ALGEBRA OF STATES AND TRANSITIONS

Jan A. Bergstra

Programming Research Group, University of Amsterdam
Department of Philosophy, University of Utrecht

Abstract

We describe an algebraic specification of an abstract syntax for the construction of sequential transition systems. In these transition systems actions and states have an equally explicit role. Then the export mechanism of module algebra is used to control the visibility of states. It turns out that this leads to systems very close to the sequential processes encountered in process algebra.
This work was sponsored in part by ESPRIT project METEOR (432).

1. Introduction

This is an exercise in abstract syntax engineering. Abstract syntax engineering is a phrase that we use for the systematic development of algebraic abstract syntax for notions around computer programming, system specification and system design. If one works in the abstract syntax engineering mode the main emphasis is on the algebraic and abstract phrasing of the subject of interest. In many cases we will state propositions about the specifications without proof and we prefer to give explicit calculations and examples that illustrate the matter even better than formal and general proofs. The reason for doing so is that this work is not primarily a matter of development of theoretical computer science but rather serves as experimental work concerning algebraic specification of abstract syntax. A mistake in a theorem (proposition) should be considered as a bug that must be repaired by modifying the specification or the proposition. Thus the role of these propositions is to state what kind of properties one might wish to assert about the specification. (Of course the author has verified such propositions using informal mathematics, but such verifications may prove to be erroneous.)

The primary aim of this paper is to investigate to what extent the algebraic abstract syntax engineering approach is helpful in providing a formal sketch of the subject area and to what extent is the specification a useful basis for further specifications that address more involved and advanced topics in the same area? Evidently these questions cannot be answered by the author of this document but only by its readers. Therefore we will present experiments in abstract syntax engineering without any further comments on the effectiveness of the approach in a particular instance. The value of the work in essence is the experiment itself and not so much the quality of the result. In this sense an experiment can have a useful but negative outcome: it may provide an algebraic syntax for a topic which nevertheless fails to provide any advantages over a conventional approach.

The purpose of this particular piece of abstract syntax engineering work is to describe state transistion systems as closed expressions over a many-sorted algebraic signature. Using the technique of abstract syntax engineering we gradually extend the syntax for describing transition systems and allow for increasingly more complicated design techniques for transition system. The abstract syntax is in each case a many-sorted algebraic signature and using conditional equations it is provided with an algebraic specification that encodes the most prominent semantic aspects of the construction features for transition systems at hand.

It should be noticed that the distinction between algebraic abstract syntax engineering and algebraic specification in general is that in the special case of abstract syntax engineering each target system will be a term in a sort containing systems of an appropriate type. Many algebraic specifications, for instance the typical examples of specifications of stacks, provide a setting in which the entire specification is devoted to just modeling the stack and individual terms of type stack denote states of the stack rather than the stack mechanism. In the case of stacks it is not at

all clear however to which type the stack mechanism should belong. Abstract syntax engineering would require one to provide a general type of stack like objects (mechanisms) organised in an algebraic fashion such that a particular stack mechanism is modeled (denoted) by a closed term. This distinction between algebraic specification in general and the more restricted approach of algebraic abstract syntax engineering is of course an informal one. The underlying observation is that algebraic specifications seem to be quite adequate for the sub-problem of abstract syntax engineering whereas algebraic specification in general often leads to a stage in which the restrictions of many-sorted (conditional) equational logic are felt to be too severe.

We will use an informal style of description of algebraic specifications. This means that declaration of signature elements, variables and axioms is done in an intuitively clear but informal way without emphasis on overall consistency in the notation.

We are not aware of existing research that aims at similar goals. Already the first abstract syntax specification AST(A, S) seems to be new, which, however, hardly constitutes a virtue for something so trivial as this axiom system. The next step, however, involving export and hiding (= non-export) of states by means of an integration with module algebra of [BHK 86] seems to be novel. Interestingly this step just constitutes the transition from state transition systems to processes. Thus the conceptual issue that is addressed in this paper is how to view process as a derivative of a transition system by hiding the (names of states of the transition system). Stated differently we investigate the distinction between explicit and implicit states. This itself may have some relevance for work on the design of formal specification languages. It should be noticed however that our abstract syntax is only meant to illustrate the concepts and cannot as such be taken to be a fragment of the abstract syntax for a specification language with practical ambitions.

One may question the use of hiding states in a transition system. It is not clear which role that mechanism has to play in systems engineering, specification and verification. Rather than answering this question we suggest to put forward the thesis that for every structuring mechanism in a specification language there must be a corresponding abstraction mechanism that allows to remove the structure introduced by the mechanism. Thus together with the structuring of abstract data types there comes information hiding as an abstraction mechanism. Together with structured sequential programming there is denotational semantics which provides a meaning for a program independent of its structure. Together with the visible actions of process algebra there is the silent (hidden) action of Milner's CCS. This complementarity of declaration and visibility control ought to be systematic for all major ingredients of systems specification languages.

2 AST (algebra of states and transitions / algebra of sequential transition systems)

The specification AST(A, S) provides abstract syntax for the construction of finite transition systems with states in a set S and actions taken from a set A. These sets are parameters of the construction. Formally we introduce a constant for each action in A and these constants are collected in a sort AA. Similarly for each state in S there is a corresponding constant in a sort SS. The availability of these constants allows to state most relevant facts in terms of closed expressions.

The transition systems described in AST are not equipped with a root or with any marker that indicates a current state. If needed such features can be added. The specification AST says nothing more than that a transition system consists of a set of states and a set of transitions. The constructors are therefore the empty transition system, a function T that introduces a single state as a transition system, the introduction of a single transition and the combination (union) of transition systems. The only non-trivial axiom says that a transition declares its initial and

final state. With asmodule (abstract syntax module) we will denote specification modules that contain at least one sort which allows to denote systems that go beyond the level of a basic data type. Regrettably this is a vague distinction. Formally it plays no role of course. Specifications of elementary data types are headed with the keyword 'module'.

Conventions used in writing the specifications.

(i) The variables which have been defined in a module M are exported to all other modules that import M.

(ii) The name of a module often contains several parameters placed between brackets. These denote sets such that for each element of the set a constant is introduced in a corresponding sort. This is done in order to have many closed terms and to be able to formulate theorems about closed terms only.

(iii) No use is made of parametrisation and of hidden functions and sorts. The meaning of all specification modules is determined by expanding each of its imports.

(iv) If a module A imports module B then it exports al components and equations of A as well. Thus the import instruction can simply be considered as an abbreviation of the imported module.

(v) In each case the initial algebra of the specification is one of the intended models, i.e. the specification is not just a collection of interesting laws (or uninteresting according to the reader's taste) but the specification should ensure certain relevant properties for the initial algebra already. In many cases the specifications are supposed to have a loose semantics as well, which means that the initial algebra has one or more useful and non-trivial homomorphic images.

asmodule AST (A, S)	(algebra of states and transitions over actions A and states S)
begin	
sorts	
AA	(actions)
SS	(states)
TS	(transition systems)
constants	
$\underline{\alpha}: \rightarrow AA$	(for $\alpha \in A$)
$\underline{\sigma}: \rightarrow SS$	(for $\sigma \in S$)
$\emptyset: \rightarrow TS$	(empty transition system)
functions	
$T: SS \rightarrow TS$	(declaration of a state for a TS)
$(. \rightarrow . \rightarrow .): SS \times AA \times SS \rightarrow TS$	(declaration of a state transition)
$+: TS \times TS \rightarrow TS$	(combination/overlap of transition systems)

variables

$x, y, z :\rightarrow TS$, $a :\rightarrow AA$, $s, t :\rightarrow SS$

equations

$x + y = y + x$

$(x + y) + z = x + (y + z)$

$x + x = x$

$x + \emptyset = x$

$(s \rightarrow a \rightarrow t) = (s \rightarrow a \rightarrow t) + T(s) + T(t)$

end

To any closed expression over $\Sigma(AST(A, S))$ of sort TS one can assign a transition system. The states of that transition system are all states occurring in the expression and the transitions are also exactly the transitions that are enumerated in the expression.

The initial algebra I(AST(A, S)) is the standard model for this specification. Indeed let P and Q be two closed expressions of sort TS then these have the same interpretation in the initial algebra if and only if they denote isomorphic transition systems.

A transition system is called *deterministic* if for every state and action there exists at most one outgoing transition from the state labeled by the action. A transition system that is not deterministic is called *non-deterministic*.

Examples of TS-expressions (the underlying state space and action alphabet are left implicit).

(ii) $(s1 \rightarrow a1 \rightarrow t1) + (s2 \rightarrow b \rightarrow t2) + (s1 \rightarrow c \rightarrow s2) + (s2 \rightarrow c \rightarrow s4) +$
$(s4 \rightarrow a \rightarrow s1) + T(s5) + T(t3)$

(ii) $A = (box\_open \rightarrow close \rightarrow box\_closed) + (box\_closed \rightarrow open \rightarrow box\_open)$

(iii) $B = A + (box\_open \rightarrow insert\_a \rightarrow box\_open * a\_\in\_box) +$
$(box\_open * a\_\in\_box \rightarrow take\_a \rightarrow box\_open)$

(iv) $C = B + (box\_open \rightarrow insert\_b \rightarrow box\_open * b\_\in\_box) +$
$(box\_open * b\_\in\_box \rightarrow take\_b \rightarrow box\_open)$

(iv) $D = C + (box\_open * a\_\in\_box \rightarrow close \rightarrow box\_closed * a\_\in\_box) +$
$(box\_closed * a\_\in\_box \rightarrow open \rightarrow box\_open * a\_\in\_box) +$
$T(box\_closed * b\_\in\_box)$

In the examples (ii)-(iv) an operator * is used that constructs states from attributes of states. This operator is informal in the sense that it should be viewed as a part of the naming mechanism of states in these examples. Of course it is meaningful to incorporate a combination function on state attributes within the signature. This will be done in section 6.

It should be noticed that AST provides an algebra in which finite automata can be specified as they are, whereas process oriented modeling of finite automata unavoidably leads to some form of behavioral abstraction that may conceivably be unwanted for some purposes.

3 BMAST(A, S)

3.1 The axiom system BMAST(A, S)

It is possible to define many more operators on transition systems than just those of AST. In this section we will extend AST to a module algebra setting (c.f. [BHK 86]) where the visible signature of a transition system is taken to be the collection of its states. Hiding (non-export) involves making some states invisible. In [GV 88] it is made very explicit that the name of a state is immaterial and that only the behavior of a state counts. The hiding mechanism allows to forget the identity of a state and to protect a state from modification of its immediate behavior by combination with another transition system. As such this hiding mechanism can simply be viewed as a form of information hiding and protection. Once a state has been hidden it has become impossible to add new incoming or outgoing transitions for this state within any context of it. So the virtue of BMAST is supposed to be that it explains exactly how to proceed from explicit (named) states to implicit (unnamed) states. Unavoidably the introduction of BMAST below raises the question whether partially hidden states can be imagined as well. We will address that topic in section 6 where so-called signals are introduced. These signals provide visible but non-behavioral aspects for hidden states.

As a first step we extend AST to a setting with sets of states and include a function that computes the state space of a transition system. There is an overloading of $\varnothing$ which denotes both the empty state space and the empty transition system. In this case we propose that equations that have ambiguous meaning are asserted for all correct type assignments to their components.

```
asmodule AST(A, S)/P                (AST with state spaces)
begin
    import
        AST(A, S)
    sort
        PSS                         (powerset over SS / state spaces / signatures)
    constant
        ∅:→PSS                      (empty state space)
    functions
        i: SS → PSS                 (embedding of SS into PSS)
```

T: PSS → TS (extension of T to PSS)

+: PSS x PSS → PSS (combination of state spaces)

Σ: TS → PSS (state space/visible state space of a transition system)

variables

u, v, w: → PSS

equations for PSS(+, ∅)

$u + v = v + u$

$(u + v) + w = u + (v + w)$

$u + u = u$

$u + \emptyset = u$

axioms for T on PSS

$T(\emptyset) = \emptyset$

$T(i(s)) = T(s)$

$T(u + v) = T(u) + T(v)$

axioms for the visible state space operator

$\Sigma(s \rightarrow a \rightarrow t) = i(s) + i(t)$

$\Sigma(T(u)) = u$

$\Sigma(x + y) = \Sigma(x) + \Sigma(y)$

end

This module can be extended to a module algebra where parts of the state space can be hidden. We adhere to the presentation of the axioms of module algebra in [BHK 86] as much as possible to help the reader in recognizing the axioms and the special modifications that have been applied. As a preparation we introduce the booleans as well as a module that enriches AST(A, S)/P with equality functions on states and actions.

module BOOL

begin

sort

BOOL

constants

T: → BOOL

F: → BOOL

functions

¬: BOOL → BOOL

∧: BOOL x BOOL → BOOL

∨: BOOL x BOOL → BOOL

equations

$\neg T = F, \neg F = T$

$T \wedge T = T, T \wedge F = F, F \wedge T = F, F \wedge F = F$

$T \vee T = T, T \vee F = T, F \vee T = T, F \vee F = F$

end

The next module adds equality functions on actions and states and intersection on state spaces, as well as some abbreviations that will be used lateron.

asmodule AST(A, S)/P, eq

begin

import

BOOL, AST(A, S)/P

functions

eq: AA x AA → BOOL	(equality on actions)
eq: SS x SS → BOOL	(equality on states)
∩: PSS x PSS → PSS	(intersection of signatures)

equations for the eq functions

eq(a, a) = T	
eq(c, d) = F	for different c and d in A
eq(s, t) = T	
eq(σ, τ) = F	for different σ and τ in S

equations for PSS(+, ∩, ∅)

$u \cap v = v \cap u$

$(u \cap v) \cap w = u \cap (v \cap w)$

$u \cap u = u$

$u \cap \varnothing = \varnothing$

$(u + v) \cap w = (u \cap w) + (v \cap w)$

$eq(s, t) = F \rightarrow i(s) \cap i(t) = \varnothing$

conditions/abbreviations

$s \in u$	for	$i(s) \cap u = i(s)$
$s \notin u$	for	$i(s) \cap u = \varnothing$
$u \subseteq v$	for	$u + v = v$

end

In the above specification there is an entry called conditions/abbreviations. Under this heading one finds declarations of abbreviations of identities. It is then allowed to use these abbreviations in the notation of conditions for conditional equations. Of course if the module is imported by another module the right to use these abbreviations is imported as well.

The following module introduces hiding of states. The transition systems of this module will have explicit and implicit states simultaneously.

asmodule BMAST(A, S) (basic module algebra of states and transitions)

begin

import

AST(A, S)/P, eq

sort

AR (atomic renamings)

constant

Id: $\rightarrow$ AR (identity renaming)

functions

ar: SS x SS $\rightarrow$ AR (atomic renaming construction / permutation)

$\cdot$: AR x SS $\rightarrow$ SS (application of permutation)

$\cdot$: AR x PSS $\rightarrow$ PSS (application of permutation)

$\cdot$: AR x TS $\rightarrow$ TS (application of permutation)

Σ: AR $\rightarrow$ PSS (signature of renaming)

$\Box$: PSS x TS $\rightarrow$ TS (export operator)

variables

r: $\rightarrow$ AR, t': $\rightarrow$ SS

equatiόns for AR regarding SS and PSS

$ar(s, s) = Id$

$ar(s, t) = ar(t, s)$

$\Sigma(Id) = \varnothing$

$eq(s, t) = F \rightarrow \Sigma(ar(s, t)) = i(s) + i(t)$

$ar(s, t) \cdot s = t$

$eq(s, t) = F \,\&\, eq(s, t') = F \rightarrow ar(t, t') \cdot s = s$

$r \cdot i(s) = i(r \cdot s)$

$r \cdot (u + v) = (r \cdot u) + (r \cdot v)$

$r \cdot \varnothing = \varnothing$

module algebra axioms on the sort TS

$\Sigma(s \rightarrow a \rightarrow t) = i(s) + i(t)$ [S1']

$\Sigma(T(u)) = u$ [S2]

$\Sigma(x + y) = \Sigma(x) + \Sigma(y)$ [S3]

$\Sigma(u \,\Box\, x) = u \cap \Sigma(x)$ [S4]

$\Sigma(r \cdot x) = r \cdot \Sigma(x)$ [S5]

$x + y = y + x$ [C1]

$(x + y) + z = x + (y + z)$ [C2]

$T(x) + T(y) = T(x + y)$ [C3]

$x + T(\Sigma(x)) = x$ [C4]

$x + x = x$ [C5-]

$r \cdot (s \rightarrow a \rightarrow t) = (r \cdot s \rightarrow a \rightarrow r \cdot t)$ [R1']

$r \cdot T(u) = T(r \cdot u)$ [R2]

$r \cdot (x + y) = (r \cdot x) + (r \cdot y)$ [R3]

$r \cdot (u \,\square\, x) = (r \cdot u) \,\square\, (r \cdot x)$ [R4]

$r \cdot (r \cdot x) = x$ [R5]

$\Sigma(r) \cap \Sigma(x) = \varnothing \rightarrow r \cdot x = x$ [R6']

$\Sigma(x) \,\square\, x = x$ [E1]

$u \,\square\, (v \,\square\, x) = (u \cap v) \,\square\, x$ [E2]

$u \,\square\, (T(v) + x) = T(u \cap v) + (u \,\square\, x)$ [E3]

$\Sigma(x) \cap \Sigma(y) \subseteq u \rightarrow u \,\square\, (x + y) = (u \,\square\, x) + (u \,\square\, y)$ [E4]

end

Comments on the axioms.

(i) The axioms for module algebra that have been included above exclude the axiom C5 from [BHK 86], $x + (u \,\square\, x) = x$ which has been omitted because it seems to have no intuitive plausibility in this case at all. The difficulty with this axiom is that it allows to add to a transition system a copy of itself with as an alternative with all states hidden. Put in a context that adds additional transitions this will make a difference however. It seems to be the case that this axiom is characteristic for purely declarative modules. The presence of behavioral aspects makes it implausible. The present axiom C5- is a weaker version of C5.

(ii) The axioms C1 and C2 have been included to enhance the similarity of this axiom table with that of [BHK86] but were already included in AST(A, S). Similarly the axioms S1', S2 and S3 have been included but were already contained in AST(A, S) /P.

(iii) The axioms S1', R1' and R6' are minor modifications of the corresponding axioms in [BHK86]. The modifications are caused by the fact that the construction of atomic systems (i.e. the single transitions) differs from the construction of atomic modules in BMA[fol] of [BHK86].

(iv) We provide an explanation of each of the axioms in informal terms in the comments below. These explanations should provide the reader with an intuitive understanding of BMAST(A, S). It should be noticed that we are working within an axiomatic approach and that

the axioms are not primarily designed as to specify a given semantic model. The primary intuitions are these:

(1) The name of a hidden state is unimportant. This implies that the name may be changed as long as name clashes are avoided. A renaming mechanism is introduced to express this matter.

(2) What matters about hidden states is the behavior that is shown from these states. Moreover a system is always supposed to start its active life in a visible state, which implies that invisible states matter only in as far as these can be reached via a sequence of actions from a visible state.

(3) The axioms of BMAST(A, S) express these intuitions only partially and more axioms supported by these intuitions will be added lateron. The main virtue of BMAST(A, S) is that it allows to prove a normalisation theorem for closed expressions of sort TS.

(v) The axioms for atomic renamings explain that an atomic renaming $r = ar(s, t)$ permutes the names s and t in all objects to which it is applied.

(vi) The axioms S1', S2-5 provide a recursive definition of the visible state space of a transition system. The interesting axiom is S4 it says that the visible state space of $u \,\square\, x$ contains only those visible states of x that are contained in u.

(vii) The axioms C1-3 and C5- correspond to the intuition that transition systems are sets of states and transitions. C4 states that a system contains a declaration of all of it's visible states.

(viii) The axioms R1', R2-5, R6' serve as a recursive definition of renaming on transition systems. R4 and R6 require additional explanation. R4 states that in a name permutation is to be applied to a transition system it can be applied on both the export signature (state space) and the hidden part. The justification for this is just the fact that renamings are permutations (and indeed the very reason to choose permutations as atomic renamings. Indeed let s be a hidden state name (i.e. $s \in \Sigma(x)$ but $s \notin u$) then after application of r on $u \,\square\, x$ one has $r \cdot s \notin r \cdot u$. Thus a hidden state cannot become visible after renaming. Further because permutations are injective no name clashes of any kind can occur. R6' states that renaming of a hidden name into a different hidden name does not change a transition system. This is the intuition formulated in (iv-1) above. The combined effects of the axioms discussed until now is already nontrivial. We provide some examples:

(1) Let $A = (i(s1) + i(s3)) \,\square\, ((s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3))$ and

$B = (i(s1) + i(s3))) \,\square\, (ar(s2, t2) \cdot ((s1 \rightarrow a \rightarrow t2) + (t2 \rightarrow b \rightarrow s3))$

Then $ar(s2, t2) \cdot A = A$ because of R6'. Applying R4 one obtains:

$A = ar(s2, t2) \cdot A =$

$(ar(s2, t2) \cdot (i(s1) + i(s3))) \,\square\, (ar(s2, t2) \cdot ((s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3))) =$

$(i(s1) + i(s3))) \,\square\, (ar(s2, t2) \cdot ((s1 \rightarrow a \rightarrow t2) + (t2 \rightarrow b \rightarrow s3))) = B$

(2) With A and B as in (1): $A = A + A = A + B$. It follows that duplication of hidden states is allowed. Indeed if the true identity of a state is immaterial one cannot distinguish between different copies of a state that have the same behavior.

(ix) E1 says that exporting all visible states of a transition system will leave the system unchanged, E2 says that repeated exports can be combined into single exports. E3 allows to omit hidden states for which there is no incoming or outgoing transition. This may be worth an illustration:

Let $D = (i(s1) + i(s3)) \Box (T(i(t2) + i(s2)) + (s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3))$ and A as in (viii) above, then

$D = T((i(s1) + i(s3)) \frown (i(t2) + i(s2))) +$

$((i(s1) + i(s3)) \Box ((s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3))) = T(\varnothing) + A = T(\varnothing) + A + T(\Sigma(A)) = A + T(\varnothing) + T(i(s1) + i(s3)) = A + T(\varnothing + i(s1) + i(s3)) = A + T(i(s1) + i(s3)) = A + T(\Sigma(A)) = A.$

(x) E4 allows to reduce the number of export operator applications in an expression. The effect is again illustrated by an example. Let A and B be as in the examples of (viii) above. Then $A + B = (i(s1) + i(s3)) \Box ((s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3) + (s1 \rightarrow a \rightarrow t2) + (t2 \rightarrow b \rightarrow s3)$ using a single application of E4.

FIRST NORMAL FORM THEOREM

For every closed TS-expression X over $\Sigma(BMAST(A, S))$ there exists an expression Y of the form $u \Box (X_1 + .. + X_n + T(v))$ with X_i of the form $(s \rightarrow a \rightarrow t)$ and such that X and Y are provably equivalent within BMAST(A, S).

The proof involves induction on the structure of the expression and follows the lines of the normal form theorem in [BHK 86].

3.2 The bisimulation model of BMAST(A, S).

The bisimulation model of BMAST(A, S) requires a definition of substantial length. We assume that S is infinite (otherwise the construction fails!). A may be finite or infinite. First of all we need a domain for the model. This is made up from so-called TS-objects. A TS-object X is a triple $X = (VS, IS, TR)$ with $VS \subseteq IS \subseteq S$, $TR \subseteq IS \times A \times IS$ where:

VS a finite collection of states, the visible states,

IS the finite collection of internal states including both visible and hidden states,

TR a (necessarily finite) collection of labeled transitions on the internal states.

With TSO(A, S) we denote the class of TS-objects over A and S. (Of course generalisations to objects with infinitely many states and transitions can easily be made.)

Let X = (VSX, ISX, TRX) and Y = (VSY, ISY, TRY) be two TS-objects. We say that these bisimulate if the following conditions are satisfied:

(i) VSX = VSY (we will write VS for both of them in the remainder of this definition)

(ii) There exists a relation $R \subseteq ISX \times ISY$ such that:

(a) R(s, s) for s in VS,

(b) R(s, t) and $s \in VS$ implies s = t

(c) R(s, t) and $t \in VS$ implies s = t

(d) if $s \rightarrow a \rightarrow s' \in TSX$ and R(s, t) then there exists $t' \in ISY$ such that $t \rightarrow a \rightarrow t' \in TSY$ and R(s', t')

(e) if $t \rightarrow a \rightarrow t' \in TSX$ and R(s, t) then there exists $s' \in ISX$ such that $s \rightarrow a \rightarrow s' \in TSX$ and R(s', t').

Notice that if all hidden states of a TS-object are renamed a bisimilar object is obtained. This notion of a bisimulation is a straightforward adaptation of the original definition of bisimulation due to Park [P 81] that was made popular by its application in CCS in [M 80]. We will now define the operations of BMAST on the bisimulation classes of TS-objects.

Therefore the next step is to equip the class of TS-objects with the necessary operations and additional sorts. Of course for AA and SS one can take the sets A and S itself. Then for PSS one takes the collection of finite subsets of SS and AR consists of Id together with all subsets of cardinality 2 of SS. Let again X = (VSX, ISX, TRX) and Y = (VSY, ISY, TRY) be two TS-objects, and let $u \in PSS$. Then:

(i) $\Sigma(X) = VSX$

(ii) T(u) = (u, u, ∅)

(iii) $r \cdot X = (r \cdot VSX, r \cdot ISX, r \cdot TRX)$ where renaming works pointwise on all sets involved; moreover in the case of the set of transitions the renaming of a single transition amounts to application of the renaming to both states of the transition.

(iv) $u \,\square\, X = (u \cap VSX, ISX, TRX)$

(v) $X + Y = (VSX \cup VSY, ISX \cup ISY, TRX \cup TRY)$ where we assume that $ISX \cap ISY = VSX \cap VSY$. If this condition is not satisfied renamings are applied to the hidden states of X in order to ensure this condition first. Indeed all hidden names of X can be changed while staying in the same bisimulation class.

The structure thus obtained is denoted with TSO(A, S)/bs-eq. Without proof we state that TSO(A, S)/bs-eq satisfies all axioms of BMAST(A, S). The verification of this fact is simple for all equations indeed. So we have:

PROPOSITION 1. TSO(A, S)/bs-eq $\models$ BMAST(A, S)

There is a non-trivial (or rather debatable) aspect in the definition of bisimulation between TS-objects. This is that hidden states need not be in the domain of the bisimulation. It follows that we do not imagine circumstances in which control of the transition system is in an arbitrary hidden state. Indeed the intuition is that control always has to start at a visible state. Therefore control can never arrive in a hidden state which is not accessible from a visible one via a sequence of transitions, from which it follows that such non-reachable hidden states are immaterial indeed.

3.3 Sequential composition (implicit definition)

Let begin, end and int be three special objects in S.
We define the *sequential composition* of X and Y on TSO(A, S)/bs-eq as follows:

$X \cdot' Y = (\Sigma(X + Y)) \,\Box\, (ar(\underline{int}, \underline{end}) \cdot X + ar(\underline{int}, \underline{begin}) \cdot Y)$

Here it is assumed that $\underline{int}$ is a state not in $\Sigma(X) + \Sigma(Y)$. We use $\cdot'$ instead of $\cdot$ in order to prepare for a later redefinition of $\cdot'$ that avoids the condition about int.

The idea is that the final state of X and the initial state of Y are identified and that this state in turn is made a hidden one. Technically the identification is implemented by renaming both states to a common new state int which is then hidden. An axiomatic specification of sequential composition takes the form of a conditional equation:

$z \notin \Sigma(X + Y) \rightarrow X \cdot' Y = (\Sigma(X + Y)) \,\Box\, (ar(z, end) \cdot X + ar(z, begin) \cdot Y).$

It follows from the axioms of BMAST(A, S) that there is a unique interpretation of sequential composition on TSO(A, S)/bs-eq that satisfies this defining axiom. Notice that this defining axiom for sequential composition provides an implicit definition. It would be better to have sequential composition explicitly defined as a term with two free variables over the signature of BMAST(A, S $\cup$ {begin, int , end}). This will be done below in 4.1.

An interesting question is whether one can find simple identities over Σ(BMAST(A, S)) that are valid in the bisimulation model but not deducible from BMAST(A, S) . A typical example of such an axiom is entrance distribution (ED):

$s \in u \,\&\, s' \in u \,\&\, t \notin u \,\&\, t' \notin u \rightarrow$

$u \,\Box\, ((s \rightarrow a \rightarrow t) + (s' \rightarrow a' \rightarrow t') + x) = u \,\Box\, ((s \rightarrow a \rightarrow t) + x) + u \,\Box\, ((s' \rightarrow a' \rightarrow t') + x)$

The intuitive meaning of this axiom is as follows: both transitions $(s \rightarrow a \rightarrow t)$ and $(s' \rightarrow a' \rightarrow t')$ constitute steps from the visible part to the invisible part of the state set of $u \,\Box\, ((s \rightarrow a \rightarrow t) + (s' \rightarrow a' \rightarrow t') + x)$. We call such transitions entrances. The axiom tells in this case that one

may copy the hidden parts of the system into two disjointly named but isomorphic copies in such a way that both copies have only one of both entrances. This axiom suffices to prove the identity $(X + Y) \cdot' Z = X \cdot' Z + Y \cdot' Z$ for all closed expressions X, Y such that the state end of X + Y (provided it exists) has no outgoing transitions in X + Y.

An example of this axiom is as follows: take

$X = (s1 \rightarrow a1 \rightarrow s) + (s2 \rightarrow a2 \rightarrow s) + (s \rightarrow b \rightarrow t) + (s \rightarrow c \rightarrow s)$ and

$Y = (i(s1) + i(s2) + i(t)) \,\square\, X$, then:

$Y = (i(s1) + i(t)) \,\square\, ((s1 \rightarrow a1 \rightarrow s) + (s \rightarrow b \rightarrow t) + (s \rightarrow c \rightarrow s)) +$
$(i(s2) + i(t)) \,\square\, ((s2 \rightarrow a2 \rightarrow s) + (s \rightarrow b \rightarrow t) + (s \rightarrow c \rightarrow s))$

4 BMAST(A, S)/M

4.1 Entrance distribution revisited

We will now provide a more concise formulation of the entrance distribution axiom. This requires the definition of several auxiliary operators. These auxiliary operators are of independent use so the effort of specifying them is considered useful anyhow and the improved formulation of the entrance distribution axiom is just an additional bonus. We assume the presence in S of different states begin, int and end. This generates constants begin, int and end for AA. In order to simplify notation begin, int and end are now formally introduced as constants for PSS to denote the images of the respective constants in PSS.

asmodule BMAST(A, S)/(ran, dom, out, Δ)
begin
  import
    BMAST(A, S)
  constants
    begin: → PSS (initial state)
    int: → PSS (intermediate state)
    end: → PSS (final state)
  functions
    -: PSS x PSS → PSS (subtraction of state spaces)
    out: PSS x TS → TS (out(u, x) selects the transitions of x that leave from a state in u or from a hidden state)
    dom: TS → PSS (dom(x) determines those visible states of x that have an outgoing transition)
    ran: TS → TS (selects the visible states with an incoming transition)
    Δ: PSS x TS → TS (hiding = non-export)

equations for begin, int , end

$$\text{begin} = i(\underline{\text{begin}})$$

$$\text{int} = i(\underline{\text{int}})$$

$$\text{end} = i(\underline{\text{end}})$$

equations for -

$$\varnothing - u = \varnothing$$

$$u - (v + w) = (u - v) - w$$

$$(u + v) - w = (u - w) + (v - w)$$

$$u - u = \varnothing$$

$$eq(s, t) = F \rightarrow i(s) - i(t) = i(s)$$

axiom for hiding

$$u \Delta x = (\Sigma(x) - u) \,\square\, x$$

equations for out

$$out(u, \varnothing) = \varnothing$$

$$out(u, T(v)) = \varnothing$$

$$out(u, x + y) = out(u, x) + out(u, y)$$

$$s \notin u \rightarrow out(u, (s \rightarrow a \rightarrow t)) = \varnothing$$

$$s \in u \rightarrow out(u, (s \rightarrow a \rightarrow t)) = (s \rightarrow a \rightarrow t)$$

$$out(u, v \Delta x) = v \Delta out(u + v, x)$$

equations for dom

$$dom(\varnothing) = \varnothing$$

$$dom(T(u)) = \varnothing$$

$$dom(x + y) = dom(x) + dom(y)$$

$$dom(s \rightarrow a \rightarrow t) = i(s)$$

$$dom(u \,\square\, x) = u \cap dom(x)$$

equations for ran

$$ran(\varnothing) = \varnothing$$

$$ran(T(u)) = \varnothing$$

$$ran(x + y) = ran(x) + ran(y)$$

$$ran(s \rightarrow a \rightarrow t) = i(t)$$

$$ran(u \,\square\, x) = u \cap ran(x)$$

abbreviation/condition

$$process(x) \text{ for } \Sigma(x) = \text{begin} + \text{end} \;\&\; \underline{\text{begin}} \notin ran(x) \;\&\; \underline{\text{end}} \notin dom(x)$$

end

Comments on the axioms of BMAST(A, S)/(ran, dom, out, Δ).

(i) Subtraction of state sets is an instance of subtraction of sets and the axioms are fairly obvious. Notice the following identity that is derivable for closed expressions of sort PSS:

$u \cap (v - w) = (u \cap v) - w$

(ii) Hiding is complementary to export: rather than to name the states that have to be exported one names the states that are not to be exported. For closed expressions many useful laws about hiding can be derived from the defining axiom and the BMAST(A, S), we mention some examples:

$\Sigma(u \Delta x) = \Sigma(x) - u$

$u \Box x = (\Sigma(x) - u) \Delta x$

$r \cdot (u \Delta x) = (r \cdot u) \Delta (r \cdot x)$

$\varnothing \Delta x = x$

$u \Delta (v \Delta x) = (u \cup v) \Delta x$

$u \Delta (T(v) + x) = T(v - u) + (u \Delta x)$

$u \Delta x = u \Delta (T(u) + x)$

$\Sigma(x) \cap \Sigma(y) \cap u = \varnothing \rightarrow u \Delta (x + y) = (u \Delta x) + (u \Delta y)$.

(iii) The function out(u, x) removes from a transition system x all transitions that start in a visible state outside u and thereafter removes all visible states which have no remaining ingoing or outgoing transitions. Said differently all transitions from a visible state in u or from a hidden state are filtered out. The intention is to restrict a transition system to its behavior as far as it is observable from an initial point in u.

(iv) An example for the application of the axioms for the function out is as follows: let

$X = (s1 \rightarrow a \rightarrow s2) + (s2 \rightarrow b \rightarrow s3) + (s2 \rightarrow c \rightarrow s4)$

$Y = (i(s2) \Delta X) + (s4 \rightarrow d \rightarrow s5)$

Then:

$out(i(s1), Y) = out(i(s1), i(s2) \Delta X) + out(s4 \rightarrow c \rightarrow s5) = out(i(s1), i(s2) \Delta X) =$

$i(s2) \Delta out(i(s1) + i(s2), X) = i(s2) \Delta X$.

(v) The function dom computes all visible states of its argument from which an outgoing transition is possible. The functioin ran computes the visible states to which an ingoing transition is possible.

(vi) Interesting identites about out, dom, ran provable for closed terms are:

$out(u, x) = out(u \cap \Sigma(x), x)$

$x = out(\Sigma(x), x) + T(\Sigma(x))$

$out(u, out(v, x)) = out(u \cap v, x)$

$out(u, x) = out(u \cap dom(x), x)$

$\Sigma(out(\Sigma(x), x)) = dom(out(\Sigma(x), x)) + ran(out(\Sigma(x), x))$

(vii) Let X be a closed expression of sort TS. Then define the sequence X_n as follows.

$X_0 = out(\varnothing, X)$, $X_{n+1} = out(\Sigma(X_n), X)$. After finitely many steps this sequence will arrive at a fixed point equal to $out(\Sigma(X), X)$.

(viii) A transition system is called a process if its only visible states are begin and end and if begin has only outgoing transitions and end has only ingoing transitions.

The next module introduces sequential composition (multiplication in terms of process algebra) by means of an explicit definition, as well as a generalised form ED' of the axiom ED and an axiom that allows to remove larger inaccesible parts of a state space than is already allowed by the axiom E3 of BMAST(A, S).

asmodule BMAST(A, S)/M

import

BMAST(A, S)/(ran, dom, out, Δ)

function

$\cdot : TS \times TS \to TS$ (sequential composition)

axioms

$out(u, x) = out(u, y) \to u \Delta (x + y) = (u \Delta x) + (u \Delta y)$ [ED']

$out(v + w, x) = out(v, x) + out(w, x)$ [LO]

$dom(x) = \varnothing \to x = T(\Sigma(x))$ [ISSR]

$x \cdot y = int \Delta ((ar(\underline{end}, \underline{int}) \cdot (int \Delta x)) + (ar(\underline{begin}, \underline{int}) \cdot (int \Delta y)))$ [SC]

end

Comments on the axioms.

(i) The names of the equations have the following explanations: ED' denotes an improved version of the entrance distribution axiom ED. LO denotes the linearity of out. ISSR stands for inaccessible state space removal. SC stands for the defining axiom for sequential composition.

(ii) The intuition behind SC is clear from previous discussions. As said before the only improvement of SC upon the defining axioms provided before is that it takes the form of an explicit definition by means of a polynomial expression.

(iii) The consequences of ISSR are illustrated as follows:

(1) $i(s2) \Box (s1 \to a \to s2) = T(s2)$.

(2) $(i(s2) + i(s4)) \Box ((s1 \to a \to s2) + T(s3) + (s1 \to b \to s4) = T(s2) + T(s4)$.

It is easy to prove that ISSR holds in the bisimulation model. Similarly one may easily verify LO in the bismulation model.

(iv) The intuition of ED' is not so easy to explain. It is not difficult to explain, however, why ED' is valid in the bisimulation model in the case that it is applied on transition systems X and Y without hidden states and with $\Sigma(X) = \Sigma(Y)$ and under the assumption that $u \subseteq \Sigma(X)$. Indeed suppose that in these circumstances $out(u, X) = out(u, Y)$ and let $A = u \Delta (X + Y)$ and $B = (u \Delta X) + (u \Delta Y)$. Now clearly $\Sigma(A) = \Sigma(B)$. For a bisimulation we need a relation between the hidden states of A and B. Now notice that for each element u, B will contain exactly two copies as hidden states (whereas A has only one). One copy is inherited from X the other one is inherited from Y. The bisimulation relation will relate a hidden state s in u of A with both of its counterparts in B. In checking that this is a bisimulation relation indeed one

uses that out(u, X) = out(u, Y). With some effort the general case can be reduced to this restricted case.

Open problem.
Can one find a finite set of axioms that proves all identities that hold in the bisimulation model for finite transition systems?

In principle this must be possible because equality in the bisimulation model is a decidable property. The decision method needs no more than an exhaustive search over the finite (but large) number of possible bisimulation relations between two closed module objects. The problem is quite related to proving the equivalence of regular expressions, finite automata or regular processes.

4.2 A second normal form theorem.

It is possible to prove a second normal form theorem. This second normal form theorem allows to present a transition system in such a way that all hidden states of the normal form can be reached from a visible state in a finite number of steps. For a proper definition of the second normal form we need the notion of a computation sequence.

4.2.1 Computation sequences

A (finite) computation sequence for a transition system is a (finite) sequence of transitions $(si \rightarrow ai \rightarrow ti)$ such that for each n the target state of the n-th transition coincides with the initial state of the (n + 1)-th transition (if it exists).

SECOND NORMAL FORM THEOREM

For every closed TS-expression X over $\Sigma(BMAST(A, S))$ there exists an expression Y of the form $u \,\square\, (X_1 + .. + X_n + T(v))$ with each X_i of the form $(s \rightarrow a \rightarrow t)$ and such that X and Y are provably equivalent within BMAST(A, S)/M. In addition it is required that

(i) $u \subseteq w = \Sigma(X_1 + .. + X_n + T(v))$,

(ii) Every state in w lies on some computation sequence that starts in some state in u.

PROOF. Rather than a proof we provide an example of normalisation in the second style. This example features all complications that will arise in a full proof. Let the following TS-expression be given, in first normal form:
$X = (i(b1) + i(e1) + i(e2) + i(b2)) \,\square\, ((b1 \rightarrow a1 \rightarrow c1) + (c1 \rightarrow a2 \rightarrow e1) + (c2 \rightarrow a2 \rightarrow c1) + (c2 \rightarrow a3 \rightarrow e2) + T(c3))$. The normalisation to second normal form works as follows.
Let $U = i(b1) + i(e1) + i(e2) + i(b2)$ and
$X1 = (b1 \rightarrow a1 \rightarrow c1) + (c1 \rightarrow a2 \rightarrow e1) + (c2 \rightarrow a2 \rightarrow c1) + (c2 \rightarrow a3 \rightarrow e2)$

$X2 = X1 + T(c3)$ then $X = U \,\square\, (X2)$. Now:
(1) $V = \Sigma(X2) = i(b1) + i(c1) + i(c2) + i(c3) + i(e1) + i(e2)$
(2) $X2 = \Sigma(X2) \,\square\, X2 = V \,\square\, X2$
(3) $X = U \,\square\, (V \,\square\, X2) = (U \cap V) \,\square\, X2 = W \,\square\, X2$ with $W = i(b1) + i(e1) + i(e2)$.
(4) $X = W \,\square\, X2 = W \,\square\, (X1 + T(c3)) = W \,\square\, (X1 + T(i(c3))) = T(W \cap i(c3)) + W \,\square\, X1 = T(\varnothing) + W \,\square\, X1 = W \,\square\, X1$
(5) Decompose X1 as follows: $X1,1 = (b1 \rightarrow a1 \rightarrow c1) + (c1 \rightarrow a2 \rightarrow e1)$, $X1,2 = (c2 \rightarrow a2 \rightarrow c1) + (c2 \rightarrow a3 \rightarrow e2)$, $X1 = X1,1 + X1,2$ here X1,1 contains all states and transitions that will appear in the second normal form (i.e. the normal form that we are looking for will be the expression $W \,\square\, X1,1$).
(6) $W \,\square\, X1 = W \,\square\, (W + i(c1)) \,\square\, X1))$. Now $(W + i(c1)) \,\square\, X1 = (W + i(c1)) \,\square\, (X1,1 + X1,2) = ((W + i(c1)) \,\square\, X1,1) + (W + i(c1)) \,\square\, X1,2)$ because $\Sigma(X1,1) \cap \Sigma(X1,2) = i(c1) \subseteq W + i(c1))$. Further $dom((W + i(c1)) \,\square\, X1,2) = \varnothing$ and consequently $(W + i(c1)) \,\square\, X1,2 = T(\Sigma(W + i(c1)) \,\square\, X1,2)) = T(i(e2))$. It follows that $X = W \,\square\, X1 = W \,\square\, ((W + i(c1)) \,\square\, X1,1 + T(i(e2)))$.
(7) $X = W \,\square\, ((W + i(c1)) \,\square\, X1,1 + T(e2)) = T(W \cap i(e2)) + W \,\square\, ((W + i(c1)) \,\square\, X1,1) = T(i(e2)) + W \,\square\, ((W + i(c1)) \,\square\, X1,1) = T(i(e2)) + (W \cap (W + i(c1))) \,\square\, X1,1 = T(i(e2)) + W \,\square\, X1,1 = T(i(e2)) + W \,\square\, X1,1 + T(\Sigma(W \,\square\, X1,1)) = T(i(e2)) + W \,\square\, X1,1 + T(W) = W \,\square\, X1,1 + T(i(e2) + W) = W \,\square\, X1,1 + T(W) = W \,\square\, X1,1$. This expression is in second normal form indeed.

4.3 Sequential composition revisited

The module M provides an explicit definition of sequential composition which we will take as the definitive definition of sequential composition in terms of BMAST. Indeed the hiding operator can be replaced by an application of export provided subtraction of state sets is allowed. It should be noted that the definition differs from the earlier definition ($\cdot'$) in case x or y contains int.

At this point it is in order to state (without proof) the following proposition.

PROPOSITION 2. For all closed TS-expressions X, Y and Z over $\Sigma(BMAST(A, S))$:

$BMAST(A, S)/M \quad \vdash \quad out(end, X + Y) = \varnothing \rightarrow (X + Y) \cdot Z = X \cdot Z + Y \cdot Z$
$BMAST(A, S) \quad \vdash \quad (X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

At this stage a connection can be made with process algebra as described in [BK 84]. Let A play the role of the collection of atomic actions (excluding the deadlock action δ). Then one can translate the atomic actions of process algebra as TS-expressions as below:

$[a] = (\underline{begin} \rightarrow a \rightarrow \underline{end})$

$[\delta] = T(begin + end)$.

The simplest structuring operations of process algebra are + (alternative composition) and · (sequential composition). Translations of these operations is obvious:

$[x + y] = [x] + [y]$

$[x \cdot y] = [x] \cdot [y]$.

Using this translation all finite closed process expressions of $BPA_\delta(A)$ can easily be coded as a finite closed TS-expression. Thus the process expression $a \cdot (b \cdot \delta + c \cdot d \cdot e)$ is translated to $[a] \cdot ([b] \cdot [\delta] + [c] \cdot [d] \cdot [e])$. We obtain the following proposition.

PROPOSITION 3. Suppose that X, Y and Z are closed process expression over $BPA_\delta(A)$. The following formal proofs exist in this case. :

$BMAST(A, S)/M \;\&\; process([X]) \;\&\; process([Y]) \;\&\; process([Z]) \vdash$

$X + Y = Y + X$

$(X + Y) + Z = X + (Y + Z)$

$X + X = X$

$(X + Y) \cdot Z = X \cdot Z + Y \cdot Z$

$(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$

$X + \delta = X$

$\delta \cdot X = \delta$

These are in fact the axioms A1-7 of ACP, the algebra of communicating processes in [BK 84]. Rather than a proof of the proposition we provide examples of the last four equations that are not themselves axioms of BMAST.

(i) $((\underline{begin} \rightarrow a \rightarrow \underline{end}) + (\underline{begin} \rightarrow b \rightarrow \underline{end})) \cdot (\underline{begin} \rightarrow c \rightarrow \underline{end}) =$

$int \Delta ((ar(\underline{end}, \underline{int}) \cdot (int \Delta ((\underline{begin} \rightarrow a \rightarrow \underline{end}) + (\underline{begin} \rightarrow b \rightarrow \underline{end})))) +$

$(ar(\underline{begin}, \underline{int}) \cdot (int \Delta (\underline{begin} \rightarrow c \rightarrow \underline{end})))) =$

$int \Delta ((ar(\underline{end}, \underline{int}) \cdot ((\underline{begin} \rightarrow a \rightarrow \underline{end}) + (\underline{begin} \rightarrow b \rightarrow \underline{end}))) +$

$(ar(\underline{begin}, \underline{int}) \cdot (\underline{begin} \rightarrow c \rightarrow \underline{end}))) =$

$int \Delta ((\underline{begin} \rightarrow a \rightarrow \underline{int}) + (\underline{begin} \rightarrow b \rightarrow \underline{int}) + (\underline{int} \rightarrow c \rightarrow \underline{end})) =$

$int \Delta ((\underline{begin} \rightarrow a \rightarrow \underline{int}) + (\underline{int} \rightarrow c \rightarrow \underline{end}) + (\underline{begin} \rightarrow b \rightarrow \underline{int}) + (\underline{int} \rightarrow c \rightarrow \underline{end})) = *$

$int \Delta ((\underline{begin} \rightarrow a \rightarrow \underline{int}) + (\underline{int} \rightarrow c \rightarrow \underline{end})) +$

$int \Delta ((\underline{begin} \rightarrow b \rightarrow \underline{int}) + (\underline{int} \rightarrow c \rightarrow \underline{end})) =$

$((\underline{begin} \rightarrow a \rightarrow \underline{end}) \cdot (\underline{begin} \rightarrow c \rightarrow \underline{end})) +$

$((\underline{begin} \rightarrow b \rightarrow \underline{end}) \cdot (\underline{begin} \rightarrow c \rightarrow \underline{end}))$.

The identity * depends on the axiom ED'. This requires the following verification.
out(int, (begin → a → int) + (int → c → end)) = (int → c → end) =
out(int, (begin → b → int) + (int → c → end)).

(ii) ((begin → a → end) · (begin → b → end)) · (begin → c → end) =
(int Δ ((begin → a → int) + (int → b → end))) · (begin → c → end) =
(i(int1) Δ ((begin → a → int1) + (int1 → b → end))) · (begin → c → end) =
(int Δ ((i(int1) Δ ((begin → a → int1) + (int1 → b → int)))) + (int → c → end)) =
(int Δ ((i(int1) Δ ((begin → a → int1) + (int1 → b → int)))) +
(i(int1) Δ (int → c → end))) =
int Δ (i(int1) Δ ((begin → a → int1) + (int1 → b → int) + (int → c → end))) =
(int + i(int1)) Δ ((begin → a → int1) + (int1 → b → int) + (int → c → end))) =
using a similar derivation =
(begin → a → end) · ((begin → b → end) · (begin → c → end)).

(iii) (begin → a → end) + [δ] = (begin → a → end) + T(begin + end) =
(begin → a → end) + T(i(begin)) + T(i(end)) = /AST/ = (begin → a → end).

(iv) δ · (begin → a → end) = T(begin + end) · (begin → a → end) =
int Δ (T(begin) + (int → a → end))) = T(begin) + (int Δ (int → a → end)) = /ISSR/ =
T(begin) + T(end) = [δ].

5 Product of transition systems.

In this section we will describe some additional operators on transition systems. The operators of 5.1 will be introduced once more in section 6 in the context of list structured states. The description in this section serves to display the specifications of these structuring operations in their simplest context.

5.1 State removal and operational encapsulation

An operation that is needed when stepwise design of a transition system is performed is the removal of a state and all transitions leading to and from it. This is a form of encapsulation. It declares states inaccesible rather than invisible. Similarly one many need an operation that removes all transitions involving a certain action. The second mechanism is called operational encapsulation. Let H be a subset of A.

asmodule BMAST(A, S)/∂_H, Δ

begin

import

BMAST(A, S)

functions

Δ: PSS x TS $\rightarrow$ TS (state removal)

∂_H: TS $\rightarrow$ TS (encapsulation, for each $H \subseteq A$)

axioms for state removal

$\Delta_U(x + y) = \Delta_U(x) + \Delta_U(y)$

$\Delta_{U+V}(x) = \Delta_U(\Delta_V(x))$

$\Delta_U(\varnothing) = \varnothing$

$\Delta_U(T(v)) = T(v-u)$

$s \in u \rightarrow \Delta_U(s \rightarrow a \rightarrow t) = \varnothing$

$t \in u \rightarrow \Delta_U(s \rightarrow a \rightarrow t) = \varnothing$

$u \cap \Sigma(x) = \varnothing \rightarrow \Delta_U(x) = x$

$v \subseteq \Sigma(x) \rightarrow \Delta_U(v \,\square\, x) = v \,\square\, \Delta_{U \cap V}(x)$

axioms for operational encapsulation

$\partial_H(x + y) = \partial_H(x) + \partial_H(y)$

$\partial_H(T(u)) = T(u)$

$\partial_H(s \rightarrow \underline{\alpha} \rightarrow t) = s \rightarrow \underline{\alpha} \rightarrow t$ for $\alpha \in H$

$\partial_H(s \rightarrow \underline{\alpha} \rightarrow t) = T(s) + T(t)$ for $\alpha \notin A - H$

$\partial_H(u \,\square\, x) = u \,\square\, \partial_H(x)$

end

5.2 A pairing function of the state space

The next step is to introduce structure on the state space. The presence of an injective pairing function <.,.> will be required in the specification below.

asmodule BMAST(A, S)/∂_H, Δ, <,>

begin

import

BMAST(A, S)/ ∂_H, Δ

functions

<.,.>: SS x SS $\rightarrow$ SS (pairing of states)

<.,.>: PSS x SS $\rightarrow$ PSS (lright global pairing on state sets)

<.,.>: SS x PSS $\rightarrow$ PSS (left global pairing on state sets)

<.,.>: PSS x PSS $\rightarrow$ PSS (global pairing on state sets)

<.,.>: TS x SS $\rightarrow$ TS (left global pairing on TS)

$<.,.>: SS \times TS \to TS$ (right global pairing on TS)

variables

$s1, s2, t1, t2: \to SS$

axioms for equality on states

$<s1, t1> = <s2, t2> \to s1 = s2$

$<s1, t1> = <s2, t2> \to t1 = t2$

$eq(s, t) = eq(t, s)$

$eq(\underline{\sigma}, <s, t>) = F$ for each state $\sigma \in S$

$eq(<s1, t1>, <s2, t2>) = eq(s1, s2) \wedge eq(t1, t2)$

axioms for global pairing on state sets

$<u, \varnothing> = \varnothing$

$<s, i(t)> = i(<s, t>)$

$<s, u + v> = <s, u> + <s, v>$

$<\varnothing, u> = \varnothing$

$<i(t), s> = i(<t, s>)$

$<u + v, s> = <u, s> + <v, s>$

$<u, i(s)> = <u, s>$

$<u, v + w> = <u, v> + <u, w>$

$<i(s), u> = <s, u>$

$<v + w, u> = <v, u> + <w, u>$

equations for global pairing on TS

$<s, \varnothing> = \varnothing$

$<s, T(t)> = T(<s, t>)$

$<s, (t1 \to a \to t2)> = (<s, t1> \to a \to <s, t2>)$

$<s, x + y> = <s, x> + <s, y>$

$<s, u \square x> = <s, u> \square <s, x>$

$<\varnothing, s> = \varnothing$

$<T(t), s> = T(<t, s>)$

$<(t1 \to a \to t2), s> = (<t1, s> \to a \to <t2, s>)$

$<x + y, s> = <x, s> + <y, s>$

$<u \square x, s> = <u, s> \square <x, s>$

end

5.3 The systems product operator based on state pairing

After tedious preparations it is possible to introduce an operator ⊗ which yields the cartesian product of two transition systems. We add a mechanism that allows synchronous operation/communication of actions.

asmodule BMAST(A, S)/⊗

begin

import

BMAST(A, S)/ ∂_H, Δ, <,>

constant

δ: → AA (deadlock / impossible action)

functions

⊗: TS x TS → TS (product of transition systems)

| : AA x AA → AA (communication function on actions)

variables

b, c: → AA

equations for the communication function

a | b = b | a

(a | b) | c = a | (b | c)

δ | a = δ

equation for δ

(s → δ → t) = T(s) + T(t)

axioms for product

∅ ⊗ x = ∅

x ⊗ ∅ = ∅

T(i(s)) ⊗ x = <s, x>

x ⊗ T(i(s)) = <x, s>

(s1 → a → s2) ⊗ (t1 → b → t2) = (<s1, t1> → a | b → <s2, t2>) +

(<s1, t1> → a → <s2, t1>) + (<s1, t2> → a → <s2, t2>) +

(<s1, t1> → b → <s1, t2>) + (<s2, t1> → b → <s2, t2>)

(x + y) ⊗ z = (x ⊗ z) + (y ⊗ z)

x ⊗ (y + z) = (x ⊗ y) + (x ⊗ z)

(u □ x) ⊗ (v □ y) = (<u, v>) □ (x ⊗ y)

end

5.4 A comment on the connection with process algebra

At this point we can provide a definition of the merge of processes in the terminology of AST. (Recall that a process is a transition system X that satisfies the condition process(X).):

$X \parallel Y =$

$T(\text{begin} + \text{end}) \,\square\, (\text{ar}(\underline{\text{begin}}, \langle \underline{\text{begin}}, \underline{\text{begin}} \rangle) \cdot (\text{ar}(\underline{\text{end}}, \langle \underline{\text{end}}, \underline{\text{end}} \rangle) \cdot (X \otimes Y)))$.

Using this definition the interpretation of finite closed process algebra expressions that was given in section 4.3 can be extended to expressions involving merge and encapsulation.

$[x \parallel y] = [x] \parallel [y]$

$[\partial_H(x)] = \partial_H([x])$

Let γ be a communication function. The axioms that determine this communication function are collected in a set $E(\gamma) = \{\underline{\alpha} \mid \underline{\beta} = \underline{\kappa}, \mid \gamma(\alpha, \beta) = \kappa\}$. Without its tedious proof we state the following proposition:

PROPOSITION 4. Let A be an action alphabet and let γ be a commutative and associative communication function on this alphabet for which δ acts as a zero. Suppose that X and Y are closed process expressions over the action alphabet A involving the process composition operators δ, +, $\cdot$, $\parallel$ and ∂_H then

$ACP(A) \cup E(\gamma) \vdash X = Y$ if and only if $BMAST(A, S)/\otimes \cup E(\gamma) \vdash [X] = [Y]$.

6 Signals

Another feature that is reasonable to add is the attachment of attributes or signals to states. SGN is a new sort of objects. It is generated by a finite set of atomic signals ATSGN structured with a commutative, associative and idempotent combination function &. There is an intersection (restriction) operator $\cap$: SGN x SGN $\rightarrow$ SGN. This operator allows to select a subset of the signals which in fact means that it provides an quite rudimentary abstraction mechanism on signals. The algebra of signals is worth an independent specification.

asmodule BMAST(A, S)/SGN(K)
begin
  import
    BMAST(A, S)
  sorts
    ATSGN (atomic signals)
    SGN (signals)
  constants
    $\varnothing$:$\rightarrow$SGN (empty signal)
    $\underline{\sigma}$:$\rightarrow$ATSGN (for $\sigma \in$ K)

functions

i: ATSGN → SGN (embedding of atomic signals in signals)

&: SGN x SGN → SGN (signal combination)

∩: SGN x SGN → SGN (signal intersection)

∩: SGN x TS → TS (signal filtering)

<sig:.at.>: SGN x SS → TS (signal /state attribution)

variables

σ, τ, ρ: SGN

equations for the boolean algebra of signals

$\sigma \& \tau = \tau \& \sigma$

$(\sigma \& \tau) \& \rho = \sigma \& (\tau \& \rho)$

$\sigma \& \sigma = \sigma$

$\sigma \& \varnothing = \sigma$

$\sigma \cap \tau = \tau \cap \sigma$

$(\sigma \cap \tau) \cap \rho = \sigma \cap (\tau \cap \rho)$

$\sigma \cap \sigma = \sigma$

$\sigma \cap \varnothing = \varnothing$

$(\sigma \& \tau) \cap \rho = (\sigma \cap \rho) \& (\tau \cap \rho)$

$i(\underline{\sigma}) \cap i(\underline{\tau}) = \varnothing$ for σ and τ different signals in K

axioms for signal attribution

<sig: σ at s> = <sig: σ at s> + T(s)

<sig: σ at s> + <sig: τ at s> = <sig: σ & τ at s>

T(s) = T(s) + <sig: ∅ at s>

axioms for signal filtering

$\sigma \cap T(u) = T(u)$

$\sigma \cap \varnothing = \varnothing$

$\sigma \cap (x + y) = (\sigma \cap x) + (\sigma \cap y)$

$\sigma \cap (s \rightarrow a \rightarrow t) = (s \rightarrow a \rightarrow t)$

$\sigma \cap$ <sig: τ at s> = <sig: $\tau \cap \sigma$ at s>

$\sigma \cap (u \,\square\, x) = u \,\square\, (\sigma \cap x)$

end

The function <sig: . at . >: SGN x SS → TS adds to a state s a signal σ. The signals of a state remain visible even if the state itself is hidden. A model of BMAST(A, S) + SGN(K) is obtained as follows: define TS-objects as before but take care of the signals. These have to be collected from their declarations and combined as labels in their states. Then a bisimulation relation R may only relate states with equal signals irrespective of whether these states are visible or not. An example of an expression involving signals:

$$
\begin{aligned}
A = \quad & \langle \text{sig: red at 1} \rangle + \langle \text{sig: green at 2} \rangle + \langle \text{sig: blue at 3} \rangle + \\
& \langle \text{sig: yellow at 4} \rangle + T(1/2 + 2/3 + 3/4 + \text{begin} + \text{end}) \\
B = \quad & (\text{begin} \rightarrow \text{push1} \rightarrow 1) + \\
& (1 + 2) \,\square\, (A + (1 \rightarrow \text{wait} \rightarrow 1/2) + (1/2 \rightarrow \text{wait} \rightarrow 2)) + \\
& (2 + 3) \,\square\, (A + (2 \rightarrow \text{push1} \rightarrow 2/3) + (2/3 \rightarrow \text{wait} \rightarrow 4)) + \\
& (4 \rightarrow \text{stop} \rightarrow \text{end})
\end{aligned}
$$

The use of signals is in modeling systems that have states which are essentially unknown to an outside world but which can show information otherwise than by (non-)performing behavior. Most computer systems are within that category because in most cases it is impossible to reconstruct a memory state of a machine in each and every detail. The sceen contents of a micro computer can be modeled as signals for instance while the actions performed by a user are indeed modeled as actions.

7 States with list structure

In this section we will work on basis of AST(A, S)/P as given in section 3.1 and ignore visibility control of states. The problem is to provide operators on the initial algebra of AST(A, S)/P that are of practical help in the design of transition systems. The signature of AST allows only unstructured designs. Again this section is an experiment on abstract syntax engineering rather than that it provides ready made tools for the construction of transition systems. A constraint on the design of the operators is that each notation for a transition system must allow a normal form over AST(A, S), i.e. the additional operators must allow elimination, at least when applied on closed expressions. Because the states are essentially data, the question how to provide a structuring mechanism for states would be solved if a canonical approach to data structuring would be known. This however seems not to be the case. In section 5 we have used pairing as a structuring mechanism but that is not very practical if one aims at the description of concrete systems.

The approach of this section is to select an arbitrary choice of initial data structuring and to use that as the only structuring mechnism. As an initial structuring mechanism we will consider lists. Thus we will impose the list construction on the sort SS. Then the foremost structuring operation available is list concatenation, this operator is extended to transition systerms under the name global concatenation. This operator takes a transition system and a list in SS and concatenates each of the states occurring in the transition system with the list. The operation has a left version and a right version. Moreover it is useful to have sets of states available and to distribute these operations over set union. (These sets are specified again though this shows poor modularisation of the familiy of specifications throughout the paper.)

7.1 Concatenation operators on states, state sets and transition systems

The specification of this section introduces a list structure on states with concatenation as the main structuring operator. Concatenation is extended to sets of states and transition systems. In fact the appearance of this specification can be viewed as a sign of bad modularisation of the family of specifications in this paper because its equations almost coincide with those given for the pairing operators in section 5. The reasons not to use parametrisation in this case is that the clarification of semantic issues involved would take more energy and space than the duplication involved in the next module.

asmodule AST(A, S)/L
begin
  import
    AST(A, S)/P
  functions
    *: SS x SS → SS (concatenation of states)
    *: PSS x SS → PSS (right global concatenation)
    *: SS x PSS → PSS (left global concatenation)
    *: PSS x PSS → PSS (global concatenation)
    *: SS x TS → TS (left global concatenation)
    *: TS x SS → TS (right global concatenation)
    *: PSS x TS → TS (lifted left global concatenation)
    *: TS x PSS → TS (lifted right global concatenation)
  associativity of *
    (s * t) * r = s * (t * r)
  equations for * on SS, PSS and TS
    all axioms for global pairing of BMAST(A, S)/(∂H, Δ, <,>) with <,> replaced by *
  additional equations for lifted (left and right) global concatenation on state sets and
    transition systems
    ∅ * x = ∅
    i(s) * x = s * x
    (u + v) * x = (u * x) + (v * x)
    x * ∅ = ∅
    x * i(s) = x * s
    x * (u + v) = (x * u) + (x * v)
end

7.2 Example of a structured design of a transition system: box_coin_table_book

A1 = (box_closed → open → box_open) + (box_open → close → box_closed)

A2 = (box_open → verify_box_open → box_open) +
(box_closed → verify_box_closed → box_closed)

A = (A1 + A2) * (i(coin_∈_box) + i(coin _∉_box))

B1 = (coin_∈_box → take → coin_∉_box) + (coin_∉_box → put → coin_∈_box)

B2 = (coin_∈_box → verify_coin_∈_box → coin_∈_box) +
(coin_∉_box → verify_coin_∉_box → coin_∉_box)

B = box_open * (B1 + B2)

C = A + B (= A * Σ(B) + Σ(A) * B)

D = (box_on_table → put_on_floor → box_on_floor) +
(box_on_floor → put_on_table → box_on_table)

E = D * Σ(C) + Σ(D) * C

F1 = (book_on_box → take_book → box_free) +
(box_free → put_book → book_on_box)

F2 = (box_on_table * box_open * coin_∉_box * box_free → destruct_box → no_box) +
(no_box → create_box → box_on_table * box_closed * coin_∉_box * box_free)

G = Σ(F1) * E + F1 * Σ(E) + F2

With the last step in this design we run into difficulties with describing the transition system because the expression G allows transitions to and from impossible states such as box_on_table * box_open * coin_∉_box * book_on_box (this is considered impossible because a book can't be placed on the open box).

In order to proceed with the design of the transition system a state removal operator is needed that restricts a transition system that has grown too big by removing impossible states and transitions to and from these states.

7.3 State removal and operational encapsulation with list structured states

The specification of this state encapsulation requires booleans, an equality function on states, extension of concatenation to sets of states and test on inclusion for states and state sets. We will not impose apriori restrictions on the possible states based on some kind of meaning of the atomic state components. It follows that e.g. box_open * box_closed is a valid state in spite of the contradiction that is implicit in the mnemoninc meaning of the state components involved. If such a state is to be excluded it should be avoided in the construction of transition systems.

asmodule AST(A, S)/P,L
begin
  import
    BOOL
    AST(A, S)/P
    AST(A, S)/L
  functions
    eq: SS x SS → BOOL
    eq: PSS x PSS → BOOL
    Δ: PSS x TS → TS (state removal)
    ∂_H: TS → TS (operational encapsulation)
  variables
    r1, r2: → SS
  equations for eq
    eq(s, s) = T
    eq(σ, τ) = F for σ and τ with σ ≠ τ
    eq(σ * t, τ) = F
    eq(σ, τ * r) = F
    eq(σ * r1, τ * r2) = F for σ and τ with σ ≠ τ
    eq(s * r1, s * r2) = eq(r1, r2)
  the equations for state removal and operational encapsulation as given in BMAST(A, S)/∂_H,Δ
end

Within this module a subsequent step in the design of the example is possible with an appropriate definition of G:

$$G' = \Delta_{\Sigma(D)} * i(box\_open) * \Sigma(B) * i(book\_on\_box)((\Sigma(F1) * E) + (F1 * \Sigma(E)) + F2)$$

7.4 Example: a transition system for one lift serving three floors and two persons

This second extended example will also motivate the introduction of an additional operator after some steps.

A =
(lift_at_floor(1) → move_up(1, 2) → lift_at_floor(2)) +
(lift_at_floor(2) → move_up (2, 3)→ lift_at_floor(3)) +

(lift_at_floor(3) → move_down (3, 2)→ lift_at_floor(2)) +
(lift_at_floor(2) → move_down(2, 1) → lift_at_floor(1)) +

(lift_at_floor(1) → open_door → lift_at_floor(1) * d_open) +
(lift_at_floor(2) → open_door → lift_at_floor(2) * d_open) +
(lift_at_floor(3) → open_door → lift_at_floor(3) * d_open) +

(lift_at_floor(1) * d_open → close_door → lift_at_floor(1)) +
(lift_at_floor(2) * d_open → close_door → lift_at_floor(2)) +
(lift_at_floor(3) * d_open → close_door → lift_at_floor(3))

locations_p(1) = p(1)_at_floor(1) + p(1)_at_floor(2) + p(1)_at_floor(3) + p(1)_in_lift
locations_p(2) = p(2)_at_floor(1) + p(2)_at_floor(2) + p(2)_at_floor(3) + p(2)_in_lift
B = A * locations_p(1) * locations_p(2)

C(1) =
lift_at_floor(1) * d_open * (p(1)_at_floor(1) → p_in_lift → p(1)_in_lift) +
lift_at_floor(2) * d_open * (p(1)_at_floor(2) → p_in_lift → p(1)_in_lift) +
lift_at_floor(3) * d_open * (p(1)_at_floor(3) → p_in_lift → p(1)_in_lift) +
lift_at_floor(1) * d_open * (p(1)_in_lift → p_to_floor → p(1)_at_floor(1)) +
lift_at_floor(2) * d_open * (p(1)_in_lift → p_to_floor → p(1)_at_floor(2)) +
lift_at_floor(3) * d_open * (p(1)_in_lift → p_to_floor → p(1)_at_floor(3))

C(1, 2) = C(1) * locations_p(2)

C(2) =
lift_at_floor(1) * d_open * locations_p(1) *
((p(2)_at_floor(1) → p_in_lift → p(2)_in_lift) + (p(2)_in_lift → p_to_floor → p(2)_at_floor(1))) +

lift_at_floor(2) * d_open * locations_p(1) *

((p(2)_at_floor(2) → p_in_lift → p(2)_in_lift) + (p(2)_in_lift → p_to_floor → p(2)_at_floor(2))) +

lift_at_floor(3) * d_open * locations_p(1) *

((p(2)_at_floor(3) → p_in_lift → p(2)_in_lift) + (p(2)_in_lift → p_to_floor → p(2)_at_floor(3))).

D = B + C(1, 2) + C(2)

E =

(up_signal_off_at_fl(1) → push_up_signal_fl(1) → up_signal_on_at_fl(1) +

(up_signal_off_at_fl(2) → push_up_signal_fl(2) → up_signal_on_at_fl(2) +

(down_signal_off_at_fl(2) → push_down_signal_fl(2) → down_signal_on_at_fl(2) +

(down_signal_off_at_fl(3) → push_down_signal_fl(3) → down_signal_on_at_fl(3)

F =

(up_signal_on_at_fl(1) → arr_lift _at_1→ up_signal_off_at_fl(1) +

(up_signal_on_at_fl(2) → ar_lift_at_2_from_1 → up_signal_off_at_fl(2) +

(down_signal_on_at_fl(2) → arr_lift_at_2_from_3 → down_signal_off_at_fl(2) +

(down_signal_on_at_fl(3) → arr_lift_at_3 → down_signal_off_at_fl(3)

At this point we encounter a difficulty because if D, E and F are to be combined it must be said that arrival of a lift results in releasing the signal pointing in the direction of progress of the lift. Part of the solution will be to introduce the product based on state concatenation which allows a product of state spaces to be equipped with transitions that are derived up from the transitions of the components.

7.4 Systems product based on state concatenation

A further composition principle for transition systems is the product. The product operator has been introduced in section 5 but then based on pairing of states as a state structuring mechanism. Because we intend to apply the product to the specification of lifts that was presented in the previous section a product operator $\otimes*$ based on concatenation is needed. In fact the specification of product can be viewed as being parametrised by a binary composition operator on states. In section 5 the role of a composition operator was played by pairing. In this section the role of pairing will be replaced by list concatenation.

So it is assumed that states are concatenations of states from both factors of the product and that a transition with action a is possible if one of the composed processes can perform such an action, moreover if one system can perform action a and the other one can perform b the merge can perform action a | b provided this is not δ. The function | explains what happens if two actions are executed concurrently. The action δ is a new one which plays a

formal role only because it cannot be performed as is clear from the equation $(s \rightarrow \delta \rightarrow t) = T(s) + T(t)$. The specification of product is in the module below. It should be noticed that unexpected things may happen to $X \otimes_* Y$ if $\#(\Sigma(X) * \Sigma(Y)) < \#(\Sigma(X)) \times \#(\Sigma(Y))$ which is possible because * is taken to be associative. At this point we encounter a methodological difficulty. If * is not made associative the structure of the states becomes quite complex and product will not be associative. (Indeed the product based on pairing is neither commutative nor associative.) If we take * associative, the product will be associative as well but there is the problem that the same state can perhaps be obtained as a concatenation of different pairs of states. If we choose * to be commutative as well product will become commutative but the risk that states can be constructed in different ways increases (moreover the definition of equality on states has to be reconsidered if * is made commutative and its character shifts to that of set union rather than string concatenation). Here we are trapped in a seemingly unavoidable problem. The solution chosen in our definitions is an arbitrary one based on the decision that list formation will be the only data structuring primitive at this stage. Then it should be taken as a design rule for writing a specification that for each instance $X \otimes_* Y$ of product one ensures that $\#(\Sigma(X) * \Sigma(Y)) = \#(\Sigma(X)) \cdot \#(\Sigma(Y))$.

It should be mentioned that the product operator $\otimes_*$ as described below is a rather simple one in the sense that it will not provide any means for a transition system to perform its actions depending on signals of the other transition system. In other words our transition systems are missing the feature of guarded commands. Such a mechanism can be worked out. The reason not to do so here is that once again one has to select one of many options and once that has been done the matter is quite simple, but the task to motivate the selection of the option remains hard. Only a significant collection of examples suffices to generate confidence in the chosen mechanism indeed.

We proceed with a presentation of the module $AST(A, S)/\otimes_*$. This module results from $AST(A, S)/\otimes$ by replacing $\otimes$ by $\otimes_*$ throughout the module.

```
asmodule AST(A, S)/⊗*
begin
    import
        AST(A, S)/L
    constant
        δ: → AA                    (deadlock / impossible action)
    functions
        ⊗*: TS x TS → TS           (product based on concatenation)
        | : AA x AA → AA           (communication function on actions)
    variables
        b, c: → AA
```

equations

the equations introduced in BMAST(A, S)/⊗ with ⊗ replaced by $\otimes_*$ and <,> replaced by *.

end

Equipped with the merge operator we can make further progress with the lift example. We proceed by adding components to the unfinished specification of 7.3.

F1 =

(up_signal_off_at_fl(1) → arr_lift_at_1 → up_signal_off_at_fl(1) +

(up_signal_off_at_fl(2) → ar_lift_at_2_from_1 → up_signal_off_at_fl(2) +

(down_signal_off_at_fl(2) → arr_lift_at_2_from_3 → down_signal_off_at_fl(2) +

(down_signal_off_at_fl(3) → arr_lift_at_3 → down_signal_off_at_fl(3)

Then the communication function is specified as follows where only the pairs a | b have been listed for which communication will not produce the action δ.

non-trivial communications

begin

arr_lift_at_1 | move_down(2, 1)	=	move_arr(2, 1)
arr_lift_at_2_from_1 | move_up(1, 2)	=	move_arr(1, 2)
arr_lift_at_2_from_3 | move_down(3, 2)	=	move_arr(3, 2)
arr_lift_at_3 | move_up(2, 3)	=	move_arr(2, 3)

end

H = {arr_lift_at_1 , move_down(2, 1) , arr_lift_at_2_from_1 , move_up(1, 2), arr_lift_at_2_from_3, move_down(3, 2), arr_lift_at_3, move_up(2, 3)}

Finally the following description of the lift system can be given.

$$LIFT(1, 2, 3, p(1), p(2)) = \partial_H((D + E) \otimes_* (F + F1))$$

Acknowledgements. Hans Mulder has been helpful in finding the improved version ED' of ED. Moreover several discussions with Kees Middelburg and Martin Kooy (both with PTT RNL in Leidschendam) have to be mentioned, in which they have pointed out to the author that the formal distinction between implicit and explicit states might even be a practically useful one.

8 References

[BHK 86] J.A.Bergstra, J.Heering & P.Klint, *Module algebra*, Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam, (May 1986), Revised version 1988, to appear in JACM

[BK 84] J.A.Bergstra & J.W.Klop, *Process algebra for synchronous communication*, Information and Control 60 (1/3), p. 109-137, (1984)

[GV 89] J.F.Groote & F.W.Vaandrager, *Structured operational semantics and bisimulation as a congruence*, Report CS- R8845, Centre for Mathematics and Computter Science, Amsterdam (1988)

[M 80] R.Milner, *A calculus of communicating systems*, Springer LNCS 92 (1980)

[P 81] D.Park, *Concurrency and automata on infinite sequences*, Proc. 5th GI conference Springer 104, (1981)

Reports of the Programming Research Group, University of Amsterdam

---

P8701 **R.A. Groenveld**
Verification of a sliding window protocol by means of process algebra

P8702 **M.R. Dasselaar**
Development of an expert system, from theory to practice

P8703 **A.V. Hurkmans**
Een declaratieve en procedurele kennisrepresentatievorm voor kennissystemen, toegepast op NEXT

P8704 **F. Wiedijk**
Termherschrijfsystemen in Prolog

P8705 **J.L.M. Vrancken**
The algebraic specification of semicomputable datatypes

P8706 **J.C.M. Baeten, J.A. Bergstra & J.L.M. Vrancken**
Processen en procesexpressies

P8707 **J.H. Verhagen**
Expertsystemen bij de Nederlandse Spoorwegen

P8708 **S. Mauw**
Process algebra as a tool for the specification and verification of CIM-architectures

P8709 **J.C.M. Baeten & J.A. Bergstra**
Global renaming operators in concrete process algebra (revised version)

P8710 **J. Treur**
Volledigheid en definieerbaarheid in diagnostische redeneersystemen

P8711 **J. Treur**
Een logische analyse van diagnostische redeneerprocessen; redeneren met en over hypothesen

P8712 **W. Syski & J. Treur**
Reasoning about uncertainty represented by meta-reasoning; the endorsements approach

P8713 **W. Syski**
On a certain probabilistic approximation method for reasoning with uncertainty

P8714 **L.G. Bouma & H.R. Walters**
Implementing algebraic specifications

P8801 **F.R. Burggraaff & E. van der Meulen**
ASF Specification of a B-tree of order 1

P8802 **J.C.M. Baeten & J.A. Bergstra**
Recursive process definitions with the state operator

P8803 **J.C.M. Baeten & F.W. Vaandrager**
Specification and verification of a circuit in ACP

P8804 **J.A. Bergstra & J.V. Tucker**
The inescapable stack: An exercise in algebraic specification with total functions

P8805 **J. Treur**
Completeness and definability in diagnostic expert systems

P8806 **J. Treur**
Generic inference processes and their interactions in complex diagnostic tasks; a logical description

P8807 **J.L.M. Vrancken**
The implementation of process algebra specifications in POOL-T

P8808 **J.A. Bergstra**
A mode transfer operator in process algebra

P8808b **J.A. Bergstra**
A mode transfer operator in process algebra *(04/1989)*

P8809 **J. Treur**
Metakennis en meta-inferenties in expertsystemen

P8810 **R.E. Swart**
The BCA Bull Course Adviser

P8811 **F. Wiedijk**
Voorlopig rapport over de specificatie-taal Perspect

P8812 **J. Treur**
On the use of reflection principles in modelling complex reasoning

P8813 **J. Treur**
Reasoning about partial models, actions and plans

P8814 **S. Mauw & G.J. Veltink**
A process specification formalism

P8815 **J.A. Bergstra**
Process algebra for synchronous communication and observation

P8815b **J.A. Bergstra**
Process algebra for synchronous communication and observation *(03/1989)*

P8816 **J.A. Bergstra**
ACP with signals

P8817 **J. Treur**
Heuristic reasoning with incomplete knowledge

P8818 **H.K. Faber**
Uitleg en kennisrepresentatie in expertsystemen

P8819 **H.K. Faber**
Strategische uitleg in een medisch expertsysteem

P8820 **J. Treur**
A logical framework for design processes

P8821 **J.C.M. Baeten & F.W. Vaandrager**
Specification and verification of a circuit in ACP (revised version)

P8822 **J.L.M. Vrancken**
The algebraic specification of semicomputable data types (revised version)

P8823 **J.A. Bergstra, J. Heering & P. Klint**
Module Algebra (revised version)

P8824 **J.A. Bergstra & J.W. Klop**
Process theory based on bisimulation semantics

P8825 **H.W. Lenferink**
Local control of inference by means of meta-rules

P8826 **J. Treur**
Design of modular and interactive knowledge-based systems

P8901 **S. Mauw & G.J. Veltink**
An introduction to PSF

P8902 **R.P. Ogilvie**
Knowledge engineering vs. software engineering

P8903 **J.A. Bergstra**
A representation of addition and deletion lists using module algebra

P8904 **J.A. Bergstra, S. Mauw & F. Wiedijk**
Uniform algebraic specifications of finite sorts with equality

P8905 **J.A. Bergstra & J.W. Klop**
BMACP

P8906 **J.C.M. Baeten, J.A. Bergstra, S. Mauw & G.J. Veltink**
A process specification formalism based on static COLD

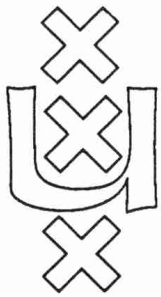
P8907 **J.A. Bergstra**
Kerninformatica en toekomst

P8908 **S. Mauw & F. Wiedijk**
Specification of the transit node in PSF_d

P8909 **J.A. Bergstra**
Algebra of states and transitions

Reports of the Expert Systems Section

PE8901 **G. Dam**
A design for the Strategic Interactive MEdical Expert System SIMEDES



University of Amsterdam
Faculty of Mathematics and Computer Sciences
Programming Research Group

Algebra of states and transitions

J.A. Bergstra

Report P8909 June 1989

Jan Bergstra

Programming Research Group
Faculty of Mathematics and Computer Sciences
University of Amsterdam

NIKHEF K A109
Kruislaan 409
1098 SJ Amsterdam
The Netherlands

P.O. Box 41882
1009 DB Amsterdam
The Netherlands

tel. +31 20 5922013

Department of Philosophy
University at Utrecht

Transitorium II kr. 1026
Heidelberglaan 2
3584 CS Utrecht
The Netherlands

tel. +31 30 532761

Universiteit van Amsterdam, 1989