The Concurrency Workbench [*]

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 [8], a tool for the automatic analysis of finite-state processes. Such tools are practically motivated; the formal detail arising from the rigorous verification of complex distributed systems rapidly becomes unmanageable without some form of computer assistance. This fact, together with the lack of consensus regarding the formal semantics of processes, has led to the implementation of a number of automated tools, each embodying a particular semantic model of computation and each capable of particular forms of correctness analysis. Notable examples include EMC [5], XESAR [28], Aldébaran [12], AUTO [3], and Winston [23]; for a survey of systems for the analysis of communication protocols see [2]. A major goal of the Workbench is to provide a uniform framework that supports several different semantics for processes and several methods for reasoning about systems.

The Workbench includes three main methods for establishing that processes meet specifications. In the first, specifications are themselves processes that describe precisely the high-level behavior required of an implementation. To verify that a system meets such a specification, one shows them to be *equivalent* in the sense of having the same behavior. Here different notions of "same behavior" yield different equivalences and hence imply different semantic models of processes. The Workbench is capable of computing a variety of different behavioral equivalences.

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 "loose" specifications if it supplies at least the behavior demanded by the specification while "filling in" some holes. This approach 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 Workbench can automatically determine if a process is more defined than its specification in this sense. As with the

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

[†]Computer Science Department, North Carolina State University, Box 8206, Raleigh NC 27695, USA

[‡]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

equivalences above, different notions of "behavior" yield different ordering relations. The Workbench supports a number of such orderings.

The third approach to verification involves the use of an expressive modal logic, the propositional (modal) mu-calculus. Propositions formulated in this logic are viewed as specifications; it is possible to determine automatically whether a process satisfies such a specification using the *model checking* facility of the Workbench. The Workbench supports different ways of checking that such propositions are true of an agent, depending on the semantic model used.

The system provides a variety of other means for analyzing the behavior of processes, including deadlock detection and simulation. There is also a feature that supports a restricted form of hierarchical development of complex systems and one that generates the minimum-state process that is equivalent to a given process.

The Workbench has been successfully applied to the analysis of communication protocols [27] and mutual exclusion algorithms [33]; it has also been used to debug the Edinburgh Computer Science Department's electronic mailing system. Moreover, it is being investigated as a tool for analyzing communications protocols by Swedish Telecom and by Hewlett-Packard. Our primary goal, however, is to demonstrate viable principles for automatic verification. For example, one main result is that a wide spectrum of different verification methods can be obtained by combining a small number of more primitive algorithms; this makes the Workbench a versatile and easily extensible tool.

The remainder of the paper is organized as follows. Section 2 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 8 contains our conclusions and directions for future work.

2 The Architecture of the Workbench

One goal in the design of the Workbench is to make available many different approaches for verification while maintaining a conceptually economical core. Accordingly, the main components of the system are organized into three layers.

- The *interface layer* oversees the interaction between the user and the Workbench; it includes routines which read processes and propositions from input and it prints the results of commands issued by the user.

- The *semantics layer* consists of procedures for transforming *transition graphs*, which the system generates from the operational account of processes in the first layer. These graph transformations enable a change in the underlying (denotational) model of agents. Examples include transformations for abstracting away from internal computation and for recording information about divergence and nondeterminism.

- The *analysis layer* consists principally of three algorithms: one for equivalence checking, one for preorder checking and one for checking whether a transition graph satisfies a modal proposition. These algorithms are *polymorphic*; they may be applied to any transition graph generated by the second layer.

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 number of automatic verification methods, 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.

3 Representing 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 operators from [24].

Nil	Termination
$\perp$	Divergence (or *bottom*)
$a.$	Prefixing by action a; unary prefix operator
$+$	Choice; binary infix operator
$\mid$	Parallel composition; binary infix operator
$\setminus L$	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 [24]. Examples of agents defined in CCS appear in Figure 1.

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 2 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.

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

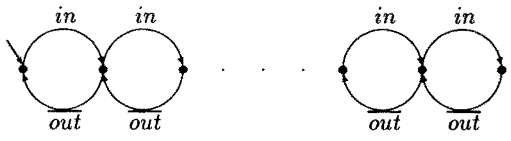
- CBUF_n defines a *compositional* buffer of capacity n.

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

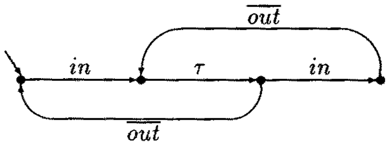
- The *partial* buffer of capacity n, PBUF_n, specifies agents that behave like buffers of capacity n, so long as no more than n elements are stored at once.

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

Figure 1: 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 2: Examples of Transition Graphs.

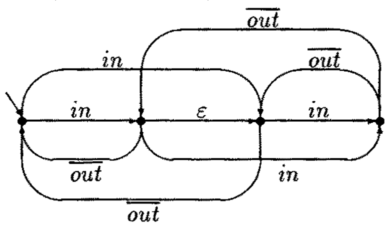


Figure 3: The Observation Graph for CBUF_2.

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.

$$n \overset{\varepsilon}{\Longrightarrow} n' \ \text{ iff } \ n \xrightarrow{\tau}{}^{*} n'$$
$$n \overset{a}{\Longrightarrow} n' \ \text{ iff } \ n \overset{\varepsilon}{\Longrightarrow} \xrightarrow{a} \overset{\varepsilon}{\Longrightarrow} n'$$

So $\overset{\varepsilon}{\Longrightarrow}$ is defined as the transitive and reflexive closure of $\xrightarrow{\tau}$, and $\overset{a}{\Longrightarrow}$ is defined in terms of relational products of $\overset{\varepsilon}{\Longrightarrow}$ and $\xrightarrow{a}$. 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.

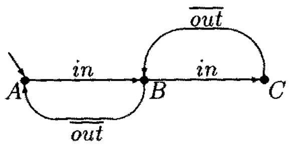
The observation graph transformation takes a graph and modifies the edges to reflect the $\overset{a}{\Longrightarrow}$ and $\overset{\varepsilon}{\Longrightarrow}$ relations instead of the $\xrightarrow{a}$ and $\xrightarrow{\tau}$ relations. It uses well-known methods for computing the product of two relations and the transitive and reflexive closure of a relation. Figure 3 indicates the nature of the transformation (for clarity, we have omitted the ε-edges resulting from the reflexive closure of $\xrightarrow{\tau}$).

A variation of the observation transformation computes *congruence graphs*, which are used to check for observational *congruence* [24] and weak *precongruence* [32]. 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 observational 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 process makes such a choice. However, other equivalences and preorders, like the *testing* preorders and equivalences [13] and *failures* equivalence [15], do not require such detailed accounts of the choice structure of agents. In order to cater for these equivalences the Workbench includes procedures for transforming graphs into various kinds of *deterministic* graphs, which are graphs having no τ-derivatives and at most one a-derivative for any action a.

The deterministic graphs constructed are all variants of *acceptance* graphs (also called *Tgraphs* in [7]). Acceptance graphs record the three pieces of information which are necessary in order to compute



where

$$\begin{aligned} A.acc &= \{\{in\}\} \\ B.acc &= \{\{in, \overline{out}\}\} \\ C.acc &= \{\{\overline{out}\}\} \end{aligned}$$

and $A.div = B.div = .C.div = false$.

Figure 4: The Tgraph for CBUF$_2$.

the testing equivalenced and preorders. The first is the *language* of the process, which consists of the set of all sequences of visible actions available from the start state. The second is the *divergence potential*, or the possibility of an infinite sequence of τ-actions, as the process attempts to "execute" a sequence of visible actions. The final piece of information necessary is the *degree of nondeterminism*, which is recorded in the form of *acceptance sets*. An acceptance set is a set of sets of actions, one set of actions for each state a process can "settle down in" (i.e. not exit via an invisible τ-action) after executing a given sequence of visible actions. The fact that an acceptance set may contain more then one set of actions indicates that a process may be capable of settling down in more than one state as the result of nondeterministic choices during the execution of a sequence. Formal definitions for these concepts may be found in [7].

To record divergence and acceptance set information, each state in an acceptance graph includes two fields: *div*, a boolean representing divergence information, and *acc*, which is a set of set of actions. In what follows, we shall refer to these as *t.acc* and *t.div* for nodes t of a given acceptance graph. (In fact, the *div* field is not necessary, strictly speaking, since the information it contains can be encoded in the *acc* field. That is, *div* is true exactly when *acc* is empty. For clarity, however, we shall continue to use it.)

The Workbench includes a procedure that, given a graph, generates a Tgraph whose root has the same language as the root of the original graph and whose nodes are labeled by *acc* and *div* fields. A node t in the Tgraph reachable by the sequence s has the labeling $t.div = false$ and $t.acc = A(n_0, s)$, where n_0 is the root of the original graph, if it is impossible to reach a divergent node from n_0 during the exection of s. Otherwise, $t.acc = \emptyset$ and $t.div = true$. The procedure for generating these Tgraphs is described in [7]; Figure 4 shows the Tgraph resulting from the transformation of CBUF$_2$.

Two closely related kinds of deterministic graphs are also constructed by the workbench. *May graphs* (referred to as *Dgraphs* in [7]) are like Tgraphs except that the *div* and *acc* fields are not computed. *Must graphs* (referred to as *STgraphs* in [7]) are like Tgraphs, except that no node n with $n.div = true$ has any transitions emanating from it; these are constructed from Tgraphs by "clipping" the Tgraph at nodes whose *div* fields are true.

In general, these graph construction procedures have exponential complexity, owing to the fact that it is theoretically possible to have a node in these graphs for each subset of old nodes. Practice, however, indicates that the number of nodes is usually smaller than the number of nodes in the original graph.

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

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 some 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. $\langle m, n\rangle \in \mathcal{C}$.

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 $\mathcal{U}$ denote the universal relation, i.e. $\mathcal{U} = N \times N$. Then a $\mathcal{U}$-bisimulation is a bisimulation in the sense of Milner [24], and $\mathcal{U}$-equivalence is strong equivalence in CCS. Also, observation equivalence corresponds to $\mathcal{U}$-equivalence on observation graphs, observation congruence to $\mathcal{U}$-equivalence on congruence graphs, and trace (or may) equivalence to $\mathcal{U}$-equivalence on deterministic graphs. For must and testing (failures) equivalence define $\langle m, n\rangle \in \mathcal{A}$ to hold exactly when 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 [7].

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, except for the strong equivalence.

4.3 The Algorithm

Our algorithm is adapted from one presented in [19]. 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 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 [25]; 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 taken up 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

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 5.1 A $\mathcal{C}$-*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 divergent and cannot be triggered by means of an a-action to reach a divergent state. (For details of this predicate see [30, 32].) Then we have:

- The bisimulation divergence preorder [30, 32] results by setting:

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

  This defines the strong version of the divergence preorder. The weak version, $\lessapprox$, where τ-actions are not observable, can be obtained from the corresponding observation graphs.

- 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 [7].

- The may preorder: $\mathcal{P}_a = N$, $\mathcal{Q}_a = \emptyset$, and $\mathcal{C} = U$.
- The must preorder: $\mathcal{P}_a = \emptyset$, $\mathcal{Q}_a = \{m | m.div = \mathit{false}\}$, 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: $\mathcal{P}_a = N$, $\mathcal{Q}_a = \{m | m.div = \mathit{false}\}$, and $\mathcal{C}$ is defined as for the must preorder.

As an example, we can establish the following.

$$\mathrm{PBUF}_n \sqsubseteq \mathrm{BUF}_m, \text{ for all } m > n$$

Here PBUF_n is used as a partial specification that is satisfied by all BUF_m for $m > n$. Partial specifications like this can be used to express that two agents, although not equivalent, may be used interchangeably in certain contexts (cf. [9]).

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 $G_1 \not\sqsubseteq_{\mathcal{C}} G_2$) or the annotations determine a $\mathcal{C}$-prebisimulation (in which case $G_1 \sqsubseteq_{\mathcal{C}} G_2$).

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 [4, 5, 6, 11, 31], 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 describe 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.

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

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. [11, 29]). 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 [5] as macros. Examples include the following.

$$\begin{aligned}
AG\ \Phi &= \nu X.(\Phi \wedge [.]X) \\
AF\ \Phi &= \mu X.(\Phi \vee (\langle . \rangle tt \wedge [.]X)) \\
\mathit{Until1}\ \Phi\ \Psi &= \nu X.(\Phi \vee (\Psi \wedge [.]X)) \\
\mathit{Until2}\ \Phi\ \Psi &= \mu X.(\Phi \vee (\Psi \wedge \langle . \rangle tt \wedge [.]X))
\end{aligned}$$

$AG\ \Phi$ 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 along every path of nodes starting at n. *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 [11].

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 [6]; we shall not described it here. Another tableau-based approach appears in [31], while a *semantics-based* algorithm is given in [11]; an automated proof system for a subset of the logic is described in [20].

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; it is also interesting to note that the observation graph transformation causes information about the eventuality properties of a process to be lost.

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

$$\mathit{Deadlock} = \neg\langle . \rangle tt$$

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

$$AG(\neg \mathit{Deadlock})$$

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 [26] 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 involves the successive transformation of 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 additional 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 actions, although it is possible to encode the passing of values by associating a unique action name to an action/value pair. In the case of infinite value domains, however, this leads to syntactically infinite agents. In [18], 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 [22].

An interface has also been built between the Workbench and the Extended Model Checker [5] (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 [17].

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 [21], 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 computation of *distinguishing formulas* [14]. 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; a technique for doing so is under investigation. Work is also underway on a graphical interface.

Another possible area of investigation involves *compositional reasoning* [4]. 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, but in a way that implies the correctness of the composite process. The preorder has been investigated in this respect [32]; 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 [30].

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, Scott Smolka 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." This volume.

[5] 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.

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

[7] Cleaveland, R. and Hennessy, M.C.B "Testing Equivalence as a Bisimulation Equivalence." This volume.

[8] 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.

[9] Cleaveland, R., Parrow, J and Steffen, B. *The Concurrency Workbench: A Semantics-Based Verification Tool for Finite-State Systems*, University of Edinburgh, Laboratory for Foundations of Computer Science, Technical Report ECS-LFCS-89-83, June 1989.

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

[11] 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.

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

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

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

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

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

[17] Jonsson, B., Kahn, A., and Parrow, J. "Implementing a Model Checking Algorithm by Adapting Existing Automated Tools." This volume.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

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