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

Rance Cleaveland † Joachim Parrow ‡ Bernhard Steffen §

September 13, 1994

Abstract

The Concurrency Workbench is an automated tool for analyzing networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. Its key feature is its breadth: 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 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 [11, 12, 13], a tool that supports the automatic verification of finite-state processes. Such tools are practically motivated: the development of complex distributed computer systems requires sophisticated verification techniques to guarantee correctness, and the increase in detail rapidly becomes unmanageable without computer assistance. Finite-state systems, such as communications protocols and hardware, are particularly suitable for automated analysis because their finitary nature ensures the existence of decision procedures for a wide range of system properties.

A goal in the design of the Workbench is to incorporate several different *verification methods*, 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 supports mixed verification strategies which use more than one method, it facilitates a comparison between many techniques for formal verification, and it makes the system easily extensible. This versatility contrasts with existing automated

*Research supported by British Science and Engineering Research Council grant GC/D69464. This paper appeared in *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

†Computer Science Department, North Carolina State University, Box 8206, Raleigh, NC 27695, USA. Much of 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. Part of the work reported here was performed while the author was on leave at the University of Edinburgh, supported by a grant from the Science and Engineering Research Council.

§Lehrstuhl für Informatik II, RWTH Aachen, Ahornstraße 55, W-5100 Aachen, GERMANY

tools, which typically embody a particular semantics and a particular form of verification. Examples of such systems include Aldébaran [22], AUTO [3], CESAR [47], COSPAN [28], EMC [6], and Winston [42]. Other tools, such as SPIN [32], perform more specialized kinds of analysis (such as deadlock detection) and are used primarily to *validate* (as opposed to *verify*) existing real-world systems.

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*, which describe the behavior of processes in terms of the communication events they may engage in. The second layer provides transformations that may be applied to transition graphs. These transformations enable the user of the Workbench to change the semantic model of processes under consideration. The third layer includes the basic analysis algorithms for establishing whether a process meets a specification. Depending on the verification method used, a specification may either be another process (describing the desired behavior) or a formula in a modal logic expressing a relevant property.

The Workbench has been successfully applied to verifying communication protocols, notably the Alternating Bit Protocol, the CSMA/CD protocol [46] and the communication layer of the BHIVE multiprocessor [25], and mutual exclusion algorithms [55]; it has also been used to debug the Edinburgh Computer Science Department's electronic mailing system. It is currently being investigated as a tool for analyzing communications protocols by Swedish Telecom and by Hewlett-Packard, and it has been successfully used in education, in industry as well as in universities.

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 it supports. Section 3 presents the model of processes used and the process transformations that enable different semantics to be supported. Sections 4, 5 and 6 discuss the equivalence-checking, preorder-checking and model-checking facilities, respectively, while Section 7 describes actual sessions with the Workbench and concludes with a discussion of performance aspects of the system. Section 8 contains a brief account of some of the ways in which the system is being extended, and Section 9 presents our conclusions and directions for future work.

2 The Architecture of the Workbench

Figure 1 provides an overview of the Workbench. In order to supply a wide variety of approaches for verification while maintaining a conceptually economical core, the system is highly modularized. The system includes three major 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 may be stored in *environments* maintained by the Workbench. There is also a package defining the *basic semantics* of processes. Following [43], a process is interpreted as labeled transition graph that describes the states a process may enter and

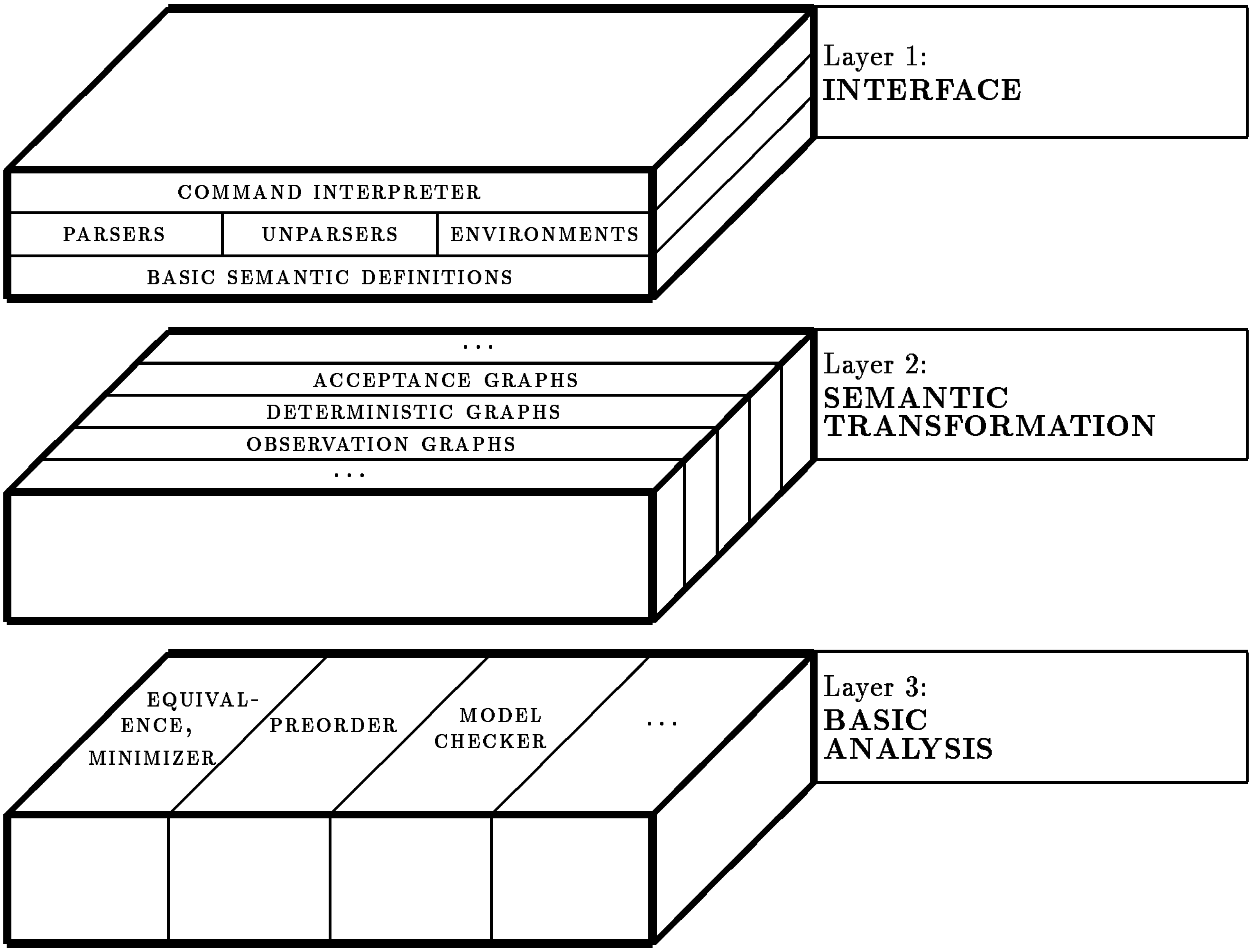


Figure 1: Overview of the Workbench

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 adds transitions by permitting visible actions to absorb sequences of internal computation steps; thus the resulting transition graphs do not record the timing of internal computations in the corresponding processes. The *deterministic* transformation makes the transition graphs deterministic in the usual sense of the word: the resulting graphs do not record nondeterministic choices or internal computation steps. The *acceptance* transformation yields deterministic graphs augmented with information regarding the potential for infinite internal computation and for nondeterminism (and hence deadlock) in the form of acceptance sets. These transformations 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 provides three main 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. The 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 of these 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; examples 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 [20, 50]. The *model-checking* algorithm determines whether a process satisfies such an assertion; it is described in Section 6.

The basic analysis algorithms are "polymorphic" in the sense that they work equally well on the different kinds of transition 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 transition graphs; if the transition 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. They enable changes in the process 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 (or *communication*) actions are divided into input actions and output actions. In the following $a, b, \ldots$ will range over input actions, and $\overline{a}, \overline{b}, \ldots$ will range over output actions. Input action a and output action $\overline{a}$ are said to be *complementary*, reflecting the fact that they represent input and output on the "port" a. We consider only communication actions without value parameters. Agents are defined using the following standard operators from [43].

Nil	Terminated process
$\perp$	Undefined process
$a.$	Prefixing by action a; unary prefix operator
$+$	Choice; binary infix operator
$\mid$	Parallel composition; binary infix operator
$\setminus L$	Restriction on (finite) set L of actions; 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 whose $b_1, b_2, \overline{b_1}$ and $\overline{b_2}$ transitions are relabeled to 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_A that may itself contain A. This enables recursively defined processes.

Agents are given an operational semantics defined in terms of a *transition relation*, $\xrightarrow{a}$, where a is an action. Figure 2 defines this relation formally. 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 to produce a τ action. $p \setminus L$ acts like p with the exception that no actions in L are allowed, while $p[f]$ behaves like p with actions renamed by f. The agents Nil and $\perp$ are incapable of any transitions; the former represents a terminated process, while the latter can be thought of as a "don't care" state, or as an agent whose behavior is unknown.

An agent is said to be *(globally) divergent* if it is capable of an infinite sequence of τ actions or if it may reach a state containing an *unguarded* occurrence of $\perp$ by performing some number of τ actions. Here, one agent expression, E, is *guarded* in another, E', if $a.E$ is a subexpression of E'

$$
\begin{array}{rcl}
a.p \xrightarrow{a} p & & \\
p \xrightarrow{a} p' & \Rightarrow & p+q \xrightarrow{a} p' \\
q \xrightarrow{a} q' & \Rightarrow & p+q \xrightarrow{a} q' \\
p \xrightarrow{a} p' & \Rightarrow & p|q \xrightarrow{a} p'|q \\
q \xrightarrow{a} q' & \Rightarrow & p|q \xrightarrow{a} p|q' \\
p \xrightarrow{a} p', q \xrightarrow{\overline{a}} q' & \Rightarrow & p|q \xrightarrow{\tau} p'|q' \\
p \xrightarrow{a} p', a, \overline{a} \notin L & \Rightarrow & p \setminus L \xrightarrow{a} p' \setminus L \\
p \xrightarrow{a} p' & \Rightarrow & p[f] \xrightarrow{f(a)} p'[f] \\
p_A \xrightarrow{a} p' & \Rightarrow & A \xrightarrow{a} p' \text{ where } p_A \text{ is the agent expression bound to identifier } A
\end{array}
$$

Figure 2: The Operational Semantics of CCS.

for some action a. If E is unguarded in E', then E is "top-level" with respect to E' in the sense that the initial transitions available to E also affect the initial transitions of E'.

Examples of agents defined in CCS appear in Figure 3.

3.2 Transition Graphs

The Workbench uses *transition graphs* (or *rooted labeled transition systems*) 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 an action-indexed family of sets of edges (corresponding to transitions between processes). If an edge comes from a set labeled by a, we say that it is labeled by a. An edge labeled by a has source n and target n' iff $p \xrightarrow{a} p'$ holds of the corresponding processes. The root is indicated by an unlabeled arrow. Figure 4 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.

It should be noted that the transition graph corresponding to an agent is not necessarily finite, owing to the fact that CCS permits the definition of processes that can create an arbitrary number of subprocesses. For example, an "unbounded counter" p can be defined as $i.(d.Nil \,|\, p)$. Here i stands for increment and d for decrement; when an increment occurs the counter creates a new copy of itself that then runs in parallel with $d.Nil$, the subprocess that can respond to decrement requests. The graph of p can be seen to have infinitely many states. Of course the Workbench cannot construct such graphs explicitly, and when presented with agents such as p the algorithm for graph generation will not terminate. In general, most interesting decision problems are undecidable on agents with infinite state spaces.

- BUF_n defines a buffer of capacity n.

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

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

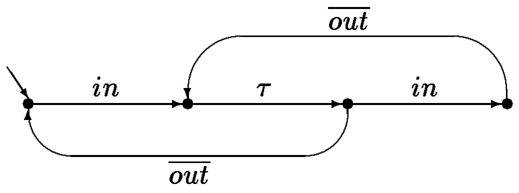
$$\mathrm{CBUF}_n \quad = \quad (\,\mathrm{BUF}_1\,[x_1/out] \mid \underbrace{\ldots \mid \mathrm{BUF}_1\,[x_i/in, x_{i+1}/out] \mid \ldots}_{i=1,\ldots,n-2} \mid \mathrm{BUF}_1\,[x_{n-1}/in]\,) \setminus \{x_1, ., x_{n-1}\}$$

- The *partial buffer* of capacity n, PBUF_n, specifies agents that behave like buffers of capacity n, provided that no attempt is made to insert additional elements when the buffer is full.

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

Figure 3: 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 4: Examples of Transition Graphs.

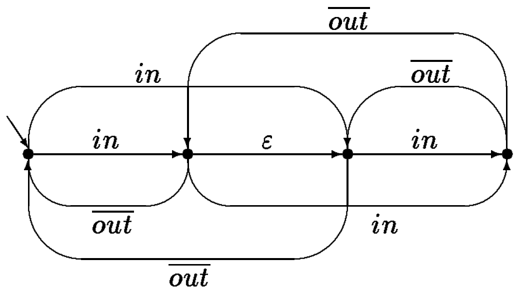


Figure 5: The Observation Graph for CBUF_2 .

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.

3.3.1 Observation Graphs

The transition graphs as described in Section 3.2 are synchronous in the sense that they faithfully represent τ actions, 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.

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

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

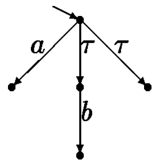
The observation graph transformation takes a graph and modifies the edges to reflect the $\overset{a}{\Longrightarrow}$ and $\overset{\varepsilon}{\Longrightarrow}$ relations instead of the $\overset{a}{\longrightarrow}$ and $\overset{\tau}{\longrightarrow}$ relations. It uses well-known methods for computing the product of two relations and the transitive and reflexive closure of a relation. Figure 5 indicates the nature of the transformation (for clarity, we have omitted the ε-loops resulting from the reflexive closure of $\overset{\tau}{\longrightarrow}$; there will be one such self-looping edge from each node).

The transformation takes time that is cubic in the number of nodes in the graph. In most cases the subroutine for transitive closure accounts for more than 80 per cent of the execution time when determining observation equivalence.

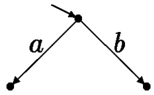
A variation of the observation transformation computes *congruence graphs*, which are used to check for observational *congruence* [43] and weak precongruence [54]. Intuitively, these graphs are observation graphs that record the possibility of initial τ-actions. To construct them, a copy of the root 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 transition 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 [29] (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 compute these relations the Workbench includes an algorithm for transforming transition graphs into language-equivalent *deterministic* graphs (also called *Dgraphs* in [9]), i.e. graphs having no τ-derivatives and at most one a-derivative per node for any action a. As a simple example, the transition graph



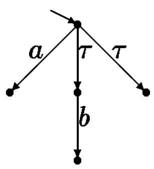
is transformed to



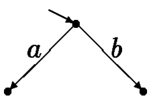
As another example, the deterministic graph of CBUF_2 is precisely the transition graph of BUF_2. The algorithm for computing deterministic graphs is well-known from automata theory (see e.g. [33]). In general, this transformation has an exponential complexity, owing to the fact that it is theoretically 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 [29, 31], require information about an agent's *divergence potential* 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* in [9]); 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 element in the acceptance set corresponds to a stable state (a state from which no internal action is immediately possible) in the original graph, and it contains precisely the actions that are immediately possible from this stable state. For example,

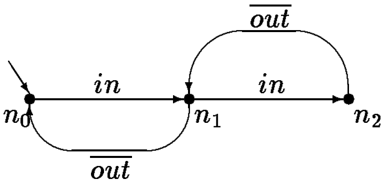


has the acceptance graph



where the root node has acceptance set

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



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

Figure 6: The acceptance graph for CBUF_2.

and each leaf node has acceptance set

$$\{\emptyset\}.$$

Here, 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 it is not possible to reach a stable state in which only an a is possible from the start state of the original graph. As another example, Figure 6 shows the acceptance graph resulting from the transformation of CBUF_2.

Acceptance sets may also be used to record divergence information. By convention, if an acceptance graph node has an empty acceptance set, then the execution of the sequence of actions leading from the root node to the present node can diverge, i.e. result 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 ([29]). Must graphs are like acceptance graphs, except that divergent nodes have no outgoing edges. The algorithm for generating acceptance and must graphs is described in [9].

4 Equivalence Checking

The first analysis procedure we present computes equivalences 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 transition graphs are 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 root nodes of the two transition 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. $\langle m, n\rangle \in \mathcal{C}$.

Two transition graphs are said to be $\mathcal{C}$-equivalent if there exists a $\mathcal{C}$-bisimulation relating the root nodes of the transition graphs.

4.2 Derived Equivalences

Many equivalences turn out to be instances of $\mathcal{C}$-equivalence combined with graph transformations.

- Let U denote the universal relation, i.e. $U = N \times N$. A U-bisimulation is a bisimulation in the sense of Milner [43] and U-equivalence is *strong equivalence* in CCS. *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.

- Let $\mathcal{A}$ be defined by: $\langle m, n\rangle \in \mathcal{A}$ 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. Two transition 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 [9].

As an example, recall the definitions for BUF_n and CBUF_n (see Section 3). For each 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 [37]. It works by attempting to find a $\mathcal{C}$-bisimulation relating the root nodes of the transition graphs. To do so, it maintains a *partitioning* of the nodes in G_1 and G_2, the transition 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 transition graphs: two nodes are related precisely when they belong to 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 transition graphs end up in different blocks (in which case the transition graphs are not equivalent) or the induced equivalence relation on the nodes becomes a $\mathcal{C}$-bisimulation (in which case the transition graphs are $\mathcal{C}$-equivalent). In order to determine whether a partition needs further refinement, the algorithm examines each block in the partition. If a node in a block B_1 has an a-derivative in a block B_2, then the algorithm examines all the other nodes in B_1 to see whether they also have a-transitions into the same block. If some nodes do not, then the block is split into the set of those nodes that do and the set of those nodes that do not, resulting in a refined partition.

The time and space complexities of this algorithm are $O(k * \ell)$ and $O(k + \ell)$ respectively, where k is the number of nodes, and ℓ is the number of transitions, in the two transition graphs. In [44] an algorithm is proposed whose worst case time complexity is $O(log(k) * \ell)$; 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 graph construction and transformation.

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 graph 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 computes preorders 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 preorder is based upon the following generalization of the notion of equivalence introduced in Section 4.1.

5.1 Definition of the Preorder

The general preorder used by the Workbench is similar to the general equivalence discussed in Section 4.1 in that it is also based on a notion of node matching; however the requirements on matched nodes are relaxed somewhat. Intuitively, one transition graph is less than another if the states of the first can be matched to those in the second in such a way that

1. if a state in the lesser graph is matched to a state in the greater, then the information field of the former must be "less" than the latter (the appropriate notion of "less" will in general depend on the precise preorder being computed);

2. if a state in the lesser graph is matched to a state in the greater, and the latter has valid a-transitions, then each a-transition of the former must be matched by some a-transition of the latter (the appropriate notion of "valid" will depend on the preorder being computed);

3. if a state in the lesser graph is matched to a state in the greater, and the former has "viable" a-transitions, then each a-transition of the latter must be matched by some a-transition of the former (the appropriate notion of "viable" will depend on the preorder being computed); and

4. the start state of the lesser is matched to the start state of the greater.

To formalize this, 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" (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 $\mathcal{Q}_a$ capture the notions of "viable" and "valid" mentioned above; in other words, they are used to determine the states from which a-transitions must be matched.

Definition 5.1 A parameterized *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 parameterized preorder is defined by: $G_1 \sqsubseteq_{\mathcal{C}}^{\mathcal{P},\mathcal{Q}} G_2$ if there exists a parameterized prebisimulation relating the roots of the two transition graphs.

Note that when $\mathcal{P}_a = \mathcal{Q}_a = N$ and $\mathcal{C}$ is an equivalence relation, then a parameterized prebisimulation is just a $\mathcal{C}$-bisimulation.

5.2 Derived Preorders

Many preorders turn out to be instances of the parameterized preorder combined with graph transformations. 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 [52, 54].

- The bisimulation divergence preorder [52, 54] results by setting:

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

  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 [9].
  - 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.

A preorder can be regarded as a *specification-implementation* relation in which $P \lessapprox Q$ is interpreted as "Q is closer to an implementation than P". This interpretation is based upon regarding divergent states as being *underspecified.* For example, $\perp$ can be seen as the totally unspecified state that allows any process as a correct implementation. We shall sometimes call processes containing divergent nodes *partial* specifications, in contrast to *complete* specifications that do not possess any divergent nodes. Partial specifications are interesting in their own right, since a system designer might want to allow freedom for the implementation. In addition, the divergence preorder yields a *compositional proof technique* for bisimulation equivalences that enables partial specifications to be used in proofs that implementations are equivalent to complete specifications (cf. [14, 15, 39, 54]). The key observation underlying this technique is that some processes, although not 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:

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

Now assume that we have already proved the protocol correct when MEDIUM is replaced by the partial buffer of capacity n PBUF_n, for some n:

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

We may then proceed to prove that

$$\text{PBUF}_n \lessapprox \text{BUF}_m$$

for any particular $m > n$. Since "$|$" and "$\backslash L$" contexts preserve $\lessapprox$, we therefore obtain:

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

Together with (2) and the transitivity of $\lessapprox$, this enables us to conclude the following.

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

It is easy to see that the preorder $\lessapprox$ 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 the 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 parameterized preorder works by attempting to find a parameterized prebisimulation relating the roots of the transition 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.

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

Figure 7: The syntax of formulas

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 reduces the annotation of each node until the root node of G_2 no longer is in the annotation of the root node of G_1 (in which case $G_1 \not\sqsubseteq_{\mathcal{C}}^{\mathcal{P},\mathcal{Q}} G_2$) or the annotations determine a $\mathcal{C}$-prebisimulation (in which case $G_1 \sqsubseteq_{\mathcal{C}}^{\mathcal{P},\mathcal{Q}} G_2$). The reduction of the annotation of a node proceeds according to two rules. First, if the node has an a-derivative n then each node in its annotation that satisfies $\mathcal{P}_a$ must also have an a-derivative that is in the annotation of n; nodes not meeting this condition are deleted from the annotation. Second, if the node satisfies $\mathcal{Q}_a$ and a node n in its annotation has an a-derivative n' then the node must have an a-derivative having n' in its annotation; otherwise, n is deleted from the annotation as well.

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 transition graphs. In [16] an algorithm is proposed whose worst case time complexity is $O(\ell^2)$. The implementation of this algorithm is underway. The loss of efficiency compared with the equivalence algorithm is due to the fact that we cannot use the same compact representation of behavioral relations as in Section 4.3.

6 Model Checking

The Workbench also supports a verification method based on model checking [5, 6, 7, 20, 53], 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, an *interface logic* and a *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 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 grammar in Figure 7. In this description 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 positively, i.e. in the scope of an even number of negations. The 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 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 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 $\bigwedge_{i=0}^{\infty}\Phi_i$. Dually 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 tremendous expressive power to the logic (cf. [20, 50]). 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 and then "code up" these operators as macros. For instance, it is possible to define the operators of the temporal logic CTL [6] as macros. Examples include the following.

$$
\begin{aligned}
\mathit{AG}\ \Phi &= \nu X.(\Phi \wedge [.]X)\\
\mathit{AF}\ \Phi &= \mu X.(\Phi \vee (\langle.\rangle tt \wedge [.]X))\\
\mathit{AU1}\ \Phi\ \Psi &= \nu X.(\Phi \vee (\Psi \wedge [.]X))\\
\mathit{AU2}\ \Phi\ \Psi &= \mu X.(\Phi \vee (\Psi \wedge \langle.\rangle tt \wedge [.]X))
\end{aligned}
$$

AG Φ holds of n if Φ holds of every node reachable (via some sequence of transitions) from n, while AF Φ holds of n if along every sequence of transitions leaving n, some state n' satisfies Φ. *AU1* Φ Ψ 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. *AU2* Φ Ψ is the same as *AU1* Φ Ψ, except that here Φ additionally is required to hold eventually. *AU1* corresponds to the CTL "weak" until operator, while *AU2* 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 [20].

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 describe it here. Another tableau-based approach appears in [53], while a *semantics-based* algorithm is given in [20]; an automated proof system for a subset of the logic is presented in [38].

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 $\overline{out}$, by automatically verifying that CBUF_n satisfies the following formula.

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

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. A linear-time algorithm for a particular subclass of this logic has been proposed in [17] and will be incorporated in future versions of the Concurrency Workbench.

7 Using the Concurrency Workbench

In this section we illustrate the main analysis procedures of the Workbench by verifying a very simple communications protocol. We then discuss a more substantial example in which we use the Workbench to debug a faulty implementation of the Alternating Bit Protocol. The section concludes with a discussion of performance aspects of the system.

7.1 A Simple Protocol

In the next three subsections we analyze the following simple communications protocol in order to illustrate how one uses the verification facilities implemented in the Workbench. The *service specification* of the protocol requires that any message sent must be received before a second message may be sent. The *protocol specification* requires two processes, a sender and a receiver, and a medium that connects them. After sending a message via the medium to the receiver, the sender blocks until the receipt of an acknowledgement enables it to send a new message. An acknowledgement is sent from the receiver as soon as it receives a message. It is also sent via the medium. After sending the acknowledgement the receiver is ready to receive another message.

What follows is an edited account of an actual session with the Workbench. We show how the above protocol may be formalized and verified in different frameworks using the equivalence, preorder and model checking features of the system.

When the Workbench is first invoked, the following appears on the screen.

```
          Welcome to the Concurrency Workbench! (V 5.0).

Command:
```

The system employs a "command loop"; it accepts commands from the user, prints its responses on the screen, and prompts for the next command. It is possible to define processes and propositions, and to invoke the analysis procedures this way.

7.2 Defining Agents and Checking Equivalence

In order to verify the communications protocol, we first must enter its service specification into the Workbench. For equivalence checking, we formalize specifications and implementations as processes. Processes can be entered using the command `bi` (for `b`ind `i`dentifier). In response to this command the system then prompts us for the corresponding arguments.

```
Command: bi
Identifier: SERVICE
Agent: send.'receive.SERVICE

Command:
```

In fact, the command above defines a process `SERVICE` that is capable of performing an infinite sequence of strictly alternating `send` and `'receive` actions. The `'` is Workbench notation for action complementation.

The protocol specification can be formalized as the parallel combination of three processes: one for the sender, one for the receiver, and one for the medium. The sender initially waits for a message to send, after which it passes the message to the medium using the channel `from` and then awaits an acknowledgement on the channel `ack_to`. When the medium receives a message along its channel `from` it makes it available on its channel `to`, and when it receives an acknowledgement on its channel `ack_from` it makes it available on its channel `ack_to`. When the receiver gets a message on channel `to`, it announces that the message is available for receipt and then sends an acknowledgement along the channel `ack_from`. The corresponding processes in the Workbench are defined using the following commands.

```
Command: bi
Identifier: SENDER
```

```
Agent: send.'from.ack_to.SENDER

Command: bi
Identifier: MEDIUM
Agent: from.'to.MEDIUM + ack_from.'ack_to.MEDIUM

Command: bi
Identifier: RECEIVER
Agent: to.'receive.'ack_from.RECEIVER

Command: bi
Identifier: PROTOCOL
Agent: (SENDER | MEDIUM | RECEIVER)\{from, to, ack_from, ack_to}
```

The use of the restriction operator `\{from, to, ack_from, ack_to}` in the definition of `PROTOCOL` ensures that the channels listed are "internal", i.e. not accessible to processes other than the sender and receiver. To view the process definitions entered so far, we may now invoke the `pe` (print environment) command.

```
PROTOCOL = (SENDER | MEDIUM | RECEIVER)\{from, to, ack_from, ack_to}
RECEIVER = to.'receive.'ack_from.RECEIVER
MEDIUM = from.'to.MEDIUM + ack_from.'ack_to.MEDIUM
SENDER = send.'from.ack_to.SENDER
SERVICE = send.'receive.SERVICE
```

In the equivalence checking verification method, we show the correctness of this protocol by establishing that it is semantically *equivalent* to the service specification. We do so by invoking the equivalence checking commands of the Workbench. In this example, we use the commands `eq` to determine whether the `PROTOCOL` and `SERVICE` are *observationally equivalent* and `testeq` to determine if they are *testing equivalent*. In each case, the Workbench prompts the user for the necessary process arguments.

```
Command: eq
Agent: PROTOCOL
Agent: SERVICE
true

Command: testeq
Agent: PROTOCOL
Agent: SERVICE
true
```

The responses show that the two processes are observationally equivalent and testing equivalent.

7.3 Preorder Checking

To illustrate the use of preorders in the Workbench we employ the technique described at the end of Section 5.2 to prove another version of the protocol correct. We modify the definition of the medium in the protocol specification; the new definition is partial, reflecting the fact that there may be semantically different implementations for the medium that still lead to a correct overall implementation of the service specification. As in Section 5.2 we then show how to use the preorder to deduce that the protocol is still correct when the specification of the medium is replaced by any of its implementations (i.e. any agent that it precedes in the preorder).

The new medium specification, and the protocol obtained by substituting this specification for the old medium, are given as follows.

```
Command: bi
Identifier: PARTIAL_MEDIUM
Agent: from.('to.PARTIAL_MEDIUM + ack_from.@) +
       ack_from.('ack_to.PARTIAL_MEDIUM + from.@)

Command: bi
Identifier: PARTIAL_PROTOCOL
Agent: (SENDER | PARTIAL_MEDIUM | RECEIVER)\{from, to, ack_from, ack_to}
```

The symbol `@` is Workbench notation for the agent $\perp$. Intuitively, after receiving a message on the channel `from`, `PARTIAL_MEDIUM` may either deliver it on its channel `to`, or receive a message on its channel `ack_from`, in which case its behavior is unspecified. The medium behaves "dually" in response to acknowledgements. It should be noted that `PARTIAL_PROTOCOL` is both observationally equivalent and testing equivalent to `SERVICE`; this may be checked using the equivalence checking commands illustrated above.

We now define an implementation of this medium specification. It consists of two one-place buffers running in parallel: one for messages, and one for acknowledgements.

```
Command: bi
Identifier: NEW_MEDIUM
Agent: MESSAGE_BUFFER | ACK_BUFFER

Command: bi
Identifier: MESSAGE_BUFFER
Agent: from.'to.MESSAGE_BUFFER

Command: bi
Identifier: ACK_BUFFER
Agent: ack_from.'ack_to.ACK_BUFFER
```

Using the preorder checker, it is now possible to show that the protocol implementation

```
NEW_PROTOCOL = (SENDER | NEW_MEDIUM | RECEIVER)\{to, from, ack_to, ack_from}
```

is *observationally* equivalent to `SERVICE`. We first check that the partial medium is less than `NEW_MEDIUM` in the bisimulation preorder using the command `wpr` (weak preorder).

```
Command: wpr
Agent: PARTIAL_MEDIUM
Agent: NEW_MEDIUM
true
```

One can also verify that PARTIAL_PROTOCOL never reaches an underspecified state using, for example, the model checking facility. Therefore NEW_PROTOCOL is equivalent to PARTIAL_PROTOCOL, and hence to SERVICE. Notice that we could substitute other implementations of PARTIAL_MEDIUM for NEW_MEDIUM and still conclude that the result is observationally equivalent to SERVICE; additional analysis of SENDER and RECEIVER is unnecessary.

7.4 Model Checking

To illustrate the model checking facility of the Workbench, we show how the service specification of the protocol definition may be recast as formulas in the temporal logic CTL in the Workbench, and then use the model checker to establish that PROTOCOL (as defined in Section 7.2) satisfies these formulas. The model checking facility is also useful in establishing that systems enjoy specific properties of interest, such as deadlock-freedom and freedom from divergence.

We first define some propositional macros using the Workbench command **bmi** (**b**ind **m**acro **i**dentifier).

```
Command: bmi
Identifier: AG

Enter parameters one at a time, terminating with "end".
Proposition parameters must begin with an uppercase letter,
action parameters otherwise.

Parameter: P
Parameter: end
Body: max(X. P & [.]X)
```

In a similar way, we can define the macros **Can** and **Can't**; the results of these definition can then be displayed using the **pmi** (**p**rint **m**acro **e**nvironment) command as follows.

```
Command: pme

Name:           Can't
Parameters:     a
Body:           ~(Can a)

Name:           Can
Parameters:     a
Body:           min(X.<a>T | <t>X)

Name:           AG
Parameters:     P
Body:           max(X. P & [.]X)
```

Intuitively, a state (process) satisfies **AG P** if every state reachable from the argument state satisfies P, while a state satisfies **Can a** if **a** may occur as the next visible action. **Can't** is the negation of **Can**; a state satisfies **Can't a** if, no matter how much internal computation is performed, an `a` is impossible. Here `max` is the maximum fixed point operator, while `min` is the minimum fixed point operator. In formulas, `|` is the disjunction operator and `&` the conjunction operator, while `T` represents the proposition *tt*. The internal τ action is represented by `t` in the Workbench. These macros may now be used to define formulas in the Workbench.

The `bpi` (bind propositional identifier) command can now be used to enter the four formulas that the service specification comprises. This results in the following proposition environment, which can be viewed by means of the `ppe` (print propositional environment) command.

```
SERVICE1 = AG ((Can send) | (Can 'receive))
SERVICE2 = AG ([send](Can 'receive) & ['receive](Can send))
SERVICE3 = AG ([send](Can't send) & ['receive](Can't 'receive))
SERVICE4 = Can send
```

The first formula states that the protocol is always in a state where either a `send` or a `'receive` may happen. The second says that it is always the case that after a `send`, the process can `'receive`, and vice versa, while the third prohibits two consecutive `send`'s or `'receive`'s from happening without an intervening visible action. The final formula stipulates that a `send` must initially be possible. Taken together, these formulas specify a process that engages an infinite alternating sequence of `send`'s and `'receive`'s, beginning with a `send`.

To verify the protocol against this specification, one invokes the command `csp` (check strong proposition) for the conjunction of the four formulas. The result is the following.

```
Command: csp
Agent: PROTOCOL
Proposition: SERVICE1 & SERVICE2 & SERVICE3 & SERVICE4
true
```

This establishes the correctness of the protocol specification in the logical framework.

7.5 Debugging a Protocol

In this section we develop a version of the well-known Alternating Bit Protocol [1]; although apparently correct, the system is faulty, and we show how to debug the system with the help of utilities supplied by the Workbench. The following analysis highlights the dramatic effect that often-unstated assumptions have on the correctness of concurrent systems; in this case the correctness of the protocol hinges subtly on the communication medium having a very specific property that is not mentioned in the informal description of the protocol. It is just such considerations that motivate the need for automated verification tools.

The Alternating Bit Protocol [1] is designed to ensure the reliable transfer of data over faulty communications lines; it is an example of a sliding window protocol (with window size one). In the protocol, sending and receiving processes alternate between two states in response to the receipt of messages (in the case of the receiving process) and acknowledgements (in the case of the sending process). Senders and receivers may also time out while waiting for acknowledgements and messages, respectively. A full account of the protocol may be found in [1].

Here we formalize a version of this protocol consisting of one sender and one receiver, with the sender sending data values consisting of a single bit over the given medium to the receiver. The general structure of our system appears in Figure 8. The sender receives the bits it is to send via

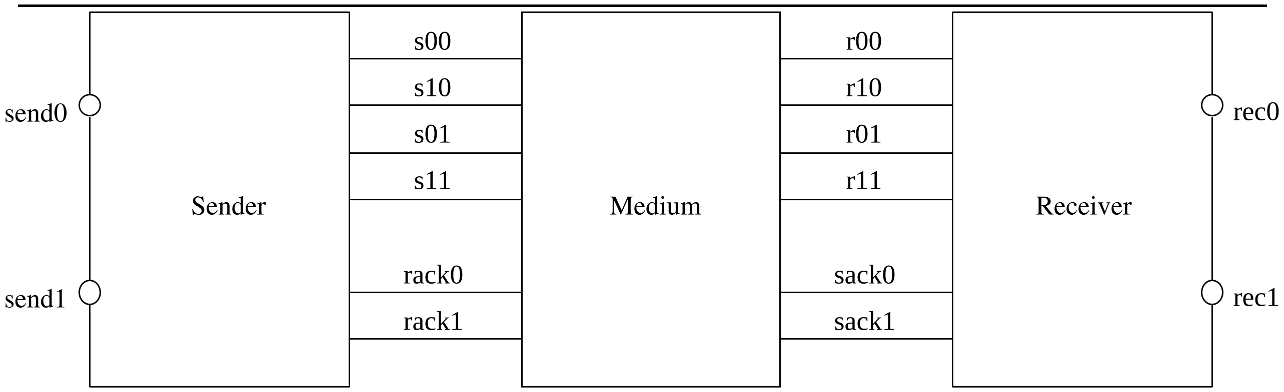
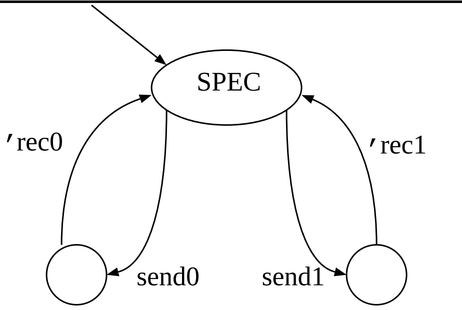


Figure 8: The network structure for the Alternating Bit Protocol.



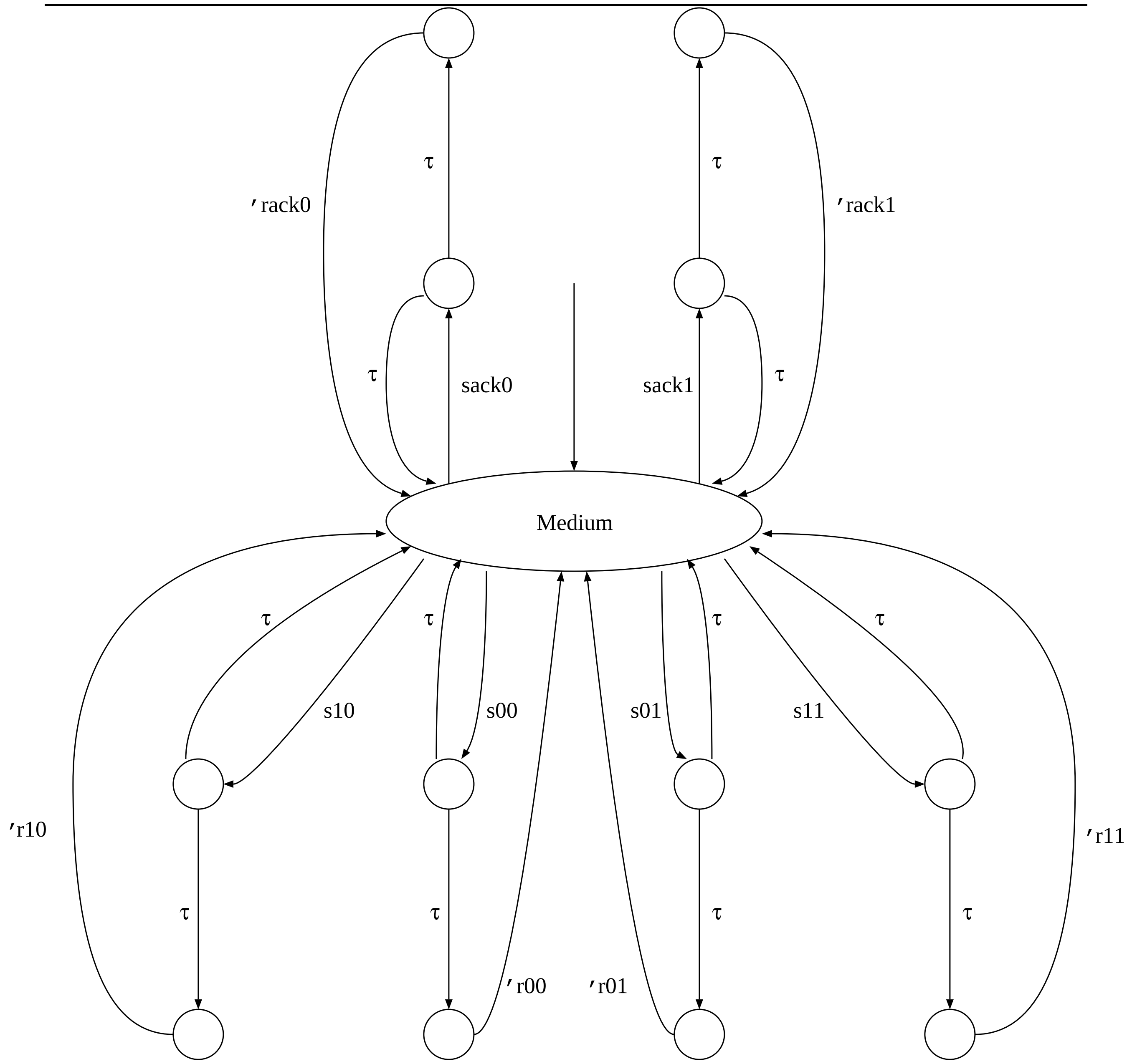

```
SPEC = send0.'rec0.SPEC + send1.'rec1.SPEC
```

Figure 9: The specification of the Alternating Bit Protocol.

the **send0** and **send1** channels from the users of the protocol; intuitively, users wishing to send a "0" will use **send0**, while users wishing to send a "1" will use **send1**.[1] Similarly, the receiver will deliver the values sent to it on ports **rec0** or **rec1**, depending on whether the value is 0 or 1. The medium conveys messages containing two bits—the data bit and a "sequence bit"—from the sender to the receiver and acknowledgements containing a sequence bit from the receiver to the sender. The specification of the protocol is that every message that is sent is correctly received; the formalization of this appears in Figure 9.

We now turn to the formalizations of the medium, sender and receiver. In each case, we give the corresponding finite-state machine as well as a listing of the associated Workbench code (obtained using the **pe** command). As is usual in the development of communication protocols, we assume that the medium to be used is given. In this case, the medium is unreliable; it may internally decide either to deliver a message that has been given to it or to lose it. This is modeled by the choice involving τ actions that appears after every **s** action; either the medium delivers the message (by

[1]This encoding of message values into port names is a standard technique for mimicking message passing in pure CCS.




```
Medium =  s00.(t.'r00.Medium + t.Medium) +
          s10.(t.'r10.Medium + t.Medium) +
          s01.(t.'r01.Medium + t.Medium) +
          s11.(t.'r11.Medium + t.Medium) +
          sack0.(t.'rack0.Medium + t.Medium) +
          sack1.(t.'rack1.Medium + t.Medium)
```

Figure 10: The medium.

allowing an `r` action), or it returns to its initial state (meaning that the message was lost). (Recall that `t` is Workbench notation for τ.) Figure 10 contains the formalization of the medium.

The sender accepts bits to be sent (by responding to the appropriate **send**i actions) and uses the medium to deliver them to the receiver. In order to detect when the medium has lost a message, the sender also appends a one-bit "sequence number" to each message it sends out. After giving a message to the medium, the sender awaits an acknowledgement from the receiver (via the medium) containing the same sequence number; if this happens, then the sequence number is incremented (modulo 2), and the sender is ready for the next value. However, if the sender receives an acknowledgement with the wrong sequence number, then it resends the same message (with the same sequence number) and awaits the appropriate acknowledgement. The sender may also time out (by executing a `t` action) while awaiting an acknowledgement, in which case it resends the last message it sent and waits for the acknowledgment. In our implementation, the sender uses the medium to send data value i with sequence number j by executing the action `'s`ij and awaits an acknowledgement with sequence number j by executing the action `rack`j. In states `S_0` (the start state), `S00` and `S10` the sequence number used is 0; in `S_1`, `S01` and `S11` it is 1. The sender appears in Figure 11.

The receiver awaits messages from the medium with a particular sequence number. If the sequence number of the message matches the one it expects, it makes the data value in the message available by executing the appropriate `'rec`i action and sends out an acknowledgement containing the sequence number; it then increments the sequence number (modulo 2) that it expects of the next message. If the sequence numbers do not match, then the receiver sends out an acknowledgement of the sequence number that it received and then waits for the sender to "resend". The receiver may also time out while waiting for a message, in which case it performs the same actions as when it receives a message with the wrong sequence number. In our implementation, the medium receives data value i with sequence number j by executing action `r`ij, and it sends an acknowledgement with sequence number j by executing `'sack`j. The receiver is given in Figure 12.

We now begin analyzing the protocol. The first thing we do is assemble the sender, medium and receiver into one unit, and restrict all actions involving the medium to be *local*, by executing the following.

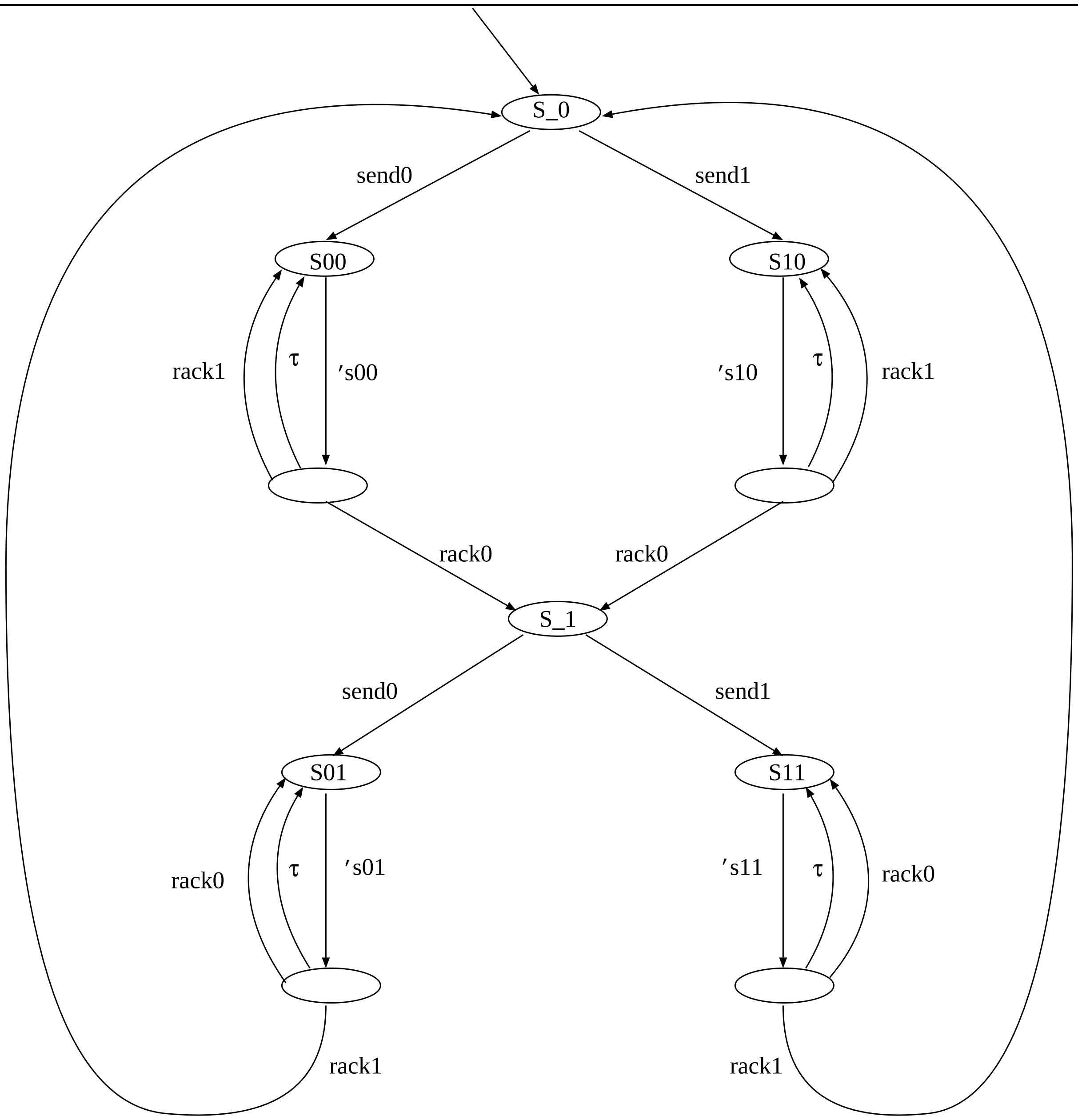
```
Command: bi
Identifier: ABP
Agent: (S_0 | Medium | R0)
        \{r00, r10, r01, r11, s00, s10, s01, s11,
          rack0, rack1, sack0, sack1}
```

Even though none of the components contains more than 13 states, the resulting system has 181.

To check the correctness of `ABP` we compare it with `SPEC` using the `eq` command; the time taken for this on a Sun SparcStation with 16 MB of memory is about one minute, which includes the construction of the finite-state machines from the CCS specifications.

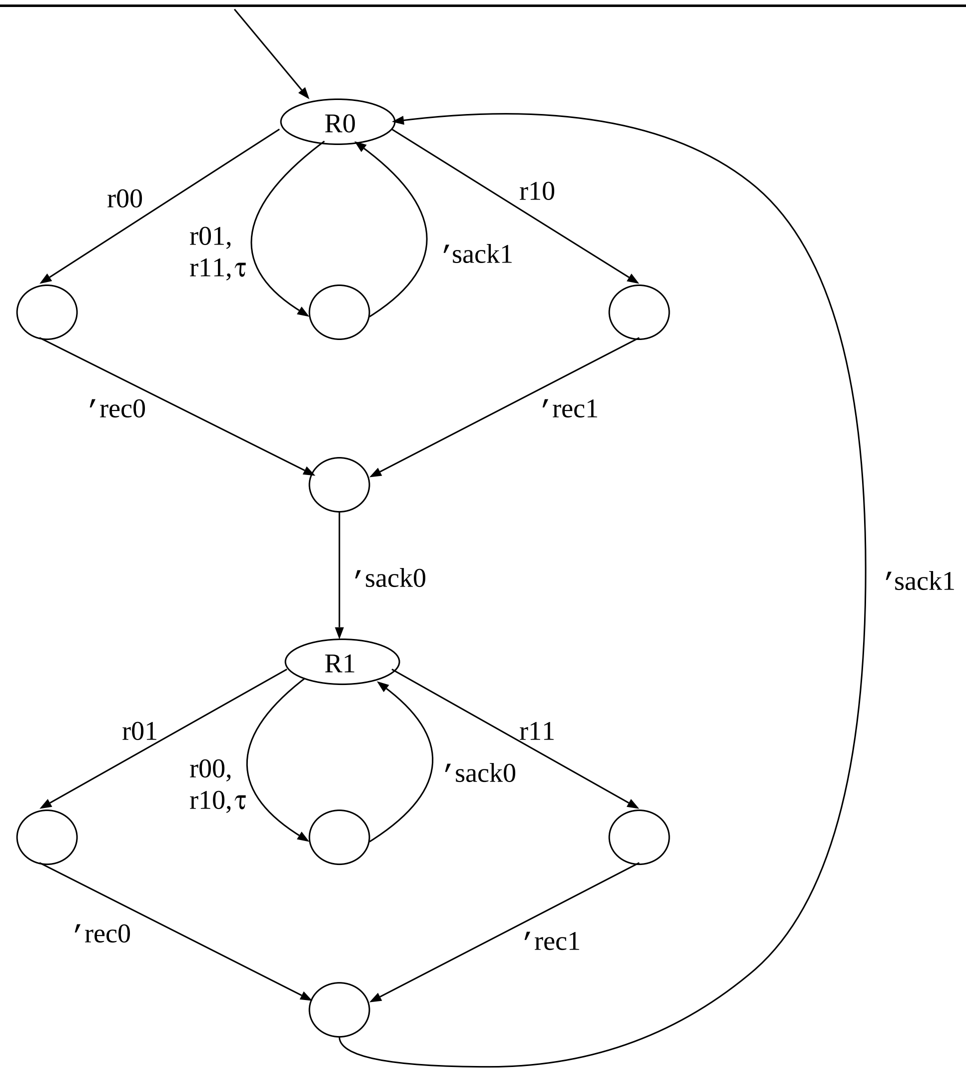
```
Command: eq
Agent: ABP
Agent: SPEC
false
```

So `ABP` is incorrect! This might seem surprising, since the correctness of Alternating Bit Protocol is well accepted; and, in fact, for slightly different definitions of the faulty `Medium` the implementation




```
S_0 =  send0.S00 + send1.S10
S00 = ’s00.(rack0.S_1 + rack1.S00 + t.S00)
S10 = ’s10.(rack0.S_1 + rack1.S00 + t.S00)
S_1 = send0.S01 + send1.s11
S01 = ’s01.(rack1.S_0 + rack0.S01 + t.S01)
S11 = ’s11.(rack1.S_0 + rack0.S11 + t.S11)
```

Figure 11: The sender.




```
R0 = r00.'rec0.'sack0.R1 + r10.'rec1.'sack0.R1 +
     r01.'sack1.R0 + r11.'sack1.R0 + t.'sack1.R0
R1 = r01.'rec0.'sack1.R0 + r11.'rec1.'sack1.R0 +
     r00.'sack0.R1 + r10.'sack0.R1 + t.'sack0.R0
```

Figure 12: The receiver.

just described *is* observationally equivalent to `SPEC`. As we shall see, the correctness of the protocol in fact depends on a property of media that `Medium` does not have, and fixing the system will entail changing our implementations of the sender and receiver to circumvent this problem.

In order to repair `ABP`, our first task is to locate the source of the faults in the system.[2] In general, the following strategy is useful in locating reasons why two systems are observationally inequivalent.

1. Check that the two processes have the same *sorts* (i.e. are capable of exactly the same actions). Often, errors arise because of typos in action names; these kinds of errors would be caught in this step.

2. Check whether the two processes are *language equivalent*. If they are not, then one can use the various state space exploration commands to isolate the source of the inconsistency.

3. Check for deadlocks in the implementation. Often, if two systems are language equivalent but not observationally equivalent, then the reason has to do with the potential for deadlock in one system.

To apply this strategy to debug `ABP`, we first compute the sorts of `SPEC` and `ABP` using the command `so`.

```
Command: so
Agent: SPEC
{'rec0, send0, 'rec1, send1}

Command: so
Agent: ABP
{send1, send0, 'rec1, 'rec0}
```

In this case, the two processes are capable of exactly the same actions. We now check for language (or may) equivalence; in this case, the result takes approximately a minute to compute on the same workstation.

```
Command: mayeq
Agent: ABP
Agent: SPEC
true
```

So there are no problems here, either. We now check for deadlocks. First, we use the model checker to determine if the system can deadlock. To do so, we define the proposition `Deadlock` (with the `bpi`, or "bind propositional identifier" command) and then examine whether the system is deadlock-free. In this case, the answer is computed in approximately 10 seconds.

```
Command: bpi
Identifier: Deadlock
Proposition: ~<.>T
```

[2]Future versions of the Workbench will have error diagnostic facilities based on techniques developed in [8], thus considerably easing the task of locating the faults in systems.

```
Command: csp
Agent: ABP
Proposition: AG(~Deadlock)
false
```

So the system is not deadlock-free! To locate specific deadlocked states in ABP, we use the command fd, which outputs a list of all deadlocked states together with a sequence of actions leading from the start state to the deadlocked state. The (partial) result in this case is the following; it takes approximately 90 seconds.

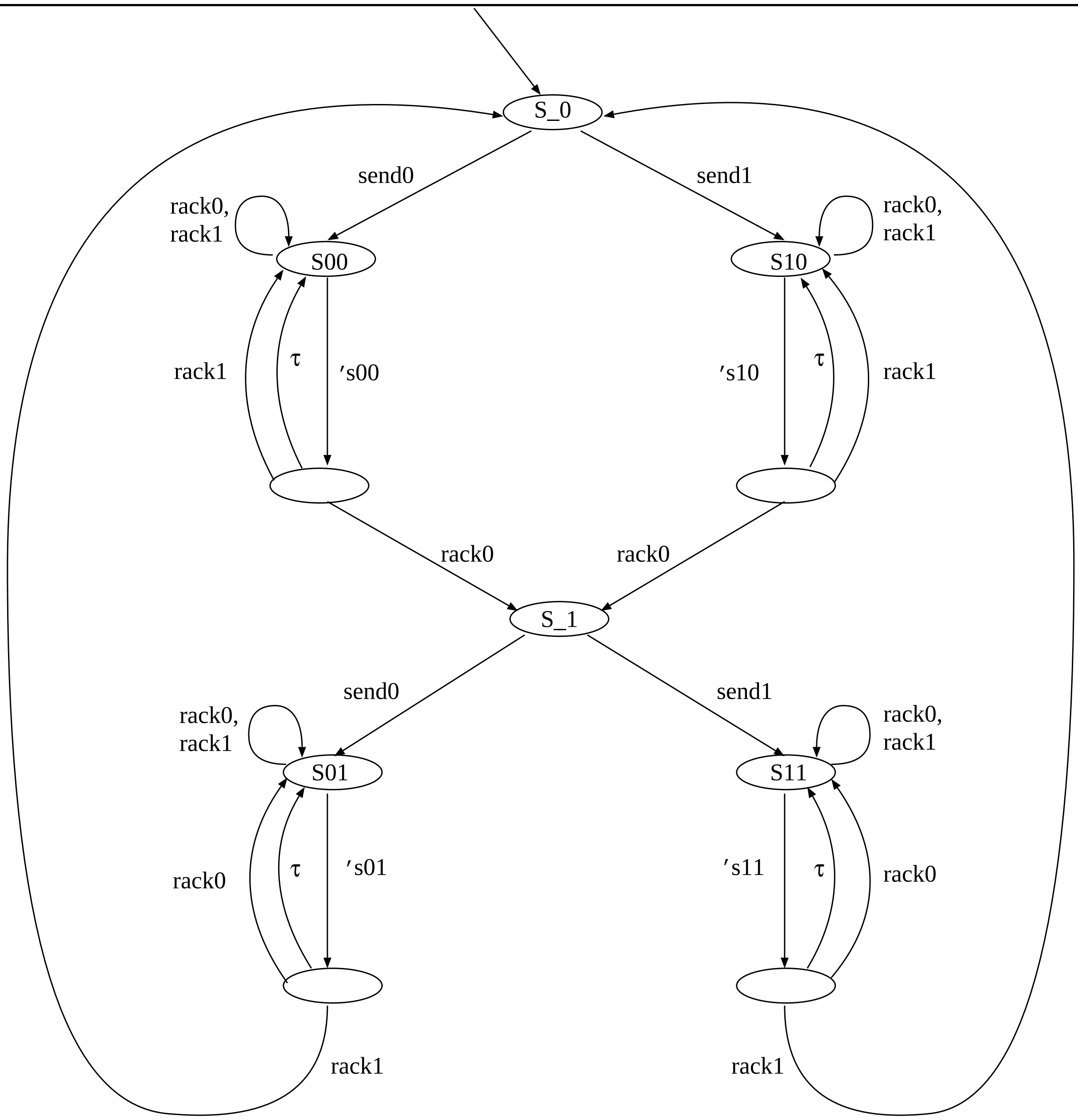
```
    Command: fd
    Agent: ABP

...

    ---send0 t t t t---> (S00 | 'r00.Medium | 'sack1.R0)
                         \{r00, r10, r01, r11, s00, s10, s01, s11,
                           rack0, rack1, sack0, sack1}
...
```

In fact, there are 17 deadlocked states altogether (where a deadlocked state may engage in t actions but is never capable of performing any observable actions). In the above case, and in fact in the other cases as well, the source of the deadlock comes from collisions on the medium. In the given deadlocked state, the medium wishes to deliver a message to the receiver (by executing 'r00), while the receiver wishes to send an acknowledgement (by executing 'sack1). As the medium makes no provisions for handling these collisions, deadlock results. Note that if the medium were to allow sends to "overwrite" messages awaiting delivery, then ABP would be correct; it is this property on which the correctness of the system, as it stands, relies.

Fixing the ABP can be done in one of two ways: either a new medium can be designed (this is not always feasible), or the sender and receiver can be altered so that they do not depend on the resolution of collisions by the medium. We follow the latter approach by by requiring the sender and receiver to "flush" the medium (by receiving and discarding any messages from the medium) before sending a message. The corrected agents appear in Figures 13 and 14. The new ABP system now contains 212 states, and it is observationally equivalent to SPEC.

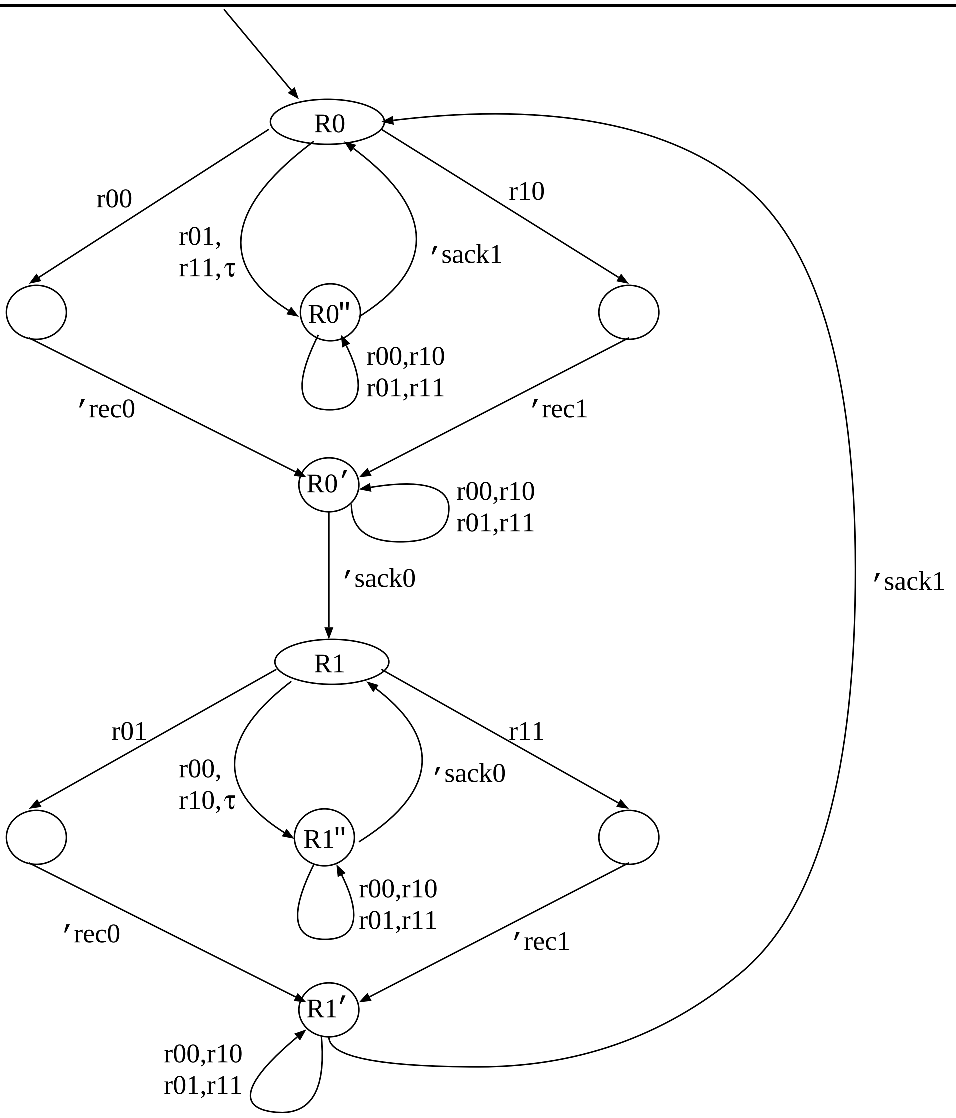
7.6 Performance

The Concurrency Workbench was originally conceived as a prototype tool designed to support the development and analysis of different verifications methods for distributed systems. Accordingly, a main emphasis during the implementation of the system was on flexibility, which we achieved by using a very high-level programming language (Standard ML) that permitted us to develop parameterized versions of the various analyses using higher-order functions. Of course this versatility has a performance penalty, as we indicate in this section; the Workbench is slower than other, more specialized tools for certain analyses. On the other hand, the modular structure of the Workbench makes it easy to replace existing modules with new ones, and as the interlanguage mechanisms




```
S_0 =  send0.S00 + send1.S10
S00 = 's00.(rack0.S.1 + rack1.S00 + t.S00) + rack0.S00 + rack1.S00
S10 = 's10.(rack0.S.1 + rack1.S00 + t.S00) + rack0.S10 + rack1.S10
S_1 = send0.S01 + send1.S11
S01 = 's01.(rack1.S.0 + rack0.S01 + t.S01) + rack0.S01 + rack1.S01
S11 = 's11.(rack1.S.0 + rack0.S11 + t.S11) + rack0.S11 + rack1.S11
```

Figure 13: Corrected version of the sender.




```
R0  = r00.'rec0.R0' + r10.'rec1.R0' + r01.R0" + r11.R0" + t.R0"
R0' = 'sack0.R1 + r00.R0' + r10.R0' + r01.R0' + r11.R0'
R0" = 'sack1.R0 + r00.R0" + r10.R0" + r01.R0" + r11.R0"
R1  = r01.'rec0.R1' + r11.'rec1.R1" + r00.R1" + r10.R1"+ t.R1"
R1' = 'sack1.R0 + r00.R1' + r10.R1' + r01.R1' + r11.R1'
R1" = 'sack0.R1 + r00.R1" + r10.R1" + r01.R1" + r11.R1"
```

Figure 14: Corrected version of the receiver.

n	States in $SCHED^n$	Transitions in $SCHED^n$	Observation graph transitions	Minimized states
4	117	283	2,009	64
5	285	831	7,346	160
6	669	2,287	25,949	384
7	1,533	9,307	89,534	896

Table 1: Size of $SCHED^n$ for some n before, during and after minimization.

for ML become more sophisticated, it will be possible to include very efficient routines written in lower-level languages in the system.

In order to provide a flavor of the performance of the automatic analyses in the Workbench, we have chosen the task of minimizing transition graphs with respect to observation equivalence. The agents are taken from Chapter 5 of [43], and are as follows:

$$\begin{array}{rcl} A & = & a.c.(b.d.A + d.b.A) \\ D & = & d.A \\ SCHED^n & = & (A[f_1] \mid D[f_2] \mid \cdots \mid D[f_n]) \backslash \{c_1, \ldots c_n\} \end{array}$$

where f_i denotes the relabeling $[a_i/a, b_i/b, c_i/c, \overline{c_{i-1}}/d]$. The idea is that $SCHED^n$ represents a scheduler for n customers: it will start each customer in turn (through the actions a_i), and each customer must signal termination (through b_i) before it can be started again. The scheduler is composed of n "cyclers" (A and D); each cycler maintains communication with one customer on ports a and b, and all cyclers are linked into a ring through ports c and d. For a further explanation of this example see [43].

We will consider $SCHED^n$ for $n = 4, \ldots, 7$. (For $n < 4$ the scheduler is fairly trivial, and for $n > 7$ the state graph is too large for the Workbench.) The size of these schedulers is indicated in Table 1 (this information, and the running times of the Workbench, is collected from [21]). The column "States in $SCHED^n$" gives the number of distinct states that the Workbench constructs in the transition graph. Note that agents such as D and $d.A$ are considered distinct here, even though they intuitively represent the same state. The column "Transitions in $SCHED^n$" give the number of transitions in the graph.

The Workbench computes the minimization by first transforming the corresponding graph to an observation graph, and subsequently applying the general minimization algorithm. In "Observation graph transitions" we list the number of transitions in the observation graph for $SCHED^n$ (the states remain the same). In "Minimized states" we give the number of states in the final minimized graph.

The execution time of the Concurrency Workbench on a Sun Sparcstation with 8Mb memory is 10, 70, 400, and 2000 CPU seconds for 4, 5, 6 and 7 customers respectively. Other, more specialized tools— e.g. AUTO system [48] or the system for deciding branching bisimulation reported in [27]— have been reported to be up to several hundred times faster, if one compares the figures given here with those in [27]. It should be noted that this comparison can only yield a gross estimate of the relative efficiencies of the tools, since the timing information was collected on different machines; moreover, in the case of the Workbench, the timing information takes into account the time to construct the finite-state machine from the CCS expression, while the others do not. The fact remains, however, that the other tools are significantly more efficient *at computing observation* *equivalence* than the present version of the Workbench. On the other hand, neither of the other tools includes a preorder or model checker, and as we indicated above, there are a number of ways to improve the speed of the Workbench while maintaining the modular and flexible structure of the system.

8 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.

8.1 State Space Analysis

The Workbench includes a variety of ways of analyzing the state space of an agent. In addition to the commands for finding deadlocked states in a system, there are various procedures for computing transitions and derivatives. These types of analyses are traditionally found in automatic verification tools and will not be discussed further in this paper.

8.2 Equation Solving

The equation-solving feature of the Workbench [45] 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, A, of an implementation 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 communications protocol, where the overall service, the medium, and the sender are known.

8.3 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 actions, although it is possible to encode the passing of values by associating a unique name to an action/value pair (cf. Section7.5). In the case of infinite value domains, however, this leads to syntactically infinite agents. In [36], 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 as an extension to the Workbench [41] and is exponential in the size of the agent—in fact, the problem has been shown to be NP-hard.

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 actions, 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 [35].

Another extension is the Lunsen system [23]. Lunsen is an imperative language with primitives for communication between parallel processes. A compiler translates Lunsen code into CCS in the format acceptable to the Workbench, making it possible to use the Workbench for automatic analysis of Lunsen programs. Our experience indicates that many applications, notably distributed algorithms such as mutual exclusion algorithms, are more naturally formulated in Lunsen than in CCS.

9 Conclusion

In this paper we have presented an overview of the Concurrency Workbench. We have shown that it is possible to provide a number 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. One benefit of this modularization is that the system is relatively easy to extend.

There are a number of directions for future work on the Workbench. Internally, the speed of a number of routines could be improved, in particular since the implementation language of the system (Standard ML) now supports more constructs for efficient programming (e.g. bit arrays) than was the case when the core of the system was implemented. In addition, more efficient algorithms for preorder checking [16] and model checking [17] have been discovered, and these should be implemented in the system. Another area of investigation would involve developing techniques for reducing the size of transition graphs that the system computes when verification routines are invoked. One promising approach is to use equational reasoning as a "rewriting mechanism" to minimize the number of new states created during the generation of a graph. Another involves the use of *compositional analysis*. 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 in this respect [14, 26, 54]. It is also conceivable that the model checker could be extended to check formulas compositionally using methods developed in [5, 52, 56].

There are also several ways in which the functionality of the Workbench could be extended and improved. As an example, other equivalences and preorders, including GSOS equivalence and the $\frac{2}{3}$-bisimulation preorder (also called *ready simulation*) [2, 40], 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* [30, 8]. 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 generating such formulas for partition-refinement-based bisimulation algorithms has been proposed in [8]; work is also underway on generating similar diagnostic information for the testing equivalences. A graphical interface is also under development; this tool will permit users to design systems graphically, with the tool then generating the appropriate CCS descriptions.

Finally, it would be interesting to develop machinery for verifying systems expressed in formalisms other than CCS. A joint project with INRIA-Sophia Antipolis is being undertaken to develop a front-end generator for the Workbench (and other automated tools) that would make it possible to parameterize the system with respect to the process algebra/programming language used to build agents. Work is also underway on automated techniques for reasoning about actions with priority, probabilistic processes and real-time systems (cf. [4, 10, 18, 24, 34, 49]).

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, Patrik Ernberg, Lars-åke Fredlund, 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] Bartlett, K.A., R.A. Scantlebury and P.T. Wilkinson. "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links." *Communications of the ACM* 12, n. 5, pp. 260–261, May 1969.

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

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

[4] Camilleri, J. and Winskel, G. "CCS with Priority Choice." In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pp. 246–255. Computer Society Press, Los Alamitos, 1991.

[5] Clarke, E.M. "Compositional Model Checking." In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems.* Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1989.

[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, pp. 244–263, 1986.

[7] Cleaveland, R. "Tableau-Based Model Checking in the Propositional Mu-Calculus." *Acta Informatica*, v. 27, 1990, pp. 725–747.

[8] Cleaveland, R. "On Automatically Distinguishing Inequivalent Processes." In *Computer-Aided Verification '90*, pp. 463–477. American Mathematical Society, Providence, 1991. Also Lecture Notes in Computer Science 531, Springer-Verlag, Berlin, 1991.

[9] 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*, pp. 11–23. Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1989.

[10] Cleaveland, R. and Hennessy, M.C.B. "Priorities in Process Algebra." *Information and Computation*, v. 87, n. 1/2, 1990, pp. 58–77.

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

[12] Cleaveland, R., Parrow, J. and Steffen, B. "A Semantics-Based Tool for the Verification of Finite-State Systems." In *Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing and Verification*, June 1989, pp. 287–302. North-Holland, Amsterdam, 1990.

[13] Cleaveland, R., Parrow, J. and Steffen, B. "The Concurrency Workbench." In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, pp. 24–37. Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1989.

[14] Cleaveland, R. and B. Steffen. "When is 'Partial' Complete? A Logic-Based Proof Technique using Partial Specifications." In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pp. 440–449. Computer Society Press, Los Alamitos, 1990.

[15] Cleaveland, R. and B. Steffen. "A Preorder for Partial Process Specifications." In *Proceedings of CONCUR '90*, pp. 141–151. Lecture Notes in Computer Science 458, Springer-Verlag, Berlin, 1990.

[16] Cleaveland, R. and B. Steffen. "Computing Behavioural Relations, Logically." In *Proceedings International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science 510, Springer-Verlag, Berlin, 1991.

[17] Cleaveland, R. and B. Steffen. "A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus." In *Proceedings of Computer-Aided Verification '91.* To appear in Lecture Notes in Computer Science.

[18] Cleaveland, R. and Zwarico, A. "A Theory of Testing for Real Time". In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science.* Computer Society Press, Los Alamitos, 1991.

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

[20] 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*, pp. 267–278. Computer Society Press, Washington, 1986.

[21] Ernberg, P. and Fredlund, L.-å. "Identifying some Bottlenecks of the Concurrency Workbench". Technical Report T90002, Swedish Institute of Computer Science, 1990.

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

[23] Fredlund, L.-å, B. Jonsson and J. Parrow. "An Implementation of a Transitional Semantics for an Imperative Language." In *Proceedings of CONCUR '90*, pp. 246–262. Lecture Notes in Computer Science 458, Springer-Verlag, Berlin, 1990.

[24] Glabbeek R. van and Smolka S. A. and Steffen B. and Tofts C. M. N. "Reactive, Generative, and Stratified Models of Probabilistic Processes." In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pp. 130–141. Computer Society Press, Los Alamitos, 1990.

[25] Goyer, J.H. *Communications Protocols for the B-HIVE Multicomputer.* Master's Thesis, North Carolina State University, 1991.

[26] Graf, S. and Steffen, B. "Using Interface Specifications for Compositional Reduction." In *Computer-Aided Verification '90.* American Mathematical Society, Providence, 1991. Also Lecture Notes in Computer Science 531.

[27] Groote, J. F. and Vaandrager, F. "An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence." In *Proceedings, 17th International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science 443, Springer-Verlag, Berlin, 1990.

[28] Har'el, Z. and R.P. Kurshan. "Software for Analytical Development of Communications Protocols." *AT&T Technical Journal*, v. 69, n. 1, pp. 45–59. February, 1990.

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

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

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

[32] Holzmann, G. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, 1991.

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

[34] Jensen, C.-T. "The Concurrency Workbench with Priorities." In *Proceedings of Computer-Aided Verification '91.* To appear in Lecture Notes in Computer Science.

[35] Jonsson, B., Kahn, A., and Parrow, J. "Implementing a Model Checking Algorithm by Adapting Existing Automated Tools." In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, pp. 179–188. Lecture Notes in Computer Science 407, Springer-Verlag, Berlin, 1989.

[36] 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*, pp. 421–433. Lecture Notes in Computer Science 349, Springer-Verlag, Berlin, 1989. To appear in *Information and Computation.*

[37] Kanellakis, P. and Smolka, S.A. "CCS Expressions, Finite State Processes, and Three Problems of Equivalence." *Information and Computation*, v. 86, n. 1, pp. 43–68, May 1990.

[38] Larsen, K.G. "Proof Systems for Hennessy-Milner Logic with Recursion." In *Proceedings of CAAP*, Lecture Notes in Computer Science 299, Springer-Verlag, Berlin, 1988.

[39] Larsen, K.G. and Thomsen, B. "Compositional Proofs by Partial Specification of Processes." In *Proceedings of the Third Annual Symposium on Logic in Computer Science.* Computer Society Press, Washington, 1988.

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

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

[42] 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.

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

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

[45] Parrow, J. "Submodule Construction as Equation Solving in CCS." *Theoretical Computer Science*, v. 68, pp. 175–202, 1989.

[46] Parrow, J. "Verifying a CSMA/CD-Protocol with CCS." In *Proceeding of the Eighth IFIP Symposium on Protocol Specification, Testing, and Verification*, pp. 373–387. North-Holland, Amsterdam, 1988.

[47] 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.* North-Holland, Amsterdam, 1987.

[48] Roy, V. and de Simone, R. "Auto/Autograph." In *Computer-Aided Verification '90*, pp. 477–491. American Mathematical Society, Providence, 1991.

[49] Smolka S.A. and Steffen B. "Priority as Extremal Probability." In *Proceedings of CONCUR'90.* Lecture Notes in Computer Science 458, Springer-Verlag, Berlin, 1990.

[50] Steffen, B. "Characteristic Formulae." In *Proceedings International Colloquium on Automata, Languages and Programming*, pp. 723–733. Lecture Notes in Computer Science 372, Springer-Verlag, Berlin, 1989.

[51] Steffen, B.U., and Ingólfsdóttir, A. "Characteristic Formulae for CCS with Divergence." To appear in *Theoretical Computer Science.*

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

[53] Stirling, C. and Walker, D.J. " Local Model Checking in the Modal Mu-Calculus", In *Proceedings of TAPSOFT.* Lecture Notes in Computer Science 352, Springer-Verlag, Berlin, 1989.

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

[55] Walker, D.J. "Analysing Mutual Exclusion Algorithms Using CCS." *Formal Aspects of Computing*, v.1, pp. 273–292, 1989.

[56] Winskel, G. "On the Compositional Checking of Validity." In Proceedings CONCUR'90, Lecture Notes in Computer Science 458, pp. 481–501, 1990.