*.

Formulas of the type $\nu X.\Phi$ and $\mu X.\Phi$ are *recursive formulas*; they correspond to certain kinds of infinite conjunctions and disjunctions in the following sense. Let Φ_0 be the proposition *tt*, and define Φ_{i+1} to be the proposition $\Phi[\Phi_i/X]$, namely, the proposition obtained by substituting Φ_i for free occurrences of X in Φ. Then $\nu X.\Phi$ corresponds to the infinite conjunction $\bigwedge_{i=0}^{\infty} \Phi_i$. Now let $\hat{\Phi}_0$ be the proposition *ff*, and let $\hat{\Phi}_{i+1}$ be defined as $\Phi[\hat{\Phi}_i/X]$. Then $\mu X.\Phi$ may be interpreted as the infinite disjunction $\bigvee_{i=0}^{\infty} \hat{\Phi}_i$.

The recursive proposition constructors add a tremendous amount of expressive power to the logic (cf. citeEmersonLei,Steffen). For example, they allow the description of invariance (or *safety*) and eventuality (or *liveness*) properties. However, the formulas are in general unintuitive and difficult to understand. We have found that the most effective way to use the model checker is to choose a collection of intuitively well-understood operators that one wishes to use to express properties and then "code up" these operators as macros. For example, it is possible to define the operators of the temporal logic CTL [6] as macros. Examples include the following.

$$\begin{aligned} AG\ \Phi &= \nu X.(\Phi \wedge [.]X) \\ AF\ \Phi &= \mu X.(\Phi \vee (\langle . \rangle tt \wedge [.]X)) \\ Until\ \Phi\ \Psi &= \nu X.(\Phi \vee (\Psi \wedge [.]X)) \end{aligned}$$

$$Until2\ \Phi\ \Psi \ = \ \mu X.(\Phi \vee (\Psi \wedge \langle . \rangle tt \wedge [.]X))$$

$AG\ \Phi$ holds of n if Φ holds of every node reachable (via some sequence of transitions) from n, while $AF\ \Phi$ holds if Φ is guaranteed to hold at some point in the future. *Until1* $\Phi\ \Psi$ holds of n if, along every maximal path of nodes starting at n, Ψ is true until a state is reached where Φ is true. *Until2* $\Phi\ \Psi$ is the same as *Until1* $\Phi\ \Psi$, except that here Φ additionally is required to hold eventually. *Until1* corresponds to the CTL "weak" until, while *Until2* corresponds to the CTL "strong" until operator (over all paths). It is also possible to write formulas expressing properties that are useful in describing fairness constraints; many of these involve the use of mutually recursive greatest and least fixed point formulas [13].

6.2 The Algorithm

The algorithm for determining whether a node satisfies a system logic formula works on *sequents* of the form $H \vdash n \in \Phi$, where n is a node, Φ is a formula, and H is a set of *hypotheses*, or assumptions of the form $n' : \nu X.\Phi'$. The (informal) interpretation of this sequent is that under the assumptions H, n satisfies Φ. The procedure is *tableau-based*, meaning that it attempts to build a top-down "proof" of $H \vdash n \in \Phi$. The method used comes from [7]; we shall not described it here. Another tableau-based approach appears in [33], while a *semantics-based* algorithm is given in [13]; an automated proof system for a subset of the logic is described in [22].

Applying the algorithm to graphs generated by the different graph transformations yields different notions of satisfaction. For instance, checking propositions against observation graphs causes the modal operators to be insensitive to τ-actions; one should also note that the observation graph transformation causes information about the eventuality properties of an agent to be lost, owing to the fact that every state in these graphs has an ε-transition to itself. Accordingly, every property of the form $AF\ \Phi$ will be false for every state that does not satisfy Φ.

As an example, it is possible to show that CBUF_n, for particular n, is deadlock-free as follows. Define the macro *Deadlock* by

$$Deadlock = \neg\langle . \rangle tt$$

This proposition is true of states that cannot perform any actions. Using the model checker, one can establish that CBUF_n satisfies the formula

$$AG(\neg Deadlock)$$

where AG is the macro defined above; this formula states that it is always the case that CBUF_n is not deadlocked. It is also possible to show that CBUF_n is *live*,

i.e. always capable of eventually engaging in either an *in* or an $\overline{out}$. The formula expressing this property is the following.

$$AG((AF\langle in\rangle tt) \vee (AF\langle \overline{out}\rangle tt))$$

For particular n, one can establish that CBUF_n satisfies this formula.

The algorithm in general has complexity that is exponential in the size of the formula being checked, although for special classes of formulas it is well-behaved. The precise complexity is still under investigation.

7 Other Features of the Workbench

The Workbench includes other facilities for examining the behavior of agents. In addition, as a result of its modular structure it is relatively easy to extend. This section describes some of these facilities and extensions.

7.1 State Space Analysis

The Workbench includes a variety of ways of analyzing the state space of an agent. In addition to commands for computing transitions and derivatives, there are features for determining which states are deadlocked and for computing sequences of experiments that lead to deadlocked states. These types of analyses are traditionally found in automatic verification tools and will not be discussed in this paper.

7.2 Equation Solving

The equation solving feature of the Workbench [28] is used to solve equations of type $(A|X)\backslash L = B$ where A, B and L are given. The method is useful within a top-down or stepwise refinement strategy: if a specification (B) and parts of an implementation (A) are known, solving such an equation amounts to constructing a specification of the missing submodules. The method works by successively transforming equations into simpler equations, in parallel with the generation of a solution. These transformations can be performed automatically by the system according to certain heuristics, or the user can apply them interactively. The tool has been used for the generation of a receiver in a communication protocol, where the overall service, medium, and sender are known.

7.3 Experimental Extensions

Two extensions to the system have been implemented and are being investigated. In the first, the model of computation has been extended to include a restricted form of *value passing*. In its "pure" form, CCS does not provide for the association

of values to communication events, although it is possible to encode the passing of values by associating a unique event name to an event/value pair. In the case of infinite value domains, however, this leads to syntactically infinite agents. In [20], an alternative encoding is proposed, in which the infinitely many data values are represented schematically. Using the resulting transitional semantics, bisimulation equivalences can be defined in such a way as to correspond exactly to the bisimulation equivalences in full CCS. This result entails a decision procedure for *data-independent* agents, i.e. agents which communicate data values but do not perform any computations or tests on the values. The decision procedure has been implemented in the Workbench [24].

An interface has also been built between the Workbench and the Extended Model Checker [6] (EMC), which is a tool for checking the satisfiability of temporal logic (CTL) formulas. EMC views processes somewhat differently than the other analysis procedures in the Workbench do; there are no communication events, and states are labeled by atomic propositions. EMC has successfully been applied to verification of nontrivial pieces of hardware. The integration with the Workbench was achieved by defining a translation ¿from labeled transition graphs to the type of structures analyzed by EMC [19].

8 Conclusion

In this paper we have presented an overview of the Concurrency Workbench. We have shown that it is possible to supply a variety of tools for deducing the correctness of processes based on a variety of different process semantics while maintaining a conceptually simple core. This has been achieved by maintaining a strict separation between the semantic models of processes and the procedures used to analyze them. This modularization makes the system relatively easy to extend.

There are a variety of directions for future work on the Workbench. Other equivalences and preorders, including GSOS equivalence [1] and the $\frac{2}{3}$-bisimulation preorder [23], also turn out to be instances of the general relations that we examine, and adding these relations to the workbench is one avenue we plan to pursue. Another involves the addition of some form of value-passing to the language. An experimental extension to the Workbench includes a procedure for determining equivalence for restricted kinds of value passing [20], and extending the ways values may be handled would be useful. Yet another involves the computation of *distinguishing formulas* [16]. At present, when agents are found not to be equivalent, no indication is given as to why. One way to convey such information is to give a formula in the mu-calculus satisfied by one agent but not by the other. Work is also underway on a graphical interface.

Another possible area of investigation involves *compositional reasoning* [5]. The parallel composition of two agents usually entails a combinatorial explosion in the

size of the state space of the resulting agent as a function of the state spaces of its components. One means of coping with this is to verify the parallel components separately in a way that implies the correctness of the composite process. The preorder has been investigated as !!a foundation for!! such a tool [34]; a possible extension to the Workbench would involve formalizing this in a way similar to the equation-solving tool. It is also conceivable that the model checker could be extended to check formulas compositionally using methods developed by Stirling [32].

Acknowledgements

We would like to thank Matthew Hennessy, Robin Milner and Colin Stirling for initiating and overseeing the Workbench project. We are also grateful to Lennart Beckman, Jo Blishen, Michael Mendler, Kevin Mitchell, Fredrik Orava, Björn Pehrsson, and David Walker for many helpful suggestions concerning the implementation of the Workbench and the development of this report.

References

[1] Bloom, B., S. Istrail and A. Meyer. "Bisimulation Can't Be Traced." Proceedings of the ACM Symposium on Principles of Programming Languages, 1988.

[2] Bochmann, G., "Usage of Protocol Development Tools: The Results of a Survey." In *Proceeding of the Seventh IFIP Symposium on Protocol Specification, Testing, and Verification*, 1987, North-Holland.

[3] Boudol, G., de Simone, R. and Vergamini, D. "Experiment with Auto and Autograph an a Simple Case Sliding Window Protocol." Inria Report 870, July 1988

[4] Clarke, E.M. "Compositional Model Checking." In *Proceeding of the the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, Grenoble 1989.

[5] Clarke, E.M. "Compositional Model Checking." In *Proceedings of the REX Summer School/Workshop*, Noordwijkerhout, the Netherlands, 1988.

[6] Clarke, E.M., Emerson, E. and Sistla, A.P. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM Transactions on Programming Languages and Systems*, v. 8, n. 2, 1986, pp. 244-263.

[7] Cleaveland, R. "Tableau-Based Model Checking in the Propositional Mu-Calculus." University of Sussex Technical Report 2/89, March 1989.

[8] Cleaveland, R. and Hennessy, M.C.B "Testing Equivalence as a Bisimulation Equivalence." In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, Grenoble 1989.

[9] Cleaveland, R., Parrow, J. and Steffen, B. *The Concurrency Workbench: Operating Instructions*, University of Edinburgh, Laboratory for Foundations of Computer Science, Technical Note 10, September 1988.

[10] Cleaveland, R., Parrow, J. and Steffen, B. "The Concurrency Workbench." In *Proceeding of the the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, Grenoble 1989.

[11] Cleaveland, R., Parrow, J. and Steffen,B.: "A Semantics-Based Verification Tool for Finite-State Systems." In *Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing, and Verification*, North Holland, Twente 1989.

[12] DeNicola, R. and Hennessy, M.C.B. "Testing Equivalences for Processes." *Theoretical Computer Science*, v. 34, 1983, pp. 83-133.

[13] Emerson, E.A. and Lei, C.-L. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus." In *Proceedings of the First Annual Symposium on Logic in Computer Science*, 1986, pp. 267-278.

[14] Fernandez, J.-C. *Aldébaran: Une Système de Vérification par Réduction de Processus Communicants.* Ph.D. Thesis, Université de Grenoble, 1988.

[15] Hennessy, M.C.B. *Algebraic Theory of Processes.* MIT Press, Boston, 1988.

[16] Hillerström, M. *Verification of CCS-processes.* M.Sc. Thesis, Computer Science Department, Aalborg University, 1987.

[17] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[18] Hopcroft, J. and Ullman, J. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, 1979.

[19] Jonsson, B., Kahn, A., and Parrow, J. "Implementing a Model Checking Algorithm by Adapting Existing Automated Tools." In *Proceeding of the the Workshop on Automatic Verification Methods for Finite State Systems*, LNCS, Grenoble 1989.

[20] Jonsson, B. and Parrow, J. "Deciding Bisimulation Equivalences for a Class of Non-Finite-State Programs." In *Proceedings of the Sixth Annual Symposium on Theoretical Aspects of Computer Science*, 1989. Lecture Notes in Computer Science 349, pp. 421-433. Springer-Verlag, Berlin, 1989.

[21] Kanellakis, P. and Smolka, S.A. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence." In *Proceedings of the Second ACM Symposium on the Principles of Distributed Computing*, 1983.

[22] Larsen, K.G. "Proof Systems for Hennessy-Milner Logic with Recursion." In *Proceedings of CAAP*, 1988.

[23] Larsen, K. and A. Skou. "Bisimulation through Probabilistic Testing." Proceedings of the ACM Symposium on Principles of Programming Languages, 1989.

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

[25] Malhotra, J., Smolka, S.A., Giacalone, A. and Shapiro, R. "Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems." In *Proceedings of the Workshop on Specification and Verification of Concurrent Systems*, University of Stirling, Scotland, 1988.

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

[27] Paige, R. and Tarjan, R.E. "Three Partition Refinement Algorithms." *SIAM Journal of Computing*, v. 16, n. 6, December 1987, pp. 973-989.

[28] Parrow, J. "Submodule Construction as Equation Solving in CCS." In *Proceedings of the Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science 287, pp. 103-123. Springer-Verlag, Berlin, 1987.

[29] Parrow, J. "Verifying a CSMA/CD-Protocol with CCS." In *Proceeding of the Seventh IFIP Symposium on Protocol Specification, Testing, and Verification*, 1987, North-Holland.

[30] Richier, J., Rodriguez, C., Sifakis, J. and Voiron, J.. "Verification in XESAR of the Sliding Window Protocol." In *Proceedings of the Seventh IFIP Symposium on Protocol Specification, Testing, and Verification*, 1987, North-Holland.

[31] Steffen, B. "Characteristic Formulae." In Proceedings ICALP, 1989.

[32] Stirling, C. "Modal Logics for Communicating Systems." *Theoretical Computer Science*, v. 49, 1987, pp. 311-347.

[33] Stirling, C. and Walker, D.J. " Local Model Checking in the Modal Mu-Calculus", In Proceedings TAPSOFT, 1989.

[34] Walker, D.J. "Bisimulation Equivalence and Divergence in CCS." In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, 1988, pp. 186-192.

[35] Walker, D.J. "Analysing Mutual Exclusion Algorithms Using CCS." University of Edinburgh Technical Report ECS-LFCS-88-45, 1988.