The Bakery Protocol: A Comparative Case-Study in Formal Verification

David Griffioen[*,†] Henri Korver[‡,§]
CWI

Abstract

Groote and the second author verified (a version of) the Bakery Protocol in μCRL. Their process-algebraic verification is rather complex compared to the protocol. Now the question is: How do other verification techniques perform on this protocol? In this paper, we present a new correctness proof by using I/O-automata theory and discuss the relative merits of both approaches.

1 Introduction

In this paper, we verify a particular version of the Bakery Protocol[1] for an arbitrary large capacity max by means of I/O automata theory. The parameter max ranges over (positive) integers and denotes the number of standing places in the bakery shop. The correctness proof is developed and checked with the aid of the Larch Prover [GH93] which is a theorem prover based on first-order logic. Our verification method is *semi-automatic* in the sense that the intelligent proof steps are provided by the user. This is to be contrasted with *fully-automatic* (finite-state) tools like CWB [CPS93], Auto [SV89], Aldébaran [FKM93], SPIN [Hol91], COSPAN [KL93], etc, where the protocol can only be treated for a fixed capacity, e.g. for the instance $\mathsf{max} = 10$.

The protocol derives its name from the well-known situation in a busy bakery shop where customers pick a number at the ticket machine in order to guarantee that they are served in a proper order. It is based on the old FIFO principal (first in, first out); the customer who's ticket matches with the clock of the baker is served first. In fact, correctness is formulated by stating that, after abstraction from the ticket machine and the clock, the Bakery Protocol has the same external behaviour as a bounded FIFO queue.

[*]Department of Software Technology, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, e-mail: griffioe@cwi.nl.

[†]Research supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125.

[‡]Department of Philosophy, P.O. Box 80126, 3508 TC Utrecht, The Netherlands, e-mail: korver@phil.ruu.nl.

[§]These investigations were supported by the Netherlands Computer Science Research Foundation (SION) with funds from the Netherlands Organisation for Scientific Research (NWO).

[1]This name is used earlier by Lamport [Lam74] for another protocol.

Recently this well-known protocol has been used as a bench-mark for testing the suitability of μCRL as a verification method for distributed systems in [GK95]. This example is considered as an interesting case-study because the protocol is concurrent and crucially depends on (infinite) data parameters on which induction is applied. In this aspect the protocol is more advanced than for instance the Alternating Bit Protocol which can be proven correct without induction.

The protocol consists of four actions: *ENTER* (entering the bakery shop), *TICKET* (picking a number from the ticket machine), *COUNTER* (walking to the counter-desk) and *OUT* (leaving the bakery). The correctness criterion is that, after abstraction of the *TICKET* and *COUNTER* actions, the Bakery Protocol has the same external behaviour as a FIFO queue. For the first time a rigorous proof of this correctness criterion has been given in [GK95] by using μCRL [GP95, GP94] which is an extended version of ACP [BW90] with abstract data types. This approach is called *process algebraic* as it is primarily based on equational reasoning. The μCRL proof is compact and precise, but rather advanced and technical for such relatively simple protocol.

In this paper, we will verify the Bakery Protocol by using *I/O automata theory* [LT89]. In this approach, correctness is proven by establishing a simulation relation between two automata (transition graphs). More concretely, here we will establish a bisimulation relation between the automata representations of the Bakery Protocol and the FIFO queue.

One of the advantages of the approach followed here compared with [GK95] is that the verification is rather intuitive and does not require advanced prescience. Another point is that our verification is easy to formalise and check by the Larch Prover [GH93]. To our knowledge, this is the first time that the Bakery Protocol has been proof-checked. We expect that computer checking the μCRL proof would require significantly more effort. The verification model used in this paper has also several disadvantages and we shall discuss the pros and cons of both the μCRL approach and the one followed here.

As a final point we mention that we came up with two different proofs for the Bakery Protocol in the I/O-automata model. One of the two appears to be far more elegant and significantly shorter (about half the size) than the other. There are no well-known heuristics indicating which proof strategies lead to the best proofs. The lesson we learned is that it is very important to choose the most appropriate simulation (bisimulation) relation before starting the whole verification. Otherwise, the verification may become unnecessary involved. In the sequel, some heuristics are given drawn from our experiences.

The paper is organised as follows. In the next section, we give a formal specification of the protocol (Bak). Then, in Section 3, we define its intended behaviour (P). Correctness is proved by establishing a bisimulation relation between Bak and P in Section 4. In Section 5 we provide an alternative correctness proof and discuss the differences with the one given in Section 4. The checking of our proofs using the Larch Prover is discussed in Section 6. In Section 7, we draw the conclusions of our work; in particular we compare our verification with the one given in [GK95]. The model in which our verification takes place is given in Appendix A. Finally, in Appendix B the Larch formalization of the protocol is listed.

2 Specification of the protocol

Data types play an important role in the specification of the Bakery Protocol. Therefore, we start with a description of the various data types that are used. The data types listed below can be found in [GH93]. They can also be found in the library which is included in the distribution packet of the Larch tools.

2.1 Data

We assume a typed signature Σ and a Σ-algebra $\mathcal{A}$ which consists of the following components:

- a type **Bool** of booleans with constant symbols true and false, and a standard repertoire of function symbols ($\wedge$, $\vee$, $\neg$, $\rightarrow$), all with the standard interpretation over the booleans. Also, we require, for all types **S** in Σ, an equality, inequality, and if-then-else function symbol, with the usual interpretation:

$$\begin{aligned} . = . &: \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ . \neq . &: \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ \mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ . &: \mathbf{Bool}\times\mathbf{S}\times\mathbf{S} \rightarrow \mathbf{S} \end{aligned}$$

  Note the (harmless) overloading of the constants and function symbols of type **Bool** with the propositional connectives used in formulas. We will frequently view boolean valued expressions as formulas, i.e., we use b as an abbreviation of $b = \mathsf{true}$.

- a type **Int** of integers, with a standard repertoire of function symbols $+$, $-$, $\leq$, ... We also need the constant max which denotes the maximal number of standing places in the Bakery Protocol. We assume that $\mathsf{max} > 0$. In fact the natural numbers would suffice here but we still use **Int** because this data type is included in the Larch Library [GH93]. In the proof it is sometimes convenient to interpret booleans as integers. Therefore we will use the conversion function $\mathsf{i} : \mathbf{Bool} \rightarrow \mathbf{Int}$ with $\mathsf{i}(\mathsf{true}) = 1$ and $\mathsf{i}(\mathsf{false}) = 0$. For readability we omit the symbol i in expressions.

- a type **Data** of customers who enter the bakery shop.

- a type **Pair** in order to attach a number to a customer, with a function symbol $[.,.] : \mathbf{Data}\times\mathbf{Int} \rightarrow \mathbf{Pair}$ for constructing pairs of customers and integers in the usual way. We write $p.\mathsf{datum}$ for the first component and $p.\mathsf{index}$ for the second component of a pair p.

- a type **Bag** of finite multisets over the domain of **Pair**, with a constant symbol $\varnothing$, denoting the empty multiset, a function symbol $\{.\} : \mathbf{Pair} \rightarrow \mathbf{Bag}$ for the operation which assigns to a pair the corresponding singleton multiset and a function symbol $\cup : \mathbf{Bag}\times\mathbf{Bag} \rightarrow \mathbf{Bag}$ for the union of multisets. Beside these constructors we have the function symbols $\mathsf{delete} : \mathbf{Pair}\times\mathbf{Bag} \rightarrow \mathbf{Bag}$, $\in$: $\mathbf{Pair}\times\mathbf{Bag} \rightarrow \mathbf{Bool}$, and $\mathsf{size} : \mathbf{Bag} \rightarrow \mathbf{Int}$. delete deletes an element from a multiset. Note that in multisets just one copy of an element is removed in case there are duplicates. Furthermore, $\in$ tests whether or not an element occurs one or more times in a multiset, and size returns the number of elements (including duplicates) that are in a multiset.

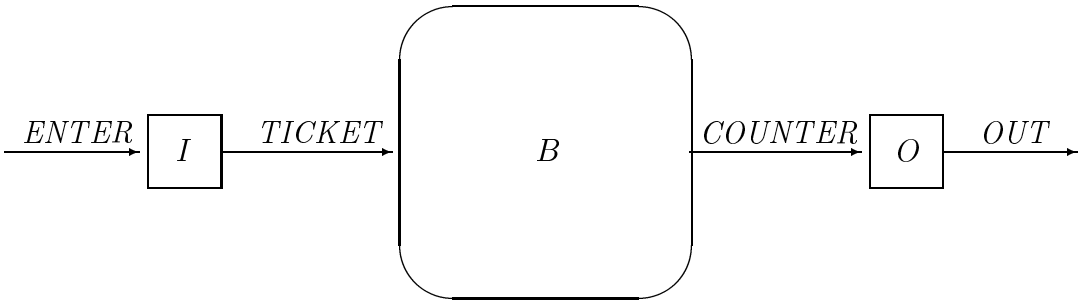

Figure 1: The Bakery Protocol.

- a type **Queue** of finite queues over the domain of **Data**, with a constant symbol ϵ, denoting the empty queue, and a function symbol append : **Data**×**Queue** → **Queue**, denoting the operation of prefixing a queue with a data element. Besides these constructors, there are function symbols head : **Queue** → **Data**, tail : **Queue** → **Queue**, and len : **Queue** → **Int**. The function head takes the element that was appended to the queue at first, tail returns the remainder of a queue after removal of the head element and len returns the length of the queue.

2.2 Protocol specification

We specify the Bakery Protocol (see Figure 1) in terms of three components:

The ticket machine Automaton I (in-sequencer) given in Figure 2 models the behaviour of the ticket machine. A customer, represented by the variable d, entering the bakery shop is modelled by the action $ENTER(d)$. The state variable *full* indicates whether (or not) the ticket machine is already occupied. If this is the case, the variable *datum* represents the customer who is standing at the ticket machine. By the action $TICKET([d, i])$ customer d actually picks a number i and the counter is incremented by one. Initially, the value of the counter is set to zero and nobody is standing at the ticket machine.

External	*ENTER*, *TICKET*
State Variables	$datum$: **Data**, $full$: **Bool**, $count$: **Int**
Initialization	$full = \mathsf{false} \wedge count = 0$

$ENTER(d : \mathbf{Data})$	$TICKET(p : \mathbf{Pair})$
Precondition	**Precondition**
$full = \mathsf{false}$	$p = [datum, count] \wedge full = \mathsf{true}$
Effect	**Effect**
$datum := d$	$full := \mathsf{false}$
$full := \mathsf{true}$	$count := count + 1$

Figure 2: Automaton I.

The shop The standing places of the bakery are modelled by automaton B (bag) given in Figure 3. The customers in the shop are modelled by the variable *bag* which denotes a multiset, i.e. a set that may contain duplicates. In contrast with a **Queue**, the data type **Bag** does not impose an ordering on the elements. This is exactly the essence of the protocol: customers are not tight to a fixed position but are free to stand wherever they want. A customer with a ticket, represented by the pair p, walking into the shop is modelled by the action $TICKET(p)$. By the action $COUNTER(p)$ he arrives at the counter-desk. Note that the function delete only removes one element from the bag in case there are duplicates.

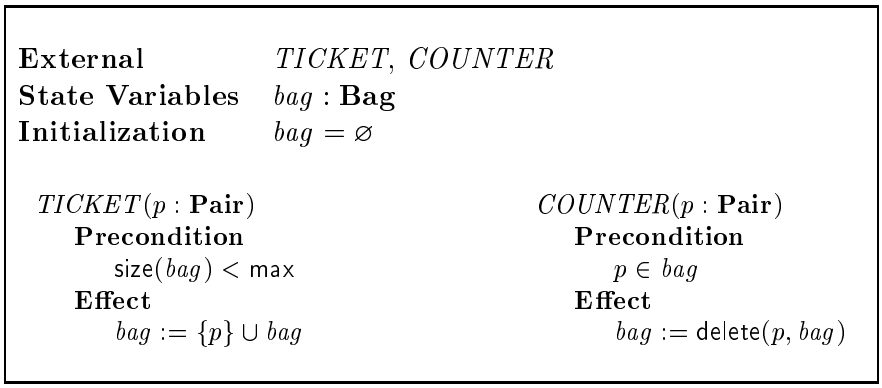

Figure 3: Automaton B.

The counter desk Automaton O (out-sequencer) given in Figure 4 models the counter desk which is equipped with a clock.

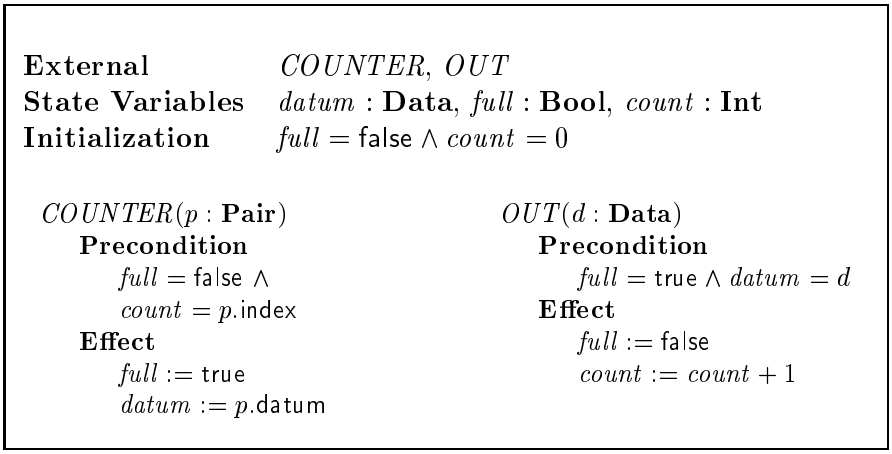

Figure 4: Automaton O.

When the boolean variable *full* is true than the variable *datum* gives the customer that is served. Otherwise, there is nobody at the counter-desk. The action $COUNTER(p)$ expresses that the customer given by p.datum is selected to be served next. This is the case when his number p.index is equal to the value of the clock, which is represented by

variable *count*. After being served, he leaves the shop by the action $OUT(p.\mathsf{datum})$. In the mean time, the counter of the clock is incremented by one. Analogous to the ticket machine, the clock starts counting from zero.

The full protocol *Bak* is defined as the parallel composition of automata I, B, and O, with communication between these components hidden:

$$Bak \triangleq \mathsf{HIDE}\ H\ \mathsf{IN}\ (I \| B \| O)$$

where $H \triangleq \{ TICKET(p), COUNTER(p) \mid p \text{ in domain } \mathbf{Pair} \}$. The fact that a customer, after picking a number at the ticket machine, directly enters the shop (represented by the variable *bag*) is modelled by synchronizing on the *TICKET* actions of automata I and B. And synchronization on the *COUNTER* actions of automata B and O expresses that a customer who's number matches the value of the clock immediately reaches the counter-desk. Declaring these actions to be internal can be interpreted as standing outside the bakery such that one can only observe customers entering and leaving the shop.

3 Correctness criterion

The Bakery Protocol is supposed to work as a bounded queue with a maximal length of $\mathsf{max}+2$; there can be max customers waiting for their turn, but there can also be a customer busy obtaining a number at the ticket machine and another customer can already have been selected to be served at the counter.

The behaviour of a queue of capacity $\mathsf{max}+2$ is specified by automaton P in Figure 5. In the next section, we prove that *Bak* and P are bisimilar, i.e. have the same behaviour. Two automata A and B are bisimilar if, starting from the root, they can mimic their external actions in every following state as follows. When A can do an external action a, B can also perform an action a, possibly preceded and followed by internal actions, and vice versa. A formal definition can be found in Appendix A.

External	*ENTER*, *OUT*
State Variables	*queue* : **Queue**
Initialization	$queue = \epsilon$

ENTER(d : **Data**)
Precondition
$\mathsf{len}(queue) < \mathsf{max}+2$
Effect
$queue := \mathsf{append}(d, queue)$

OUT(d : **Data**)
Precondition
$d = \mathsf{head}(queue) \wedge queue \neq \epsilon$
Effect
$queue := \mathsf{tail}(queue)$

Figure 5: Automaton P.

4 The correctness proof

In order to establish a bisimulation between Bak and P we must first gain insight into what are the reachable states of Bak. A well-known technique is to find a suitable number of *invariants* of the protocol, i.e. properties that are valid for all reachable states. It turned out that two simple invariants, which are given below, are sufficient.

Both invariants are proved in LP by induction on the length of the executions to the reachable states. Definitions of the notions *execution* and *reachable* can be found in Appendix A. As an illustration we have included the full proof of the invariant $INV2$ (Lemma 2). In order to distinguish between the state variables of different components of Bak, we prefix each state variable by the name of the component it originates from. The following invariant states that the capacity of the bakery shop is never exceeded.

Lemma 1. *For all reachable states of Bak the following property $INV1$ holds:*

$$\mathsf{size}(B.bag) \leq \mathsf{max}$$

The invariant $INV2$ below says that the value of the ticket machine is equal to the value of the clock plus the number of customers that are in the shop (including the person that is served at the counter-desk).

Lemma 2. *For all reachable states of Bak the following property $INV2$ holds:*

$$I.count = \mathsf{size}(B.bag) + O.count + O.full$$

Proof. Let s' be a reachable state of Bak. By induction on the length n of the shortest execution of Bak that ends in s', we prove $s' \models INV2$. If $n = 0$ then s' is a start state. Hence $s' \models I.count = 0 \wedge \mathsf{size}(B.bag) = 0 \wedge O.count = 0 \wedge \neg O.full$, which implies $s' \models INV2$.

For the induction step, suppose that s' is reachable via an execution with length $n+1$. Then there exists a state s that is reachable via an execution of length n and $s \xrightarrow{a} s'$, for some action a. By the induction hypothesis we have $s \models INV2$. We prove $s' \models INV2$ by case distinction on a.

- Assume $a = ENTER(d)$. Then $s' \models INV2$ trivially follows from $s \models INV2$ and the observation that a does not change any of the state variables mentioned in $INV2$.

- Assume $a = TICKET(p)$

  1) $s \models I.count = \mathsf{size}(B.bag) + O.count + O.full$ (by i.h.)
  2) $s \models \mathsf{succ}(I.count) = \mathsf{size}(B.bag \cup p) + O.count + O.full$ (by 1)
  3) $s' \models I.count = B.bag + O.count + O.full$ (by 2 and effect)
  4) $s' \models INV2$ (by 3)

- Assume $a = COUNTER(p)$

  1) $s \models I.count = \mathsf{size}(B.bag) + O.count$ (by pre. and i.h.)
  2) $s \models I.count = \mathsf{size}(\mathsf{delete}(p, B.bag)) + O.count + 1$ (by 1 and $p \in bag$)
  3) $s' \models I.count = \mathsf{size}(B.bag) + O.count + O.full$ (by 2 and effect)
  4) $s' \models INV2$ (by 3)

- Assume $a = OUT(d)$

  1) $s \models I.count = \mathsf{size}(B.bag) + O.count + 1$ (by pre. and i.h.)
  2) $s \models I.count = \mathsf{size}(B.bag) + \mathsf{succ}(O.count)$ (by 1)
  3) $s' \models I.count = \mathsf{size}(B.bag) + O.count + O.full$ (by 2 and effect)
  4) $s' \models INV2$ (by 3)

□

These two simple invariants are needed for establishing a bisimulation between *Bak* and P. Before embarking on the main theorem, we introduce the auxiliary function symbol $\mathsf{number} : \mathbf{Queue}\times\mathbf{Int} \to \mathbf{Bag}$. Given a queue q and a number i, number attaches consecutive numbers to elements of q counting from i downwards and puts them in a bag. This function is completely characterised by the following two axioms:

$$\mathsf{number}(\epsilon, i) = \varnothing$$
$$\mathsf{number}(\mathsf{append}(d, q), i) = \{[d, i]\} \cup \mathsf{number}(q, i-1)$$

Theorem 1. *The relation $BISIM$ defined by the following formula is a* (weak) bisimulation *between Bak and P:*

$$\begin{array}{l}(\mathsf{if}\ I.full\ \mathsf{then}\ \{[I.datum, I.count]\}\ \ \mathsf{else}\ \varnothing)\ \cup\ B.bag\ \cup \\ (\mathsf{if}\ O.full\ \mathsf{then}\ \{[O.datum, O.count]\}\ \mathsf{else}\ \varnothing) \\ \qquad\qquad = \\ \mathsf{number}(queue, I.count - 1 + I.full)\end{array}$$

This theorem is proved by using the two invariants given above. At the end of this section (Lemma 3) we present a part of the hand-written proof as an illustration. There one can see why for instance invariant $INV2$ is actually needed. In Section 6 we report on the formalisation of the proof in the Larch Prover.

The intuition behind the relation $BISIM$ given above is simple. It directly expresses that the customers in the bakery shop act as if they were placed in a FIFO queue on the basis of their number.

Corollary 1. $Bak \leftrightarrows P$.

Proof. Immediate by Theorem 1 and the definition of bisimulation $\leftrightarrows$. □

Corollary 1 says that *Bak* and P are *bisimilar*, i.e. 'have the same behaviour', see Appendix A. In other words, every external action – possibly preceded or followed by internal actions – performed by *Bak* can be mimicked by P and vice versa.

Next we will give a fragment of the proof that $BISIM$ is indeed a bisimulation. Here the manual proof is presented and in Section 6 the LP version is presented. The proof fragment is the lemma stating that if two states are bisimilar and both systems do an OUT action the new states are bisimilar again.

Lemma 3. *For all reachable states s in Bak and u in P we have that:*

$$(s,u) \in BISIM \wedge s \overset{OUT(d)}{\longrightarrow} s' \wedge u \overset{OUT(d)}{\longrightarrow} u' \quad \Rightarrow \quad (s',u') \in BISIM.$$

Proof. Assume that the left-hand side of the implication holds. Then we prove that $(s',u') \in BISIM$ as follows.

1. $s.O.full$ (by precondition of OUT)
2. $(\mathsf{if}\ s.I.full\ \mathsf{then}\ [I.datum, I.count]\ \mathsf{else}\ \varnothing) \cup s.B.bag \cup [s.O.datum, s.O.count]$
   $= \mathsf{number}(u.queue, s.I.count - 1 + s.I.full)$ (by $BISIM$ and 1)
3. $s.I.count = \mathsf{size}(s.B.bag) + s.O.count + 1$ (by $INV2$ and 1)
4. Assume $\neg s.I.full$
   4.1 $s.B.bag \cup [s.O.datum, s.O.count] = \mathsf{number}(u.queue, s.I.count - 1)$
   4.2 $s.I.count - 1 = \mathsf{size}(s.B.bag) + s.O.count$ (by 3)
   4.3 $\mathsf{number}(\mathsf{tail}(u.queue), s.I.count - 1) = s.B.bag$ (by 4.1, 4.2 and $DnumU$)
   4.4 $\mathsf{number}(u'.queue, s'.I.count - 1) = s'.B.bag$ (by effect and 4.3)
   4.5 $\neg s'.I.full \wedge \neg s'.O.full$ (by effect)
   4.6 $(s',u') \in BISIM$ (by 4.4 and 4.5)
5. Assume $s.I.full$. Analogous to the $\neg s.I.full$ case.
6. Q.E.D. (by 4 and 5)

In Step 4.3 above, data identity $DnumU$ is used:
$\mathsf{number}(q, i + \mathsf{size}(b)) = b \cup \{[d,i]\} \quad \Rightarrow \quad \mathsf{head}(q) = d \wedge \mathsf{number}(\mathsf{tail}(q), i + \mathsf{size}(b)) = b.$
Of course this identity is also checked in LP. □

5 Proving the same using a more general bisimulation

In I/O-automata proofs finding the right (bi)simulation and invariants is of course very important. The length and readability of different approaches can variate greatly. In this section we report on an "old" bisimulation relation $BISIM'$ that needed a proof more than twice as long as $BISIM$. We tried to find out what caused the longer proof, to prevent us making such mistake an other time. After presenting the old relation $BISIM'$ we will explain what is "wrong" with it. The relation $BISIM'$ is determined by the following formula:

$$\begin{array}{l}\mathsf{corr}(\ \ (\mathsf{if}\ I.full\ \mathsf{then}\ \{[I.datum, I.count]\}\ \ \mathsf{else}\ \varnothing)\ \cup\ B.bag\ \cup \\ \qquad\quad (\mathsf{if}\ O.full\ \mathsf{then}\ \{[O.datum, O.count]\}\ \mathsf{else}\ \varnothing) \\ \qquad , \ queue) = \mathsf{true}\end{array}$$

where the equation

$$\begin{array}{l}\mathsf{corr}(b,q) = \mathsf{if}\ \mathsf{isEmpty}(q)\ \ \mathsf{then}\ \mathsf{isEmpty}(b) \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{if}\ [\mathsf{toe}(q), \mathsf{max\_i}(b)] \in b \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{then}\ \mathsf{corr}(\mathsf{delete}([\mathsf{toe}(q), \mathsf{max\_i}(b)], b), \mathsf{untoe}(q)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{false}\end{array}$$

defines the function $\mathsf{corr} : \mathbf{Bag} \times \mathbf{Queue} \to \mathbf{Bool}$.

In the equation above, $\mathsf{toe}(q)$ denotes the last element that is appended to the queue q, and $\mathsf{max\_i}(b)$ denotes the largest number that is attached to the customers in the bag b. Furthermore, $\mathsf{isEmpty}(q)$ and $\mathsf{isEmpty}(b)$ stand for $q = \epsilon$ and $b = \varnothing$, respectively. The function corr determines a crucial correspondence between being the customer with the highest number in the bag b and being the customer who stands at the back of the queue q. In both situations you are served last.

This $BISIM'$ is larger than $BISIM$ ($BISIM' \supset BISIM$) which means that the proof has to be constructed by means of a weaker hypothesis. For instance $BISIM'$ does not require that the numbers in the bag are successive. And this is a crucial property of the protocol: the out-sequencer takes the customers out in successive order, if a number is missing the Bakery Protocol deadlocks.

Example 1. Let s be a state of Bak satisfying $s.I.count = 3$, $s.I.full = s.O.full = \mathsf{false}$ and $s.B.bag = \{[\mathsf{d}_2, 2], [\mathsf{d}_1, 0]\}$. And let u be a state of P satisfying

$$u.queue = \mathsf{append}(\mathsf{d}_2, \mathsf{append}(\mathsf{d}_1, \epsilon)).$$

Then $(s, u) \notin BISIM$ and $(s, u) \in BISIM'$. The states s and u are not bisimilar because P can do $OUT(\mathsf{d}_1)$, $OUT(\mathsf{d}_2)$, and Bak can not do any $OUT(d)$ action after $OUT(\mathsf{d}_1)$. But $BISIM'$ is still a valid bisimulation relation because s is not reachable.

We used Lemma 4 to prove that states like s are unreachable, it expresses that the ticket numbers are distributed in a successive order, starting from $I.count - 1$ counting downwards.

Lemma 4. *For all reachable states of Bak the following property $INV3$ holds:*

$$I.count - \mathsf{size}(B.bag) \leq i < I.count \iff \exists d.\ [d, i] \in B.bag$$

The proof of this invariant is far more involved than the proofs of the two invariants needed in the final proof. And note that this extra invariant is needed just because $BISIM'$ relates more non-bisimilar states than $BISIM$ does.

An other difference is that the new definition does not use the toe, untoe and $\mathsf{max\_i}$ functions. Also the corr function is not used but that is no real gain because we use the number function instead. We think that in practice reducing the number of functions over data can reduce the number of data lemmas.

The last advantage of the new $BISIM$ is, that in our opinion, the number function is more intuitive than the corr function. The number function recursively builds a bag of pairs by assigning indexes to the data elements in the queue. The corr function does something similar the other way around: it destructs the bag and the queue as long as the corresponding data elements are the same. We think the recursion in the corr function over the bag and the queue is more complex than the recursion in the number function, and that this extra complexity contributes to the length of the proof.

We think that this example indicates that in certain cases the choice of a (bi)simulation can influence the length of the proof considerable. Though we think that it is impossible to give a cook book to construct the (bi)simulation with the shortest proof, the above mentioned differences probably deserve some attention when comparing (bi)simulations.

6 Checking the proof in LP

In this section we will report on the use of the Larch Prover (LP) [GG, GH93] by which the proofs of the invariants and the bisimulation have been checked. LP is a proof checker or as the authors put it a proof debugger, it does not use complicated heuristics to search for a proof. It supports first-order logic and is based on rewriting. When LP is asked to prove a conjecture, it typically normalizes the conjecture using the rewrite-rule versions of the axioms and the lemmas that have already been proved. When a normal-form is reached the proof is suspended and the user can invoke a command. We will mention a few typical options: the user can start a proof by cases, making LP to generate a subgoal for each case. A proof by induction is possible when a sort has a set of generators. This set of generators must be given by the specifier. LP will generate a subgoal for each generator. An other possibility is to apply a rewrite rule in the reversed direction, this is allowed because the rewrite-rules are oriented axioms, not implications. When quantifiers are involved variables or constants can be fixed, specialized or generalized. Furthermore LP can compute critical-pairs and complete a set of rewrite-rules. Besides these proof-commands LP has commands to direct the orienting of axioms into rewrite-rules, to make rewrite-rules inactive, to make proof scripts, etc. Because a proof in LP is based on rewriting, the tool is good at it: it is fast and rewriting modulo associativity and commutativity is supported.

Before we can prove anything we must formalize the problem (the bakery protocol and it's correctness). In this case-study we used the Larch Shared Language (LSL) [GH93] for this purpose. The specifications in LSL can be translated by the `lsl` tool to LP input scripts. So from our point of view LSL is a front-end language for LP.

LSL is an algebraic specification language. It supports modules, called `traits`, that can include other traits. In a trait sort- and function-symbols are introduced, the signature of the functions is declared and properties of the functions can be defined by axioms expressed in first-order logic.

For the formalization of the I/O-automata notions we used the formalization of [LSGL94] as a starting point and adapted these for the model we use, see Appendix B for a listing of the traits involved. Because no timing is involved, we do not need the `Bounds` and `TimedAutomaton` traits of [LSGL94], in the trait `Automaton` we changed some names and the type of the effect function. Effect was a predicate

```
effect :  States[A],Actions[A],States[A] -> Bool
```

The effect predicate holds when a transition exists from the first argument state to the third argument state by the second argument action. Because the bakery protocol is deterministic, it is possible to use an effect function

```
eff:  States[A],Actions[A] -> States[A]
```

that returns the target state given the current state and the action. This formalization is less general, because non-determinism is not expressible, but it has the advantage that LP computes the new state.

In Figure 6 the trait defining the full protocol *Bak* is depicted, see Section 3 for the definition of *Bak*. The line numbers are added for reference only.

Note that the composition operator ($\|$) and the (HIDE IN) operator are not formalized in LP. We think it is hard to do so because these are higher-order notions and LP only supports

```
[ 1] Bak : trait
[ 2]   includes Automaton(Bak), Bag(Pair,B), Integer(Int)
[ 3]   Pair tuple of datum: D, index: Int
[ 4]   States[Bak] tuple of I:Sq, B:Bg, O:Sq
[ 5]   Sq tuple of full : Bool, datum : D, count : Int
[ 6]   Bg tuple of bag : B
[ 7]
[ 8]  introduces
[ 9]     TICKET, COUNTER: Pair -> Actions[Bak]
[10]     max : -> Int
[11]   asserts
[12]     Actions[Bak] generated by ENTER, TICKET, COUNTER, OUT
[13]     \forall s, s': States[Bak], d:D, i:Int, p:Pair
[14]
[15] ~isExternal(TICKET(p)) /\ ~isExternal(COUNTER(p));
[16]
[17] max > 0 ;
[18]
[19] init(s) == s.I.full = false /\ s.I.count = 0 /\ s.B.bag = {} /\
[20]            s.O.full = false /\ s.O.count = 0;
[21]
[22] pre(s,ENTER(d)) == ~s.I.full;
[23] eff(s,ENTER(d)) == [[true,d,s.I.count],s.B,s.O];
[24]
[25] pre(s,TICKET(p)) ==     p.datum       = s.I.datum
[26]                       /\ p.index       = s.I.count
[27]                       /\ s.I.full      = true
[28]                       /\ size(s.B.bag) < max;
[29] eff(s,TICKET(p)) == [[false,s.I.datum,succ(s.I.count)],[{p} \U s.B.bag],s.O];
[30]
[31] pre(s,COUNTER(p)) ==    s.O.full  =  false
[32]                       /\ s.O.count =  p.index
[33]                       /\ p       \in s.B.bag  ;
[34] eff(s,COUNTER(p)) == [s.I,[delete(p,s.B.bag)],[true,p.datum,s.O.count]];
[35]
[36] pre(s,OUT(d)) == d = s.O.datum /\ s.O.full ;
[37] eff(s,OUT(d)) == [s.I,s.B,[false,s.O.datum,succ(s.O.count)]]
```

Figure 6: Larch version of *Bak*.

first-order logic.[2]

Next we will give some elucidation to Figure 6.

line 2. The trait `Automaton` is included. Traits can have parameters, in this way the `Automaton` trait can be re-used (twice): once for the definition of trait `Bak` and once for the trait `P`. Furthermore the `Bag` and `Integer` traits are included, these are taken from the Larch Library (see [GH93]).

line 3-6. The sorts `Pair` and `States[Bak]` are tuples. A tuple is comparable to a record in Pascal or C.

line 9-10. The constant `max` and actions `TICKET`, `COUNTER` are declared.

line 12. The `generated by` clause expresses that every action is a `ENTER`, `TICKET`, `COUNTER` or `OUT` action.

line 13. The `\forall` construct declares the variables that are used in the axioms.

line 14-end. Here the definitions for the functions are given: The `init` function to denote the start states and the `pre` and `eff` functions to specify the transition system representation of the Bakery Protocol.

In Figure 7 the LP proof of $INV2$ (see Lemma 2) is depicted. The proof contains two steps, (1) it is proved that it holds in the initial states, (2) it is proved that the transitions preserve the invariant.

At line 4, LP is asked to prove that INV2 holds in the initial states by assuming the left-hand side of the implication. LP implements this by introducing a new constant, say `sc`, and adding `init(sc) = true` to the set of facts, the new goal is the right-hand side `inv2(sc)`. After normalization this seemed equal to `true`, so the implication holds. The diamonds (`<>`) and the boxes (`[]`) are generated by LP and denote that a (sub)-goal is introduced or proved, respectively.

The next `prove` command asks LP to prove that the transitions preserve the invariant. First the proof method is set to normalization and the `=>`-method. This setting causes LP to try normalization first and if the conjecture is in normal form and the top-function is an implication (`=>`) then the left-hand side is assumed. The first proof-step is a case distinction on the actions of Bak, coded in LP as a induction proof with only four base-cases and no induction step. Everything to the left of `%` is treated as a comment and ignored by LP. For the `COUNTER` step LP needs a little help. Lemma `SIZE`:

```
p \in b  =>  size(delete(p,b))  =  (size(b)  -  1)
```

is instantiated. With this help LP finishes the proof of `inv2`.

When using invariants in a proof of an other lemma we use the following:
`reachable(s) => inv2(s)`
This is exactly: for all reachable states `inv2` holds, and it follows directly from the definition

[2] In contrast to the process-algebraic notion of parallel composition also notated as ($\|$) the I/O-automata notion of composition is very easy to compute, it can be syntacticly defined for specifications in the precondition/effect style.

```
declare operator inv2 : States[Bak] -> Bool
assert inv2(s) = (s.I.count = size(s.B.bag) + s.O.count + (if s.O.full then 1 else 0))

prove :inv2: (init(s:States[Bak]) => inv2(s)) by =>
    <> => subgoal
    [] => subgoal
  [] conjecture

prove :inv2: (reachable(s:States[Bak]) /\ inv2(s) /\ isStep(s:States[Bak],a,s') => inv2(s'))
  set proof-method normalization, =>-method
  res by ind on a:Actions[Bak]
    <> basis subgoal  % ENTER
      <> => subgoal
      [] => subgoal
    [] basis subgoal
    <> basis subgoal  % TICKET
      <> => subgoal
      [] => subgoal
    [] basis subgoal
    <> basis subgoal  % COUNTER
      <> => subgoal
      ins b by sc.B.bag, p by pc in SIZE
      [] => subgoal
    [] basis subgoal
    <> basis subgoal  % OUT
      <> => subgoal
      [] => subgoal
    [] basis subgoal
  [] conjecture
qed
```

Figure 7: LP proof of Lemma 2

of reachable and the induction proof. But we did not prove this within LP but added the following implication for every invariant instead.

```
(init(s) => inv(s)) /\
(reachable(s) /\ inv(s) /\ isStep(s,a,s') => inv(s'))
=>   (reachable(s) => inv(s))
```

In Figure 8 the LP version of the proof of Lemma 3 of $BISIM$ is depicted. In this proof the same commands as in the proof above are used. The `resume by case` command instructs LP to make a case distinction on `~sc.I.full` ($\neg sc.I.full$) and `sc.I.full`.

7 Discussion

We are content with the result of our work: the essence of the proof merely consists of an elegant bisimulation relation $BISIM$ together with two nice invariants (Lemma 1 and 2).

```
prove
  :lemmaOUT:
      BISIM(s,u) /\ pre(s:States[Bak], OUT(d)) /\ pre(u:States[P],OUT(d))
  /\ reachable(s:States[Bak])
  /\ s':States[Bak] = eff(s,OUT(d)) /\ u':States[P] = eff(u,OUT(d))
  => BISIM(s',u')
  ..
  set order-method either-way
  res by =>
    <> => subgoal
    ins s:States[Bak] by sc in Icount
    res by case ~sc.I.full
      <> case ~sc.I.full
      ins
        b by sc.B.bag, d by sc.O.datum,
        i by sc.O.count, q by uc.queue in DnumU
      ..
      [] case ~sc.I.full
      <> case ~(~sc.I.full)
      ins
        b by {[sc.I.datum,sc.I.count]} \U sc.B.bag , d by sc.O.datum,
        i by sc.O.count , q by uc.queue in DnumU
      ..
      [] case ~(~sc.I.full)
    [] => subgoal
  [] conjecture
```

Figure 8: LP proof of Lemma 3

Moreover, it turned out that our reasoning could rather easily be mechanised in LP. Below we report on our experiences with the Larch Prover and we compare our verification with the μCRL verification.

7.1 Remarks on the usage of LP

We think that LP is relatively easy to use, after reading the ninety pages of "A Guide to LP, The Larch Prover" [GG91] one can start using the tool. The tool is easy because: only first-order logic is supported, no tactical language is supported and the rewrite paradigm is clean and simple. An other advantage is that the specifications in LSL are not only machine readable but also human readable, the LSL specifications are close to the natural mathematical notation.

The LP proof of this paper consists of about 430 commands, and it takes 15 minutes to run on a SUN SPARCstation 10. The proof is not optimized for length or execution time: sometimes a sequence of commands is repeated with only a small change and the first version could be deleted. But we did not because a proof with a few unnecessary commands is still a proof and computers do not bother about some useless repetition.

Within the current proof management system of LP is it not possible to use a lemma before it is proved other than adding it as an axiom. So a proof must be constructed strictly bottom-up. To manage our proof of about 20 lemmas, we wrote a small program. It is a simple thirty line `nawk`[3] program that makes it easy to chose the order in which the lemmas are proved. The input is a listing of all lemmas where for each lemma the sub-lemmas that the proof uses are mentioned. Given this information the program generates the administrative begin of a LP script to prove a lemma. That is: the axioms are loaded, the sub-lemmas are assumed and the proof obligation is stated. With this method a small script for each lemma is constructed instead of one big script for all lemmas. When finished, the scripts can be glued together to construct one script for the whole proof.

Our conclusion is that LP's strong points are: easy to read specifications, easy to understand rewrite paradigm, fast rewriting, well documented. The weak points are: only first order and no tactical language. We think that the lack of a tactical language is a real disadvantage when the proof is big and a lot of "almost" repetition occurs in it. The last problem we mention is that there is no public list of know problems/bugs. We think such list is useful for the users because they could check this list when they observe something strange and there is no need to further investigate it when it is already mentioned on the list.

7.2 Comparison with the μCRL verification

Below some differences between the μCRL and the I/O automata verification are listed which we encountered during the case-study.

- In principle, the (hand-written) μCRL verification of the Bakery Protocol fits completely within the proof theory defined in [GP94].

  The proof of the Bakery Protocol given in this paper relies on meta-theory. For instance, Corollary 1 is a 'meta-result' and has not been proof-checked. Furthermore *I/O*

[3] A pattern matching, C-like programming language that comes with most UNIX versions.

automata are treated here as a meta-linguistic notion and are not available as objects in LP. Instead they are formalised via I/O automata *generators*. Furthermore, the parallel composition and hiding operator used in the specification of the protocol (*Bak*) were not formalised due to the first-order nature of LP. Recently, in [NS95] parts of the meta-theory of I/O automata are formalised in *Isabelle/HOL* which is a theorem prover based on higher-order logic.

- The μCRL proof has not been proof-checked. The main reason for this is that verifications of a similar complexity as the verification of the Bakery Protocol have already been proof-checked in Coq (e.g. see [KS94, GvdP93, BBG95]) and therefore not much new can be learned.

- The μCRL specification and verification of the Bakery Protocol are rather compact. In [GK95] there was enough space for including almost the whole hand-written proof.

  I/O-automata theory is not very suited for writing down detailed hand-written proofs. Listing here the whole hand-written proof with a similar level of detail as the μCRL proof would cost a considerable amount of space. This is one of the reasons why most of the proofs are omitted in this paper.

- In μCRL a more advanced version of the Bakery Protocol is verified. First, the ticket machine and the clock are counting modulo a fixed number (max). Second, the standing places in the bakery shop are modelled by a sequence of one-place buffers that are put in parallel.

  In this paper we have simplified the specification of the Bakery Protocol by allowing the counters to have arbitrarily large values instead of working modulo a fixed number. The reason for this is that the verification becomes less involved without harming the essence of the protocol. For a similar reason we model the standing places of the bakery shop by a bounded bag instead of a parallel composition of one-place buffers.

- μCRL is a rather technical formalism and some effort is needed to get acquainted with it.

  Many people are already familiar with invariant theory (reasoning with pre/post conditions). Therefore, for the time being, we expect that the verification given in this paper can be understood by a larger audience.

Acknowledgements

Frits Vaandrager is thanked for directing us to this subject.

References

[BBG95] M.A. Bezem, R. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. Computing Science Report 95-02, Eindhoven University of Technology, 1995.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[FKM93] J.-C. Fernandez, A. Kerbrat, and L. Mounier. Symbolic equivalence checking. In C. Courcoubetis, editor, *Proceedings of the* 5^{th} *International Conference, CAV '93, Elounda, Greece*, volume 697 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1993.

[GG] S.J. Garland and J.V. Guttag. LP: Introduction. `http://larch-www.lcs.mit.edu:8001/larch/LP/overview.html`.

[GG91] S.J. Garland and J.V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 1991.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[GK95] J.F. Groote and H.P. Korver. A correctness proof of the bakery protocol in μCRL. In Ponse et al. [PVV95], pages 63–86.

[GP94] J.F. Groote and A. Ponse. Proof theory for μCRL: a language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Proceedings of the International Workshop on Semantics of Specification Languages*, pages 232–251. Workshops in Computer Science, Springer Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In Ponse et al. [PVV95], pages 26–62.

[GvdP93] J.F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. Logic Group Preprint Series 100, Dept. of Philosophy, Utrecht University, October 1993.

[Hol91] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings International Workshop TYPES'93,* Nijmegen, The Netherlands, May 1993, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer-Verlag, 1994. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[Jon87] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1987. DoCS 87/09.

[KL93] R.P. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification,* Elounda, Greece, volume 697 of *Lecture Notes in Computer Science*, pages 166–179. Springer-Verlag, 1993.

[KS94] H.P. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94),* Sendai, Japan, volume 789 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1994.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[LSGL94] V. Luchangco, E. Söylemez, S. Garland, and N.A. Lynch. Verifying timing properties of concurrent algorithms. In *Proceedings of the Seventh International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, pages 239–259, Berne, Switzerland, October 1994. IFIP WG6.1, Elsevier Science Publishers B. V. (North Holland). Preliminary version. Final version to be published by Chapman and Hall.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.

[NS95] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In *Proceedings International Workshop TYPES'94*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[PVV95] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors. *Algebra of Communicating Processes,* Utrecht, 1994, Workshops in Computing. Springer-Verlag, 1995.

[SV89] R. de Simone and D. Vergamini. Aboard AUTO. Technical Report 111, INRIA, Centre Sophia-Antipolis, Valbonne Cedex, 1989.

A Verification model

Throughout the paper we use a very simple version of the I/O automata model. In particular, we do not need to distinguish between input and output actions. Furthermore, in this paper, we neither require fairness assumptions. So, this part of I/O automata theory is also left out. Most of the definitions given in this appendix are an adapted version of the ones given in [HSV94].

A.1 Labelled transition systems and bisimulation

Definition 1. A *labelled transition system* or *automaton* A consists of four components:

- a (finite or infinite) set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of *start states*,
- a pair $(ext(A), int(A))$ of disjoint sets of external and internal actions, respectively. The derived set $acts(A)$ of actions is defined as the union of $ext(A)$ and $int(A)$.
- a set $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ of steps.

We let s, s', u, u',.. range over states, and a,.. over actions. We write $s \xrightarrow{a}_A s'$, or just $s \xrightarrow{a} s'$ if A is clear from the context, as a shorthand for $(s, a, s') \in steps(A)$.

Let A be an automaton. An *execution fragment* of A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of A, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment that begins with a start state. A state s of A is *reachable* if it is the final state of some finite execution of A.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of A. Then $trace(\alpha)$ is the subsequence of $a_1 a_2 \cdots$ consisting of the external actions of A. For s, s' states of A and β a finite sequence of external actions of A, we define $s \stackrel{\beta}{\Rightarrow}_A s'$ iff A has a finite execution fragment with first state s, last state s' and trace β.

Definition 2. Let A and B be two automata with the same external actions. A *(weak) bisimulation* between A and B is a relation R between the states of A and B that satisfies the following four conditions:

1. If s is a start state of A, then there is a start state u in B such that $(s, u) \in R$.
2. If u is a start state of B, then there is a start state s in A such that $(s, u) \in R$.
3. If $s \xrightarrow{a}_A s'$, $(s, u) \in R$ and both s and u reachable, then there exists a state u' of B such that $u \stackrel{\beta}{\Rightarrow}_B u'$ and $(s', u') \in R$, where $\beta = trace((s, a, s'))$.
4. If $u \xrightarrow{a}_B u'$, $(s, u) \in R$ and both s and u reachable, then there exists a state s' of B such that $s \stackrel{\beta}{\Rightarrow}_A s'$ and $(s', u') \in R$, where $\beta = trace((s, a, s'))$.

Automata A and B are called *bisimilar*, notated as $A \leftrightarrows B$, iff there exists a bisimulation between them.

A.1.1 Composition

Intuitively, the composition of a collection of automata is their Cartesian product, with the added requirement that automata synchronize the performance of shared actions. This synchronization models communication between system components: if a is an external action of A and an external action of B, then the simultaneous performance of a models communication from A to B. Since we do not want synchronization involving internal actions, we require that the automata are *compatible* in the sense that they do not share these actions.

Formally, we say that action signatures $S_1, \ldots, S_n$ are *compatible* if, for all $i, j \in \{1, \ldots, n\}$ satisfying $i \neq j$, $int(S_i) \cap acts(S_j) = \varnothing$. We say that a number of automata are *compatible* if their action signatures are compatible. The *composition* $S = \prod_{i=1}^{n} S_i$ of a finite collection of compatible action signatures $S_1, \ldots, S_n$ is defined to be the action signature with

- $ext(S) = \bigcup_{i=1}^{n} ext(S_i)$,
- $int(S) = \bigcup_{i=1}^{n} int(S_i)$.

The *composition* $A = \|_{i=1}^{n} A_i$ of a finite collection of compatible automata $A_1, \ldots, A_n$ is the automaton defined as follows:

- $sig(A) = \prod_{i=1}^{n} sig(A_i)$,
- $states(A) = states(A_1) \times \cdots \times states(A_n)$,
- $start(A) = start(A_1) \times \cdots \times start(A_n)$,
- $steps(A)$ is the set of triples $(\vec{s}, a, \vec{s'})$ in $states(A) \times acts(A) \times states(A)$ such that, for all $1 \leq i \leq n$, if $a \in acts(A_i)$ then $\vec{s}[i] \xrightarrow{a}_{A_i} \vec{s'}[i]$ else $\vec{s}[i] = \vec{s'}[i]$.

We will sometimes write $A_1 \| \cdots \| A_n$ for $\|_{i=1}^{n} A_i$.

A.1.2 Hiding

If S is an action signature and $I \subseteq ext(S)$, then the action signature HIDE I IN S is defined as the pair $(ext(S) - I, int(S) \cup I)$. If A is an automaton and $I \subseteq ext(A)$, then HIDE I IN A is the automaton obtained from A by replacing $sig(A)$ by HIDE I IN $sig(A)$, and leaving all the other components unchanged.

A.1.3 Automata generators

In the automata approach, the automata that model the basic building blocks of a system are usually specified in the so-called *precondition/effect* style. In this section we will briefly describe the syntax of this language.

We start from a typed signature Σ together with a Σ-algebra $\mathcal{A}$ which gives meaning to the function and constant symbols in Σ. To describe properties, we use a first-order language over signature Σ and a set V of (typed) variables, with equality and inequality predicates, and the usual logical connectives. If ξ is a valuation of variables in their domains, and b is a formula, then we write $\mathcal{A}, \xi \models b$ if b holds in $\mathcal{A}$ under valuation ξ. A formula b is *satisfiable* if there exists a valuations ξ such that $\mathcal{A}, \xi \models b$.

An *automaton generator* G consists of five components:

- a finite action signature $sig(G)$,
- a finite set $vars(G)$ of (typed) *state variables*,
- a satisfiable formula $init(G)$, in which variables from $vars(G)$ may occur free,
- for each action $a \in acts(G)$, a *transition type*, i.e., an expression of the form

$a(y_1, \ldots, y_n)$
Precondition
b
Effect
$x_1 := e_1$
$\vdots$
$x_m := e_m$

where the y_i are (typed) variables, b is a formula in which variables from $vars(G) \cup \{y_1, \ldots, y_n\}$ may occur free, $vars(G) = \{x_1, \ldots, x_m\}$, and the e_j are expressions with the same type as x_j, in which the variables $vars(G) \cup \{y_1, \ldots, y_n\}$ may occur.

Each automaton generator G denotes an automaton A in the obvious way: states of A are interpretations of the variables of $vars(G)$ in their domains; start states of A are those states that satisfy formula $init(G)$; for each (input, output or internal) action $a \in acts(G)$ with a transition type as above, and for each choice of values $v_1, \ldots, v_n$ taken from the domains of $y_1, \ldots, y_n$, respectively, A contains an (input, output or internal) action $a(v_1, \ldots, v_n)$; A has a transition

$$s \xrightarrow{a(v_1,\ldots,v_n)} s'$$

iff there exists a valuation ξ such that

- for all $x \in vars(G)$, $\xi(x) = s(x)$,
- for $1 \leq i \leq n$, $\xi(y_i) = v_i$,
- $\mathcal{A}, \xi \models b$, and
- for $1 \leq j \leq m$, e_j evaluates to $s'(x_j)$ under ξ;

The reader will observe that the translation from automata generators to automata is quite straightforward. In fact, Lynch and Tuttle [LT87, LT89] do not even bother to distinguish between these two levels of description. For the formalization of automata theory in Larch the distinction between the semantic and syntactic levels is of course important, which is why we have discussed it here. The definition of automata generators has been inspired by similar definitions in the work of Jonsson (see, for instance, [Jon87]). In this paper we will, like Lynch and Tuttle, often refer to automata when we actually mean automata generators.

B Listing of LSL traits

Below we list the traits used in the formalization. The trait `Bak` has already been given in Figure 6. The data types **Int**, **Bag** and **Queue** are part of the library which is included in the distribution packet of the Larch tools. Together with the data types **Bool** and **Pair** which are part of the LSL language, they can be found in the Larch handbook [GH93].

B.1 P

```
P:trait
  includes Automaton(P), Queue(D,Q), CommonActions(P)
  States[P] tuple of queue : Q
  introduces
    P_max: -> Int
  asserts
    Actions[P]  generated by ENTER, OUT
    \forall u, u': States[P], m:D

      init(u) == isEmpty(u.queue);

      pre(u, ENTER(m)) == len(u.queue) < P_max;
      eff(u, ENTER(m)) == [append(m,u.queue)];

      pre(u, OUT(m)) == ~isEmpty(u.queue) /\ head(u.queue) = m;
      eff(u, OUT(m)) == [tail(u.queue)]
```

B.2 Automaton

```
Automaton(A):trait
  includes CommonActions(A)
  introduces
    init          : States[A]                          -> Bool
    pre           : States[A], Actions[A]              -> Bool
    eff           : States[A], Actions[A]              -> States[A]
    isStep        : States[A], Actions[A], States[A]   -> Bool
    null          : States[A]                          -> StepSeq[A]
    __{__,__}     : StepSeq[A], Actions[A], States[A]  -> StepSeq[A]
    Seq           : StepSeq[A], Actions[A]             -> StepSeq[A]
    execFrag      : StepSeq[A]                         -> Bool
    first,last    : StepSeq[A]                         -> States[A]
    empty         :                                    -> Trace
    __ ^ __       : Trace, ExternalActions             -> Trace
    trace         : Actions[A]                         -> Trace
    trace         : StepSeq[A]                         -> Trace
    reachable     : States[A]                          -> Bool
  asserts  \forall s, s':States[A], a, a': Actions[A], ss:StepSeq[A]
    (pre(s,a) /\ s' = eff(s,a)) == isStep(s,a,s');
    Seq(ss,a) = ss{a,eff(last(ss),a)};
    execFrag(null(s));
    execFrag(null(s'){a,s}) == isStep(s',a,s);
    execFrag((ss{a',s'}){a,s}) == execFrag(ss{a',s'}) /\ isStep(s',a,s);
    first(null(s)) == s;
    last(null(s)) == s;
    first(ss{a,s}) == first(ss);
    last(ss{a,s}) == s;
    trace(a) == if isExternal(a) then empty ^ external(a) else empty;
    trace(null(s)) == empty;
    trace(ss{a,s}) ==
```

```
      if isExternal(a) then trace(ss) ^ external(a) else trace(ss);
    init(s) \/ \E s' \E a (reachable(s') /\ isStep(s',a,s)) <=> reachable(s)
```

B.3 CommonActions

```
CommonActions(A) : trait
  includes ExternalActions
  introduces
    ENTER      : D -> Actions[A]
    OUT        : D -> Actions[A]
    external   : Actions[A] -> ExternalActions
    isExternal : Actions[A] -> Bool
  asserts \forall m: D
    external(ENTER(m)) == ENTER(m);
    external(OUT(m)) == OUT(m);
    isExternal(ENTER(m));
    isExternal(OUT(m))
```

B.4 ExternalActions

```
ExternalActions: trait
  introduces
    ENTER : D -> ExternalActions
    OUT   : D -> ExternalActions
  asserts
    ExternalActions generated by ENTER, OUT
```

B.5 Forward

```
Forward(A,B,f) : trait
  assumes Automaton(A), Automaton(B)
  introduces f : States[A], States[B] -> Bool
  asserts \forall s, s' : States[A], u : States[B], a : Actions[A],
                  alpha : StepSeq[B]
  init(s) => \E u (init(u) /\ f(s,u));
  f(s,u) /\ isStep(s,a,s') /\ reachable(s) /\ reachable(u) =>
    \E alpha (execFrag(alpha) /\ first(alpha) = u /\
              f(s', last(alpha)) /\ trace(alpha) = trace(a))
```

B.6 BISIM

```
BISIM :trait
  includes Bak,P
  introduces
    BISIM  : States[Bak], States[P] -> Bool
    BISIM  : States[P], States[Bak] -> Bool
    number : Q ,Int                 -> B
    asserts \forall s : States[Bak], u : States[P], b: B, q : Q,
             d,e : D, i:Int

BISIM(s,u) ==  (if s.I.full then {[s.I.datum,s.I.count]} else {}) \U s.B.bag \U
```

```
                (if s.O.full then {[s.O.datum,s.O.count]} else {})
                          =
                number(u.queue, (if s.I.full then s.I.count else s.I.count-1));

BISIM(u,s) = BISIM(s,u);

number(empty,i) = {};
number(append(e,q),i) = {[e,i]} \U number(q,i-1)

implies
Forward(Bak,P,BISIM), Forward(P,Bak,BISIM)
```