A Semantics-Based Verification Tool for Finite-State Systems*

Rance Cleaveland † Joachim Parrow ‡
Bernhard Steffen §

Abstract

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

1 Introduction

This paper describes the Concurrency Workbench [7], a prototype tool for the automatic verification of finite-state processes. Automated verification tools are practically motivated: the development of increasingly

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

†Computer Science Department, MAPS, University of Sussex, Falmer, Brighton, BN1 9QH, ENGLAND

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

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

complex distributed computer systems requires increasingly complex verification techniques to guarantee correctness, and the increase in detail rapidly becomes unmanageable without computer assistance.

The analysis of concurrent systems relies heavily on the underlying formal semantics of processes, about which there is as yet no consensus. One goal in the design of the Workbench is to incorporate several different verification methods, as well as process semantics, in a modular fashion. This means that each method may be applied to different semantic models, yielding a spectrum of techniques for reasoning about processes. The versatility of the Workbench has many advantages: it facilitates a comparison between many aspects of formal verification, it supports mixed verification strategies which use more than one method, and it makes the system easily extensible. This versatility contrasts with existing automated tools, which typically embody a particular semantics and a particular form of verification. Examples of such systems include Aldébaran [10], AUTO [2], CESAR [25], EMC [4], and Winston [20]. For a survey of systems for the verification of communication protocols see [1].

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

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

The Workbench has been successfully applied to verification of communication protocols, notably the alternating bit protocol and the CSMA/CD protocol [24], and mutual exclusion algorithms [30]; it has also been used to debug the Edinburgh Computer Science Department's electronic mailing system. Furthermore, it is a valuable instrument in education, in industries as well as in universities, and is being investigated as a tool for analyzing communications protocols by Swedish Telecom and by Hewlett-Packard.

The remainder of the paper is organized as follows. In the next section we describe the conceptual structure of the workbench, and give an overview of the different verification methods. Section 3 presents the syntax and basic semantics of processes, and Section 4 some of the transformations on transition graphs. Sections 5 through 7 illustrate the basic verification methods, Section 8 mentions some additional features of the Workbench, and Section 9 contains our conclusions and directions for future work.

2 The Architecture of the Workbench

2.1 The Interface Layer

An overview of the Workbench is presented in Figure 1. The user interacts with a *command interpreter* in the interface layer, 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. A *parser* transforms the concrete syntax of such parameters into an abstract syntax in the form of parse trees, which can be stored in a small *data base*. There is also a package defining the *basic semantics* of processes. Following [21], each process yields a labeled transition graph which describes its behavior in terms of visible communication events and internal, or τ, events. A description of the syntax and semantics of processes can be found in Section 3.

2.2 The Semantics Layer

The semantics layer consists of algorithms for transforming the transition graphs generated by the interface layer. For example, the *observation* transformation extends the transitions by absorbing internal τ-events into visible events; thus the resulting graphs do not record the timing of internal computations in the corresponding processes. The *deterministic* transformation makes the graphs deterministic in the usual sense of the word: the resulting graphs do not record nondeterministic choices. The *acceptance* transformation yields deterministic graphs augmented with deadlock potentials in the form of acceptance sets. These transformations, and some minor variations, are described in Section 4.

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

2.3 The Analysis Layer

The Workbench features three main verification methods for proving that processes meet their specifications, and the analysis layer contains the corresponding basic analysis algorithms. In the first method, specifications are themselves processes that describe precisely the high-level behavior required of an implementation. The corresponding algorithm determines whether two processes are *equivalent* in the sense of having the same behavior. This algorithm can also be used to *minimize* a process, i.e. compute an equivalent process with a minimal number of states. A definition of equivalence and a brief account of the algorithm can be found in Section 5.

The second method also uses processes as specifications, but these specifications are treated as minimal requirements to be met by implementations. In this approach specifications can be annotated with "holes" (or "don't care" points); an implementation satisfies one of these *partial* specifications if it supplies at least the behavior demanded by the specification while filling in some holes. The method relies on an ordering relation, or *preorder*, between processes: a process A is "more defined than" a process B if A has the same behavior as B except for the holes in B. The preorder algorithm determines if a process is more defined than its specification in this sense. A definition of the preorder and an account of the algorithm can be found in Section 6.

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. As further described in Section 7, the *model checking* algorithm determines whether a process satisfies an assertion.

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

3 Processes in the Workbench

This section describes the syntax of the Calculus of Communicating Systems (CCS), which is used to define processes, or *agents*, in the Workbench. We also show how agents are interpreted as transition graphs. In the rest of this paper 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 internal action τ. The external actions, also called *communication events*, are divided into input events and output events. We use $a, b, \ldots$ to range over actions. If a is an input event then $\overline{a}$ is the *complementary* output event; intuitively these represent input and output on the port a. As in pure CCS we consider only communication events without value parameters. Agents are defined using the following operators from [21]:

Nil	Termination
$\perp$	Divergence (or *bottom*, "hole", or "don't care")
$a.$	Prefixing by action a; unary prefix operator
$+$	Choice; binary infix operator
$\mid$	Parallel composition; binary infix operator
$\backslash L$	Restriction on (finite) set of actions L; unary postfix operator
$[f]$	Relabeling by f, which maps communication events to communication events; unary postfix operator

We use $p, q, \ldots$ to range over agents. Relabeling functions are written as a sequence of renamings; for example $p[a_1/b_1, a_2/b_2]$ is the agent p with b_1, b_2, $\overline{b_1}$ and $\overline{b_2}$ renamed to a_1, a_2, $\overline{a_1}$ and $\overline{a_2}$, respectively.

Finally, we assume a set of *agent identifiers* (called *Constants* in [21]). An identifier A may be *bound* to an agent p, written $A \stackrel{\text{def}}{=} p$, where p may itself contain A. This enables recursively defined agents. For example, the following bindings constitute a definition of a buffer of capacity n.

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

Here BUF_n^i represents a buffer of capcity n currently holding i items. The following agent, CBUF_n, is a composite implementation of BUF_n using n buffers of capacity 1 (here $n \geq 2$).

$$
\begin{array}{l}
\mathrm{CBUF}_n \stackrel{\text{def}}{=} \\
\quad (\,\mathrm{BUF}_1[a_1/out] \mid \underbrace{\cdots \mid \mathrm{BUF}_1[a_i/in, a_{i+1}/out] \mid \ldots}_{i=1,\cdots,n-2} \mid \mathrm{BUF}_1[a_{n-1}/in]\,) \setminus \{a_1, \ldots, a_{n-1}\}
\end{array}
$$

Finally the partially specified buffer PBUF_n acts as BUF_n as long as the environment never attempts to store more than n items; otherwise its behavior is undefined.

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

3.2 Transition Graphs

Agents are given an operational semantics defined in terms of *transition relations*, $\xrightarrow{a}$, where a is an action. Intuitively, $p \xrightarrow{a} p'$ holds when p can evolve into p' by performing action a; in this case, p' is said to be an a-*derivative* of p. The transition relation is defined inductively on the basis of the operators. Thus $a.p \xrightarrow{a} p$, and $p + q \xrightarrow{a} p'$ if either $p \xrightarrow{a} p'$ or $q \xrightarrow{a} p'$. The agent $p|q$ behaves like the interleaving of p and q with the possibility of complementary actions synchronizing, yielding a τ-action. $p \setminus L$ behaves like p with the exception that no actions in L are allowed, while $p[f]$ behaves like p with actions renamed by f. A formal account of this semantics is found in [21]. We also define a *divergence* predicate on agents: agents with an unguarded occurrence of $\perp$ (i.e. with a $\perp$ not within the scope of an action prefixing operator) are considered divergent.

The Workbench adopts *transition graphs* as models of agents. The transition graph of an agent p contains a set of nodes, where each node corresponds to an agent. There is one distinguished node, called the root node, which corresponds to p itself. Edges are labeled by actions; there is an edge labeled by a with source node n and target node n' if $p \xrightarrow{a} p'$ holds of the corresponding agents. A node is *divergent* if the corresponding agent is divergent. Each node additionally carries a polymorphic *information* field, the contents of which vary according to the semantic transformations or analyses being performed on the graph.

Figure 2 contains examples of transition graphs. Divergent nodes are represented by unfilled circles, and other (convergent) nodes by filled circles. The root node is indicated by an unlabeled arrow.

4 Transition Graph Transformations

As indicated previously, various transformations on transition graphs are used in conjunction with basic analysis algorithms, yielding a variety of verification methods. We briefly describe some of the transformations here.

4.1 Observation Graphs

The transition graphs as described in Section 3.2 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 an algorithm for computing *observation* graphs. These are based on the notion of *experiment relations*, which are defined as follows.

$$n \stackrel{\varepsilon}{\Longrightarrow} n' \quad \text{if} \quad n(\xrightarrow{\tau})^* n'$$

$$n \stackrel{a}{\Longrightarrow} n' \quad \text{if} \quad n \stackrel{\varepsilon}{\Longrightarrow} \stackrel{a}{\longrightarrow} \stackrel{\varepsilon}{\Longrightarrow} n'$$

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

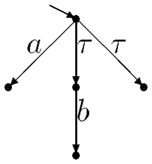
The observation graph transformation takes a transition graph and modifies the edges to represent the $\stackrel{a}{\Longrightarrow}$ and $\stackrel{\varepsilon}{\Longrightarrow}$ relations instead of the $\stackrel{a}{\longrightarrow}$ and $\stackrel{\tau}{\longrightarrow}$ relations. It uses well-known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

The transformation takes time that is polynomial in the size of the graph. Surprisingly, in most cases the subroutine for transitive closure accounts for more than 80 per cent of the execution time when determining observation equivalence.

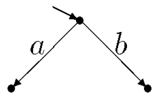
A slight variation of the observation transformation, called the *congruence* transformation, yields observation graphs that additionally record the possibility of initial τ-actions. Such graphs are suitable for determining observation congruence [21] and weak precongruence [29].

4.2 Deterministic Graphs

The strong and observation equivalences and preorders distinguish agents on the basis of the exact points during their executions where nondeterministic choices are made. Accordingly, the graphs mentioned in sections 3.2 and 4.1 faithfully record each time a agent makes such a choice. However, other equivalences and preorders, like the *may* preorders and equivalences [11] and ordinary trace (language) equivalence, do not require such detailed accounts of the choice structure of agents. In order to cater for these equivalences the Workbench includes an algorithm for transforming graphs into *deterministic* graphs (also called *Dgraphs*), which are graphs having no τ-derivatives and at most one a-derivative for any action a. As a simple example, the transition graph



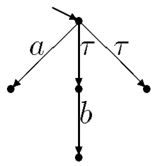
is transformed to



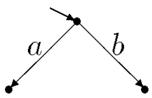
As another example, the deterministic graph of CBUF$_2$ is precisely the transition graph of BUF$_2$. The algorithm for computing deterministic graphs is well-known from automata theory (see e.g. [14]). In general, this transformation has an exponential complexity, owing to the fact that it is theoretically 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.

4.3 Acceptance Graphs

The deterministic graphs do not contain information pertinent to the deadlock properties of agents. Some verification methods, notably methods based on the *testing* ([11]) and *failures* ([13]) equivalences and preorders, require this information, but still do not require the detailed information concerning choice structure present in observation graphs. The appropriate transition graphs for these equivalences are *acceptance* graphs (also called *Tgraphs*); these are deterministic graphs whose nodes additionally contain deadlock and divergence information encoded as *acceptance sets*. The acceptance set $n.acc$ of a node n is a set of sets of actions: each member of $n.acc$ is a set of actions, representing an obligation to be able to continue from n with any action in the set. For example,



has the acceptance graph



where only the root node has a nontrivial acceptance set:

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

Each element in the acceptance set corresponds to a stable state (a state from which no internal action is immediately possible) in the original graph;

such an element contains precisely the actions that are immediately possible from the stable state. In the example, the two elements of the acceptance set correspond to the possibilities of the original graph to evolve unobservably along the right branch (where no further action is possible) or the middle branch (where only b is possible). Note that $\{a\}$ is not a member of the acceptance set since the original graph cannot evolve into a state where only a is possible. The acceptance sets in the other nodes contain only the empty set. As another example, Figure 4 shows the acceptance graph resulting from the transformation of CBUF_2.

Acceptance sets also encode information pertinent to divergence. A node in the original graph which is reachable from a *globally divergent* node (a node which can either evolve through internal actions to a divergent node, or is on an infinite path of internal actions) is characterized by an empty acceptance set of corresponding acceptance graph nodes.

A closely related kind of graph, the *must* graph (also called *STgraph*), is appropriate for the must equivalence and preorder ([11]). Must graphs are like acceptance graphs, except that a node reachable from a globally divergent node has no outgoing edges. The algorithm for generating acceptance and must graphs is described in [6].

5 Equivalence Checking

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

5.1 Definition of the Equivalence

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

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

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

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

5.2 Derived Equivalences

Many interesting equivalences turn out to be instances of $\mathcal{C}$-equivalence combined with graph transformations. Let U denote the universal relation, i.e. $U = N \times N$. Then a U-bisimulation is a bisimulation in the sense of Milner [21] and U-equivalence is strong equivalence in CCS. Also, observation equivalence corresponds to U-equivalence on observation graphs, observation congruence to U-equivalence on congruence graphs, and trace (or may) equivalence to U-equivalence on deterministic graphs. For must and testing (failures) equivalence define $\langle m, n \rangle \in \mathcal{A}$ to hold exactly when $m.acc$ and $n.acc$ are compatible, i.e. each element of $m.acc$ is a superset of an element of $n.acc$, and vice versa. Then two graphs are must equivalent if their associated must graphs are $\mathcal{A}$-equivalent, and they are testing (failures) equivalent if their associated acceptance graphs are $\mathcal{A}$-equivalent [6].

As an example, for any particular n it is the case that BUF_n and CBUF_n (see Section 3) are both observation and testing equivalent.

5.3 The Algorithm

Our algorithm is adapted from one presented in [17]. It works by attempting to find a $\mathcal{C}$-bisimulation relating the root nodes of the graphs. To do so, it maintains a *partitioning* of the nodes in G_1 and G_2, the graphs under consideration. A partitioning is a set of *blocks*, where each block is a set of nodes such that each node is contained in exactly one block. Such a partitioning naturally induces an equivalence relation on the nodes of the graphs: two nodes are related precisely when they are in the same block.

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

The time and space complexities of this algorithm are $O(k * \ell)$ and $O(k + \ell)$ respectively, where k is the number of states, and ℓ is the number of transitions, in the two graphs. A marginally more efficient algorithm appears in [22]; however, there is not yet enough evidence to suggest that this algorithm is appreciably faster in practice. In any event, this complexity is not a limiting factor; tests with the Workbench have shown that the time consumed by this algorithm is only a small fraction of the total time spent when computing observation equivalence. Most of the time is consumed in the graph transformations.

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

6 Preorder Checking

The second basic analysis within the Workbench involves checking a preorder between two agents. This is done in a way similar to equivalence checking; after converting the agents to transition graphs we then apply a general preorder algorithm. The algorithm is based upon the following generalization of the notion of equivalence introduced in Section 5.1.

6.1 Definition of the Preorder

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

Definition 6.1 A *$\mathcal{C}$-prebisimulation* between G_1 and G_2 is a relation $\mathcal{R} \subseteq N \times N$ such that $\langle m, n\rangle \in \mathcal{R}$ implies that:

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

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

6.2 Derived Preorders

Many interesting preorders turn out to be instances of the $\mathcal{C}$-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 [27,29]. Now many interesting preorders can be constructed as instances of the $\mathcal{C}$-preorder:

- The bisimulation divergence preorder [27,29] results by setting:

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

  This defines the strong version of the divergence preorder. The weak version, $\precsim$, 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 [6].

  - 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.

As an example, we can establish the following.

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

Here PBUF_n is used as a partial specification that is satisfied by all BUF_m for $m > n$. Partial specifications like this can be used to express that two agents, although not equivalent, may be used interchangeably in certain contexts. 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. The desired service of the protocol is SERVICE; a verification that the protocol is correct is to determine

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

(where $\approx$ is observation equivalence). Further assume that the protocol strictly uses the medium as a one-place buffer: when this buffer contains an item no attempt will ever be made to place another item in the buffer. We then expect (1) to hold also when MEDIUM is replaced by BUF_m for any $m > 1$, since all these buffers behave identically in this context. One way to prove this is to first prove that

$$\text{SERVICE} \quad \lessapprox \quad (\text{SENDER}|\text{RECEIVER}|\text{PBUF}_1)\backslash L \tag{2}$$

¿From $\text{PBUF}_1 \lessapprox \text{BUF}_m$ for all $m > 1$ we get

$$(\text{SENDER}|\text{RECEIVER}|\text{PBUF}_1)\backslash L \quad \lessapprox \quad (\text{SENDER}|\text{RECEIVER}|\text{BUF}_m)\backslash L \tag{3}$$

and from (2), (3), and the transitivity of $\lessapprox$ we get

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

If we further assume that SERVICE contains no divergent states, it follows from [29] that

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

The same technique is equally applicable to other equivalences. The methodology of introducing "don't care" states in order to get partial specifications could also be useful within a stepwise refinement strategy.

6.3 The Algorithm

The algorithm for computing the $\mathcal{C}$-preorder works by attempting to find a $\mathcal{C}$-prebisimulation relating the roots of the graphs. In contrast to Section 5.3, however, preorders cannot be represented by partitions. We obtain an appropriate representation by annotating every node n with a set of nodes considered to be "greater" than n.

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

The time and space complexities of this algorithm are $O(k^4 * \ell)$ and $O(k^2 + \ell)$, respectively, where k is the number of states, and ℓ is the number of transitions, in the two graphs. The loss of efficiency compared with the equivalence algorithm is due to the fact that we cannot use the same compact representation of relations as in Section 5.3.

7 Model Checking

The third basic analysis within the Workbench is based on model checking [3,4,5,9,28], in which specifications are written in an expressive modal logic, the *propositional (modal) mu-calculus*. The Workbench can check whether an agent meets such a specification.

7.1 Definition of the Logic

The formulas, or propositions, of the logic are described by the following grammar:

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

$$| \quad \nu X.\Phi \mid \mu X.\Phi$$

In the above, X ranges over propositional identifiers; these can be bound to formulas in the same way as agent identifiers can be bound to agents. Also, a ranges over actions, B over names of user-defined macros, and *arg-list* stands for a list of actions and/or formulas that B requires in order to produce a proposition. In $\nu X.\Phi$ and $\mu X.\Phi$ all free occurrences of X must be positive.

These formulas are interpreted with respect to nodes in transition graphs. The formulas *tt* and *ff* hold of every node and no node, respectively. A propositional identifier is interpreted with respect to an environment binding identifiers to propositions: a node n satisfies X if it satisfies the formula to which X is bound in the environment. The connectives $\neg, \vee, \wedge$, and $\Rightarrow$ have their usual meanings.

The modal operators $\langle a \rangle$, $[a]$, $\langle . \rangle$ and $[.]$ make statements about the edges from 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 a-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 arguments, which may be either actions or formulas, and returns a proposition. The formula *Barg-list* is then interpreted as the proposition returned by B in response to *arg-list*.

Formulas of the type $\nu X.\Phi$ and $\mu X.\Phi$ can be understood as "Φ, where X is bound to Φ". Since X may occur in Φ they represent *recursive formulas*. Formally they correspond to 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 a node satisfies $\nu X.\Phi$ if it satisfies all the Φ_i. Also let $\hat{\Phi}_0$ be the proposition *ff*, and let $\hat{\Phi}_{i+1}$ be defined as $\Phi[\hat{\Phi}_i/X]$. Then a node satisfies $\mu X.\Phi$ if it satisfies one of the $\hat{\Phi}_i$. As a trivial example, $\nu X.X$ holds of any node while $\mu X.X$ never holds.

7.2 Derived Logics

Applying the model checking algorithm to graphs generated by the different graph transformations yields different interpretations for the modal operators. For instance, model checking with observation graphs causes the modal operators to be insensitive to τ-actions, but also makes it difficult to formulate liveness properties since $n \stackrel{\varepsilon}{\longrightarrow} n$ holds for every node n in an observation graph.

The recursive formulas add a tremendous amount of expressive power to the logic (see e.g. [9,26]), since they allow the description of safety and liveness properties. However, such formulas are in general 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 define these operators as macros. For example, it is possible to define the operators of the branching time temporal logic CTL [4]. Examples include the following:

$$\begin{aligned} \mathit{Henceforth}(\Phi) &= \nu X.(\Phi \wedge [.]X) \\ \mathit{Eventually}(\Phi) &= \mu X.(\Phi \vee (\langle . \rangle tt \wedge [.]X)) \end{aligned}$$

Thus $\mathit{Henceforth}(\Phi)$ holds of n if Φ holds of every node reachable from n, while $\mathit{Eventually}(\Phi)$ holds of n if along every path leading from n there is a node satisfying Φ. In CTL, these operators are written $AG\Phi$ and $AF\Phi$ respectively.

As an example, consider the graph corresponding to $\textsc{cbuf}_2$ in figure 2. It is possible to show that it is deadlock-free as follows. Define the macro *Deadlock* by

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

This proposition is true of states that cannot perform any actions. For any particular n the model checker can establish that $\textsc{cbuf}_n$ satisfies the formula

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

where $\mathit{Henceforth}()$ is the macro defined above; therefore $\textsc{cbuf}_n$ never deadlocks.

7.3 The Algorithm

Our model checking algorithm comes from [5]; we shall not describe it here. The complexity of the algorithm is in general exponential in the length of the formula being checked, although for large classes of formulas the algorithm is well-behaved. A precise characterization of this complexity is still under investigation. Other algorithms include the tableau-based approach in [28], and a semantics-based algorithm in [9]. An automated proof system for a subset of the logic is described in [18].

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 commands for computing transitions and derivatives, there are features for determining which states are deadlocked and for computing sequences of experiments that lead to deadlocked states. These types of analyses are traditionally found in automatic verification tools, and will not be discussed in this paper.

8.2 Equation Solving

The equation solving feature of the Workbench [23] is used to solve equations of type $(A|X)\backslash L = B$ where A, B and L are given. The method is useful within a top-down or stepwise refinement strategy: if a specification (B) and parts of an implementation (A) are known, solving such an equation amounts to constructing a specification of the missing submodules. The method works by successively transforming equations into simpler equations, in parallel with the generation of a solution. These transformations can be performed automatically by the system according to certain heuristics, or the user can apply them interactively. The tool has been used for the generation of a receiver in a communication protocol, where the overall service, medium, and sender are known.

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 events, although it is possible to encode the passing of values by associating a unique event name to an event/value pair. In the case of infinite value domains, however, this leads to syntactically infinite agents. In [16], an alternative encoding is proposed, in which the infinitely many data values are represented schematically. Using the resulting transitional semantics, bisimulation equivalences can be defined in such a way as to correspond exactly to the bisimulation equivalences in full CCS. This result entails a decision procedure for *data-independent* agents, i.e. agents which communicate data values but do not perform any computations or tests on the values. The decision procedure has been implemented in the Workbench [19].

An interface has also been built between the Workbench and the Extended Model Checker (EMC), which is a tool for checking the satisfiability of temporal logic (CTL) formulas. CTL differs from the logic in Section 7 in that the view of processes is different: in CTL there are no communication events, and states can be distinguished 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 [15].

9 Conclusion

In this paper we have presented an overview of the Concurrency Workbench. We have shown that it is possible to supply a variety of algorithms for deducing the correctness of processes based on a variety of different process semantics. This has been achieved through a strict separation between the semantic models of processes and the basic algorithms used to analyze them. The modularization makes the Workbench relatively easy to extend, although it slightly impairs efficiency.

There are a variety of directions for future work on the Workbench. One involves extending the kinds of value-passing that are supported. Another involves the computation of *distinguishing formulas* [12]. At present, when processes are found not to be equivalent, no indication is given as to why. One way to convey this information is to give a modal formula satisfied by one process but not the other. Work is also underway on a graphical interface.

Another possible area of investigation involves *compositional reasoning* [3]. The parallel composition of two processes usually entails a combinatorial explosion in the size of the state space of the resulting process. 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 [29]; one possibility is to implement this in a way similar to the equation-solving tool in Section 8.2. It is also conceivable to extend the model checker to check formulas compositionally, using techniques developed by Stirling [27].

We close with some facts about the development of the Workbench. The implementation (in Standard ML under Unix) was begun by Parrow in 1986; the early versions featured mainly an observation equivalence checker and the equation solver. Since 1987 the Workbench has been developed jointly by the three of us, and we arrived at the current modular structure in 1988. The implementation now (1989) comprises approximately 10,000 lines of code, and runs on Vax, Sun, and Macintosh computers. The project has consumed more than 5 man-years of effort.

Acknowledgements

We are grateful to Matthew Hennessy, Robin Milner, and Colin Stirling for initiating and managing the Workbench project. Also, many thanks to Lennart Beckman, Jo Blishen, Michael Mendler, Kevin Mitchell, Fredrik Orava, Björn Pehrson, and David Walker for many helpful suggestions concerning the implementation of the Workbench and/or the presentation in this report.

References

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

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

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

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

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

[6] Cleaveland, R. and Hennessy, M.C.B "Testing Equivalence as a Bisimulation Equivalence." To be presented at the Workshop on Automatic Verification Methods for Finite State Systems, Grenoble 1989.

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

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

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

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

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

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

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

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

[15] Jonsson, B., Kahn, A., and Parrow, J. "Implementing a Model Checking Algorithm by Adapting Existing Automated Tools." SICS Research Report, 1989. In preparation; to be presented at the Workshop on Automatic Verification Methods for Finite State Systems, Grenoble 1989.

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

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

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

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

[20] 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.

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

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

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

[24] Parrow, J. "Verifying a CSMA/CD-Protocol with CCS." In *Proceeding of the Eigth IFIP Symposium on Protocol Specification, Testing, and Verification*, 1988, North-Holland, pp 373-384.

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

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

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

[28] Stirling, C. and Walker, D.J. " Local Model Checking in the Modal Mu-Calculus", In *Proceedings TAPSOFT* 1989, Lecture Notes in Computer Science 351, pp 369-383, Springer Verlag 1989.

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

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

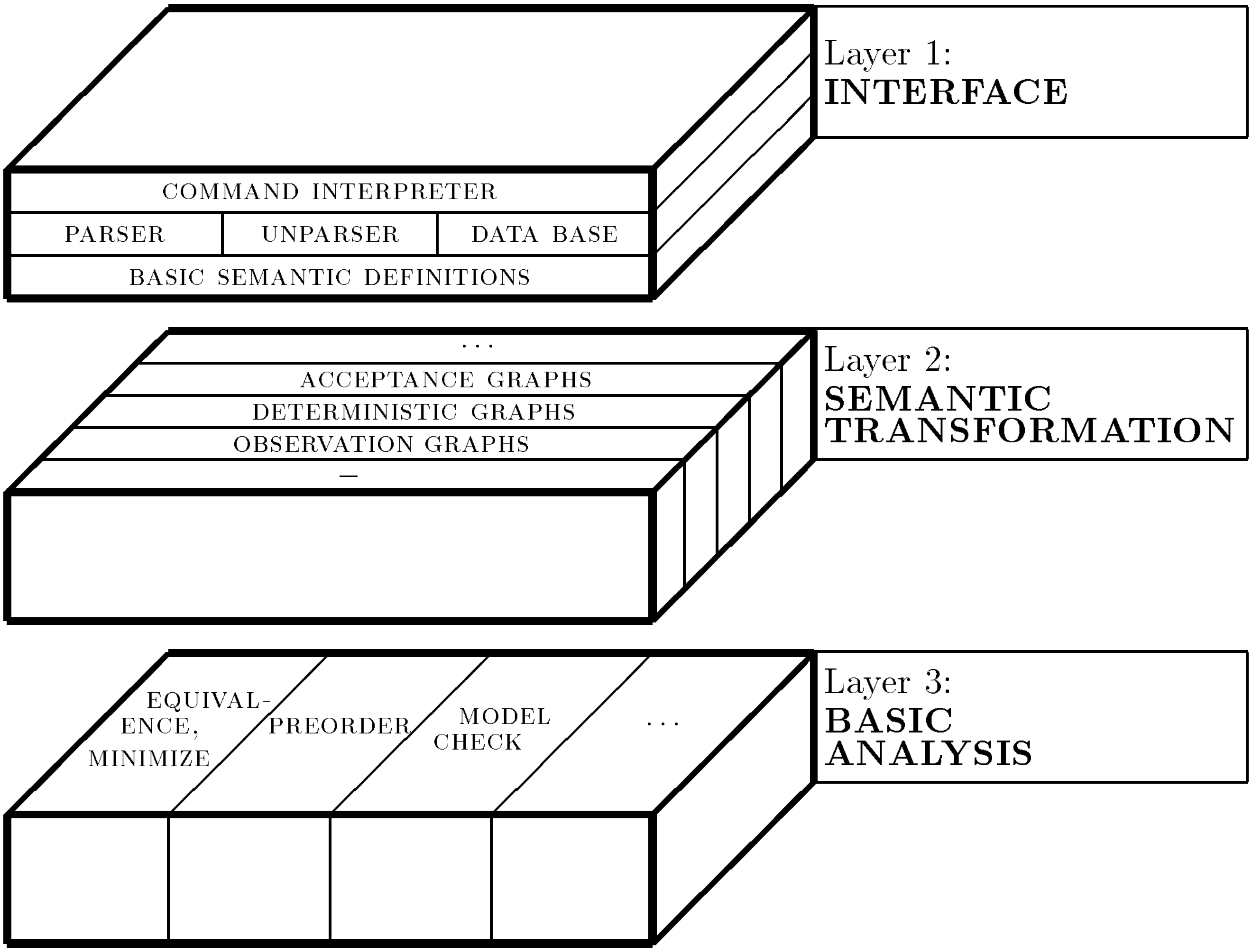
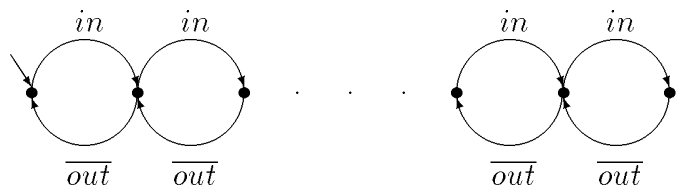
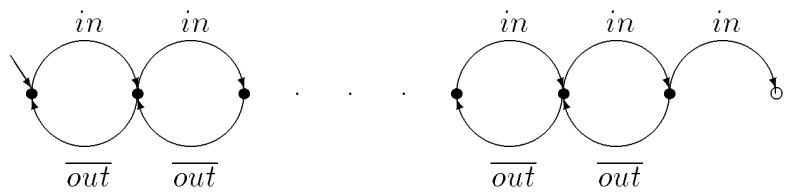


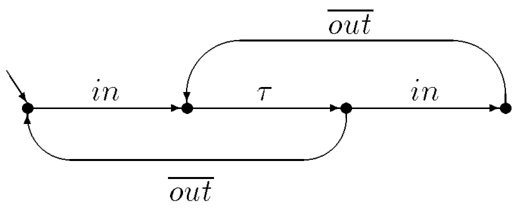
Figure 1: Overview of the Workbench



a) The transition graph for BUF_n, the buffer of capacity n. The graph has $n+1$ nodes.



b) The transition graph for PBUF_n, which has one additional divergent node.



c) The transition graph for CBUF_2, the composite buffer of capacity 2.

Figure 2: Examples of transition graphs.

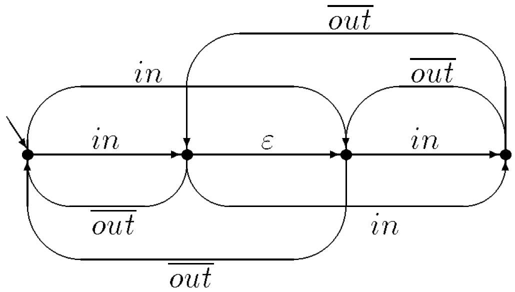
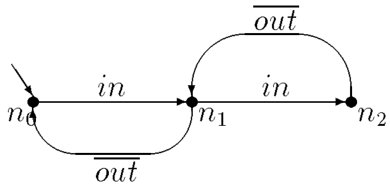


Figure 3: The observation graph corresponding to CBUF$_2$.



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

Figure 4: The acceptance graph for CBUF$_2$.