Logic and Constraint Programming

Maude

Prof. Fabrizio Fornari

June 10, 2022

Maude

Maude is a language and environment based on **rewriting logic**.

It is used **to model systems and the actions within those systems**.

It can be used **to define executable formal models** of distributed systems, and provides analysis tool to formally analyze the models.

It can model almost anything, from the set of rational numbers to a biological system to the programming language Maude itself.

Maude and rewriting logic were both developed by José Meseguer and his research group at the Computer Science Laboratory at Stanford Research Institute (SRI) International. He leads the *Formal Methods and Declarative Languages Laboratory*

Maude Usage

Maude and its formal tool environment can be used in three, ways:

- as a declarative programming language;
- as an executable formal specification language; and
- as a formal verification system.

Rewriting Logic

In rewriting logic, the **data types** of the system are **defined algebraically by *equations***. In essence, **defining data types amounts to define functions in a recursive**, functional programming style.

The **dynamic behavior** of a system is then **defined by *rewrite rules*** which describe how a part of the state can change in one step.

Maude References

Maude publicly released in 1998, and is still under active development.

Maude official website
http://maude.cs.illinois.edu/w/index.php/The_Maude_System

Maude github repository
https://github.com/SRI-CSL/Maude/releases/tag/3.2.1

Maude manual
http://maude.cs.illinois.edu/w/images/6/65/Maude-3.2.1-manual.pdf

Maude Tutorial
http://maude.cs.illinois.edu/w/images/6/63/Maude-primer.pdf

Maude binaries are provided for selected architectures and operating systems, including Linux and macOS

Basic-list example

```
fmod BASIC-LIST is
    sorts List Elt .
    subsort Elt < List .
    op nil : -> List [ctor] .
    op __ : List List -> List [ctor] .
    vars E1 E2 : Elt .
    vars L1 L2 : List .
endfm
```

***A bare-minimum list needs a sort List and an element sort, or as commonly represented, Elt, a constant for empty lists, and a concatenation operator.

***Elt declared subsort because an element of a list is really just a special case of a list: a list of size one (called a singleton).

***double underscore concatenation operator This also means we can form a list by concatenating two elements, because an operator that takes List will also take a subsort of List. This way, the Maude compiler understands not only “L1 L2” but also “E1 E2” and “E1 L1.”

***finally, the constant constructor nil is a common way to name a list of size zero.

Boolean

```
set include BOOL off .

fmod BOOLEAN is
   sort Bool .
   op true : -> Bool .
   op false : -> Bool .
   op not_ : Bool -> Bool .
   op _and_ : Bool Bool -> Bool .
   op _or_ : Bool Bool -> Bool .
   var A : Bool .
   eq not true = false .
   eq not false = true .
   eq true and A = A .
   eq false and A = false .
   eq true or A = true .
   eq false or A = A .
endfm

set include BOOL on .
```

red true and not (false or not true) .

***(
reduce in BOOLEAN : true and not (false or not true) .
rewrites: 4 in -10ms cpu (0ms real) (~ rewrites/second)
result Bool: true
)

Modules

The module is the key concept of Maude.

It is essentially a set of definitions. These define a collection of operations and how they interact (they define an algebra). An algebra is a set of sets and the operations on them.

In Maude, a module will provide a collection of sorts and a collection of operations on these sorts, as well as the information necessary to reduce and rewrite expressions that the user inputs into the Maude environment.

fmod NAME is ... endfm *functional module*
mod NAME is ... endm *system module*

all the declarations and statements are in between the beginning of the module and the end of it.

Supplied Modules

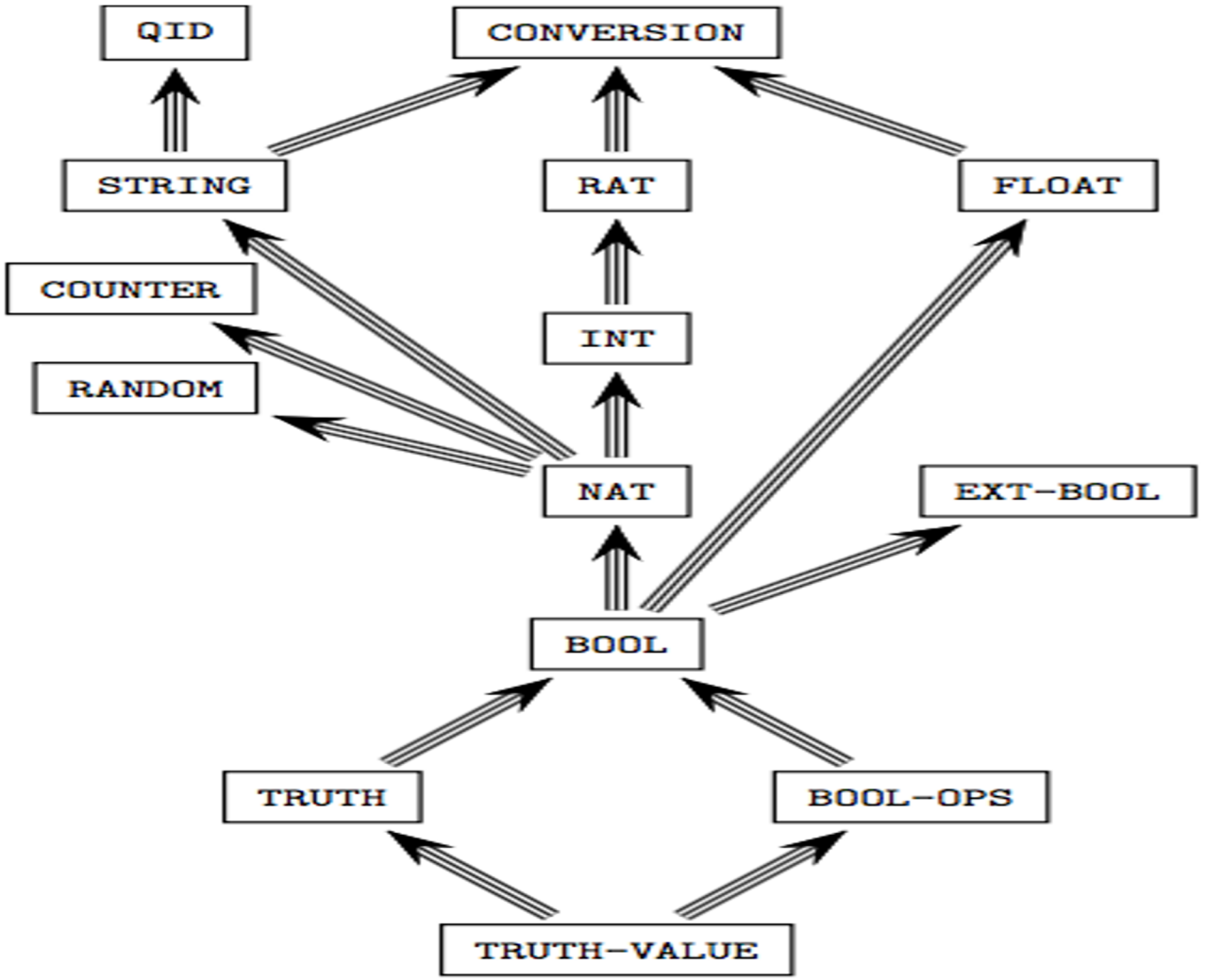
NAT Includes sort Nat, addition _+_, symmetric difference sd, multiplication _*_, quotient _quo_, modulo _rem_, and a host of other useful operations, and all the natural numbers written in normal, numeric notation.

INT, **FLOAT**, and **RAT** support integer, floating point, and rational numbers.

STRING module, while handles strings of characters and provides useful functions for searching and determining length. A String is any group of characters enclosed in quote marks, for example, "hello world!"

QID, or **Quoted IDentifier**. An apostrophe just before any word creates a quoted identifier, which can be used as a name.

Supplied Modules



Importing Modules

As in most programming languages, one can import a module from another.

protecting MODULENAME . ***Not modified anyway

including MODULENAME . ---Can change the sense in which the
---declarations were used.

(INT example)

extending MODULENAME . ---somewhere between these two extremes

junk - adding new ground terms (constructors and constants) to a module's sort(s).

confusion - redefining the already extant terms of the imported module.

Sorts

The first thing a specification needs to declare are the types of the data being defined and the corresponding operations.

A ***sort*** is a category for values. A sort can pretty much describe any type of value, including lists and stacks of other values.

subsorts, are further specific groups all belonging to the same sort.

sorts Rational Integer Positive Negative .

subsorts Positive Negative < Integer < Rational .

Membership Axioms

“Membership” simply refers to how certain terms are “members” of sorts. When we declare a variable, we declare it as a member of a sort using the colon, which one can think of a symbol for “is a member of.”

```
var N : Nat .
```

membership logic is at the bottom of pretty much every declaration in Maude.

```
op _+_ : Nat Nat -> Nat .
```

```
subsort NzNat < Nat .
```

Variables

A ***variable*** is an indefinite value for a sort. Just as *x* is a common variable for a number, one can declare a variable *x* as being of sort *number*.

Variables are never used as constants. Maude variables never have a definite value assigned to them. The only important use of variables is as placeholders, when defining operations through equations and rewrite laws.

```
var x : number .
vars c1 c2 c3 : color .
```

The expression var x : [number] . declares a variable for the kind of number as opposed to the sort.

Operations

You can think an operation as a pathway between sorts. Maude understands both prefix and mixfix notation for operations.

```
ops + * : Nat Nat -> Nat .
ops _ + _ _*_ : Nat Nat -> Nat .
```

Operation names may contain the special characters, such as the parentheses and brackets, commas, and periods.

Constant operations act as the equivalent of constants in other programming languages

```
op pi : -> Irrational .
ops red blue yellow : -> Color .
```

Overloaded operators

Maude allows overloaded operators; that is, we can define two different operators with the same name.

```
op _+_ : Integer Integer -> Integer .
op _+_ : Nat Nat -> Nat .              *** subsort overloading, requires (same flags)
op _+_ : Note Note -> Chord .          *** Ad-hoc overloading
op _+_ : Wrong Wrong -> Right .        *** Ad-hoc overloading
```

Peano numbers

Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function

The five Peano axioms are:

1. Zero is a natural number.
2. Every natural number has a successor in the natural numbers.
3. Zero is not the successor of any natural number.
4. If the successor of two natural numbers is the same, then the two original numbers are the same.
5. If a set contains zero and the successor of every number is in the set, then the set contains the natural numbers.

Peano numbers

Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function

```
fmod PEANO-NAT-EXTRA is
     sort Nat .
     op 0 : -> Nat .
     op s : Nat -> Nat .
     op _+_ : Nat Nat -> Nat .
     vars N M : Nat .
     eq 0 + N = N .
     eq s(M) + N = s(M + N) .
endfm
```

The Maude Environment

First, one must write the modules for the algebras to be used in the environment, either by typing them directly into the prompt, or by storing them in a file directory that the Maude environment can access, and then typing load MODULENAME into the prompt.

```
Maude> reduce s(0) + s(s(0)) .
result Nat: s(s(s(0)))
```

For those interested in seeing exactly how an expression is reduced, step by step, by the Maude interpreter, type:

```
Maude> set trace on .
```

Peano numbers

Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function

```
fmod PEANO-NAT-EXTRA is
     sort Nat .
     op 0 : -> Nat .
     op s : Nat -> Nat .
     op _+_ : Nat Nat -> Nat .
     vars N M : Nat .
     eq 0 + N = N .
     eq s(M) + N = s(M + N) .
endfm
```

```
red s(0) + s(s(s(s(s(0))) + s(s(0)))) .
```

```
***(
reduce in PEANO-NAT-EXTRA : s(0) + s(s(s(s(s(0))) + s(s(0)))) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(s(0)))))))
)
```

Equation Strategy

The first and most important strategy is recursion. Nearly every set of equations is defined with some level of recursion in mind.

```
fmod PEANO-NAT-MULT is
     protecting PEANO-NAT-EXTRA .
     op _*_ : Nat Nat -> Nat .
     vars M N : Nat .
     eq N * 0 = 0 .
     eq N * s(M) = N + (N * M) .
endfm
```

$$
\begin{aligned}
3*5 &= 3+(3*4)\\
&= 3+(3+(3*3))\\
&= 3+(3+(3+(3*2)))\\
&= 3+(3+(3+(3+(3*1))))\\
&= 3+(3+(3+(3+(3+(3*0)))))\\
&= 3+(3+(3+(3+(3+(0)))))
\end{aligned}
$$

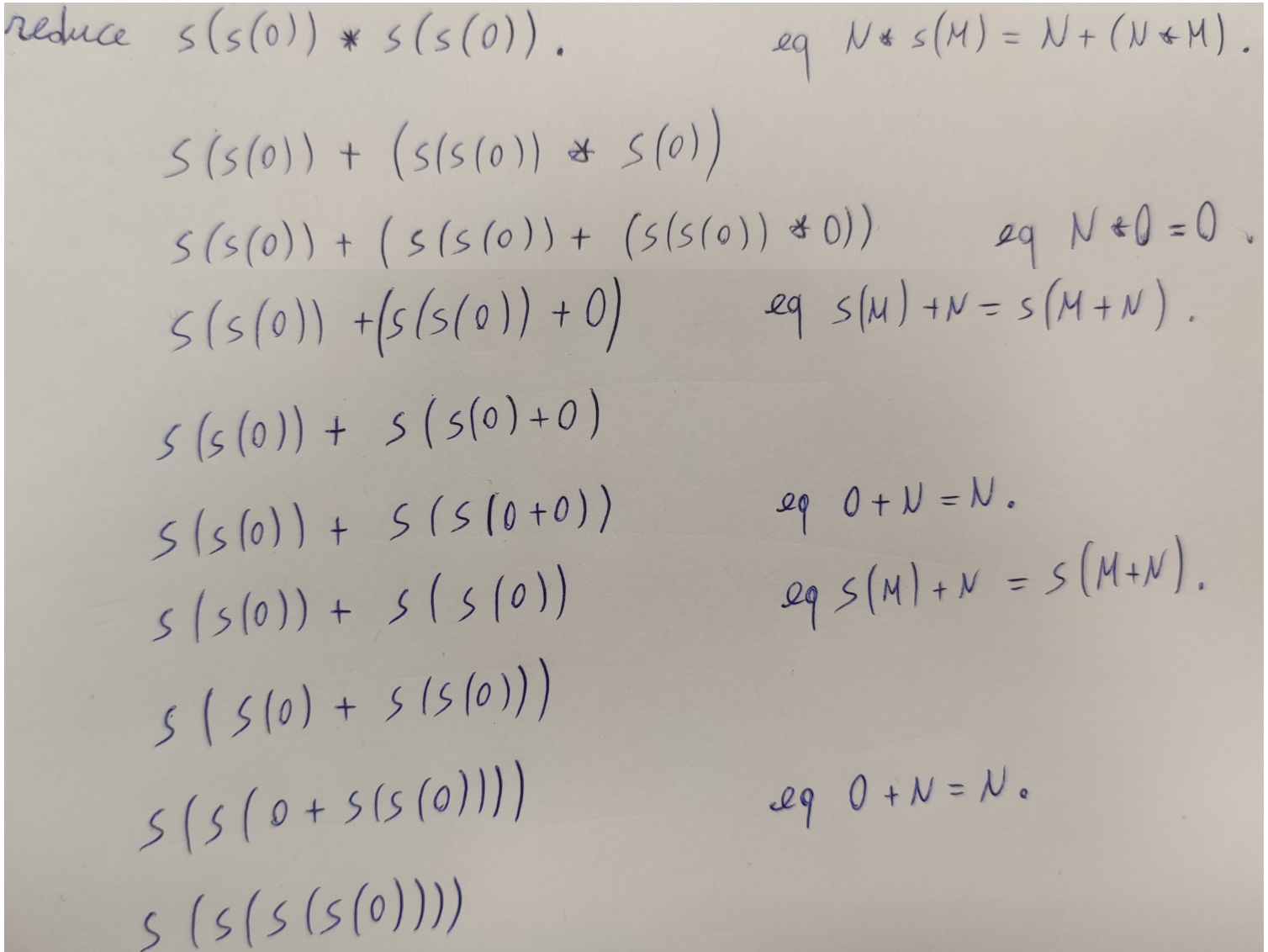
The Maude interpreter evaluates an expression in an equational Maude program by applying the **equations** "from left to right" until no equation can be applied, thereby computing the normal form (or "value") of the expression.

```
Maude> reduce s(s(s(0))) * s(s(s(s(s(0))))) .
reduce in PEANO-NAT-MULT : s(s(s(0))) * s(s(s(s(s(0))))) .
rewrites: 26 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))
```

Equation Strategy

reduce s(s(0)) * s(s(0)) .

Try to reduce it manually applying the equations. It corresponds to finding the execution trace



Maude> **reduce s(s(0)) * s(s(0)) .**
reduce in PEANO-NAT-MULT : s(s(0)) * s(s(0)) .
rewrites: **9** in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(0))))

Equation Strategy

```
fmod FIBONACCI is
   protecting NAT .
   op fibo : Nat -> Nat .

   var N : Nat .
   eq fibo(0) = 0 .
   eq fibo(1) = 1 .
   eq fibo( s s N) = fibo(N) + fibo(s N) .
endfm
```

Maude> reduce fibo(35) .
reduce in FIBONACCI : fibo(35) .
rewrites: 44791054 in 6392ms cpu (7054ms real) (7007361 rewrites/second)
result NzNat: 9227465

Maude> reduce fibo(50) .

Equation Strategy

```
fmod FIBONACCI is
   protecting NAT .
   op fibo : Nat -> Nat [memo] .

   var N : Nat .
   eq fibo(0) = 0 .
   eq fibo(1) = 1 .
   eq fibo( s s N) = fibo(N) + fibo(s N) .
endfm
```

Whenever an application will perform an operation many times, it may be useful to give that operator the **memo** attribute.

```
Maude> reduce in FIBONACCI : fibo(35) .
rewrites: 103 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 9227465
```

```
Maude> reduce fibo(50) .
reduce in FIBONACCI : fibo(50) .
rewrites: 46 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 12586269025
```

Memo

Without Memo

```
Maude> reduce fibo(3) .
reduce in FIBONACCI : fibo(3) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2
```

With Memo

```
Maude> reduce fibo(3) .
reduce in FIBONACCI : fibo(3) .
rewrites: 4 in 0ms cpu (0ms real) (~rewrites/second)
result NzNat: 2
```

Look at their trace. With the command: set trace on .

Memo

reduce fibo(3) .
without and with memo

```
Maude> reduce fibo(3) .
reduce in FIBONACCI : fibo(3) .
********** equation
eq fibo(s_^2(N)) = fibo(N) + fibo(s N) .
N --> 1
fibo(3)
--->
fibo(1) + fibo(s 1)
********** equation
eq fibo(1) = 1 .
empty substitution
fibo(1)
--->
1
********** equation
eq fibo(s_^2(N)) = fibo(N) + fibo(s N) .
N --> 0
fibo(2)
--->
fibo(0) + fibo(1)
********** equation
eq fibo(0) = 0 .
empty substitution
fibo(0)
--->
0
********** equation
eq fibo(1) = 1 .
empty substitution
fibo(1)
--->
1
********** equation
(built-in equation for symbol _+_)
0 + 1
--->
1
********** equation
(built-in equation for symbol _+_)
1 + 1
--->
2
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2
```

```
Maude> reduce fibo(3) .
reduce in FIBONACCI : fibo(3) .
********** equation
eq fibo(s_^2(N)) = fibo(N) + fibo(s N) .
N --> 1
fibo(3)
--->
fibo(1) + fibo(s 1)
********** equation
(memo table lookup for symbol fibo)
fibo(1)
--->
1
********** equation
(memo table lookup for symbol fibo)
fibo(2)
--->
1
********** equation
(built-in equation for symbol _+_)
1 + 1
--->
2
rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2
```

Operator and Statement Attributes

memo flag. When Maude comes across an operation with memo among its attributes, it "memorizes" the reduced form of any expression with that operator at the top (that is, if the expression were written in prefix mode, the outermost operator).

Whenever such an expression appears, Maude refers to its memorization table and produces the reduced form much quicker than if it had to continually apply reduction equations over and over again.

This is useful when we write programs where the same expression (important: the same expression, not just the same operator) pops up thousands and thousands of times, such as highly recursive number theory problems.

Operator Attributes

Three key flags ***assoc***, ***comm***, and ***id***, impose the equational attributes of ***associativity***, ***commutativity***, and ***identity*** on any binary operator.

op __ : List List -> List [ctor assoc id: nil] .

Associativity just means that the list formed by “(L1 L2) L3” is the same as the list formed by “L1 (L2 L3)” is the same as the list formed by “L1 L2 L3.”

Identity property of addition n+0=n. The identity property of lists is that “L1 nil” is the same as just “L1.” In other words, concatenating a list with an empty list reduces to the original list. One may also define identities for the left and right arguments *left id:* and *right id:* .

op __ : Set Set -> Set [ctor assoc comm id: none] .

Commutativity: for sets (not lists) order does not matter, that “S1 S2 S3” is the same as “S1 S3 S2” is the same as “S2 S1 S3” etc.

Operator Precedence

When there is any fear of ambiguous expressions the user may declare precedence for the operator using the flag prec and a natural number.

expression 3 + 3 * 3 is ambiguous since (3 + 3) * 3 and 3 + (3 * 3) are valid parses

```
op _+_ : Num Num -> Num [prec 35] .
op _*_ : Num Num -> Num [prec 25] .
```

gathering pattern

X + Y makes no use of the gather flag

X + Y + Z, involves the addition of a variable to another addition: either_+_(X,_+_(Y,Z)) or _+_(_+_(X,Y),Z) If,say, we were to declare the _+_ operator with the flag gather (E e), then only the latter is possible.

E = smaller or equal precedence & = any precedence value

e = strictly lower precedence

Other flags

idem - which denotes the equational attribute idempotency, the property that repeated elements are discarded. assoc and idem flags may not be used together in Maude.

A useful flag for unary operators that must be iterated over and over again for example, the s_ operator where ‘five’ takes the monstrous form of s s s s s 0 – is the ***iter*** flag, which allows such chains of iteration to be expressed as a single instance of the operator, raised to the number of iterations.

We may write s s s s s 0 as s_^5(0).

owise attribute

It is rather useful in cases where one side of the condition is much easier to express than the other:

```
eq suicideking?( K of Hearts ) = true .
eq suicideking?( C:Card ) = false [owise] .
```

Other attributes are:

```
[ label metadata nonexec print ditto frozen ]
```

Peano numbers

Peano numbers are a simple way of representing the natural numbers using only a zero value and a successor function

```
fmod PEANO-NAT-EXTRA is
     sort Nat .
     op 0 : -> Nat .
     op s : Nat -> Nat .
     op _+_ : Nat Nat -> Nat [comm] .
     vars N M : Nat .
     eq 0 + N = N .
     eq s(M) + N = s(M + N) .
endfm
```

```
red s(0) + s(s(s(s(s(0))) + s(s(0)))) .
```

```
***(
reduce in PEANO-NAT-EXTRA : s(0) + s(s(s(s(s(0))) + s(s(0)))) .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(s(s(0))))))))
)
```

Equation Strategy

The first and most important strategy is recursion. Nearly every set of equations is defined with some level of recursion in mind.

```
fmod PEANO-NAT-MULT is
     protecting PEANO-NAT-EXTRA .
     op _*_ : Nat Nat -> Nat [comm] .
     vars M N : Nat .
     eq N * 0 = 0 .
     eq N * s(M) = N + (N * M) .
endfm
```

$$
\begin{aligned}
3*5 &= 3+(3*4)\\
&= 3+(3+(3*3))\\
&= 3+(3+(3+(3*2)))\\
&= 3+(3+(3+(3+(3*1))))\\
&= 3+(3+(3+(3+(3+(3*0)))))\\
&= 3+(3+(3+(3+(3+(0)))))
\end{aligned}
$$

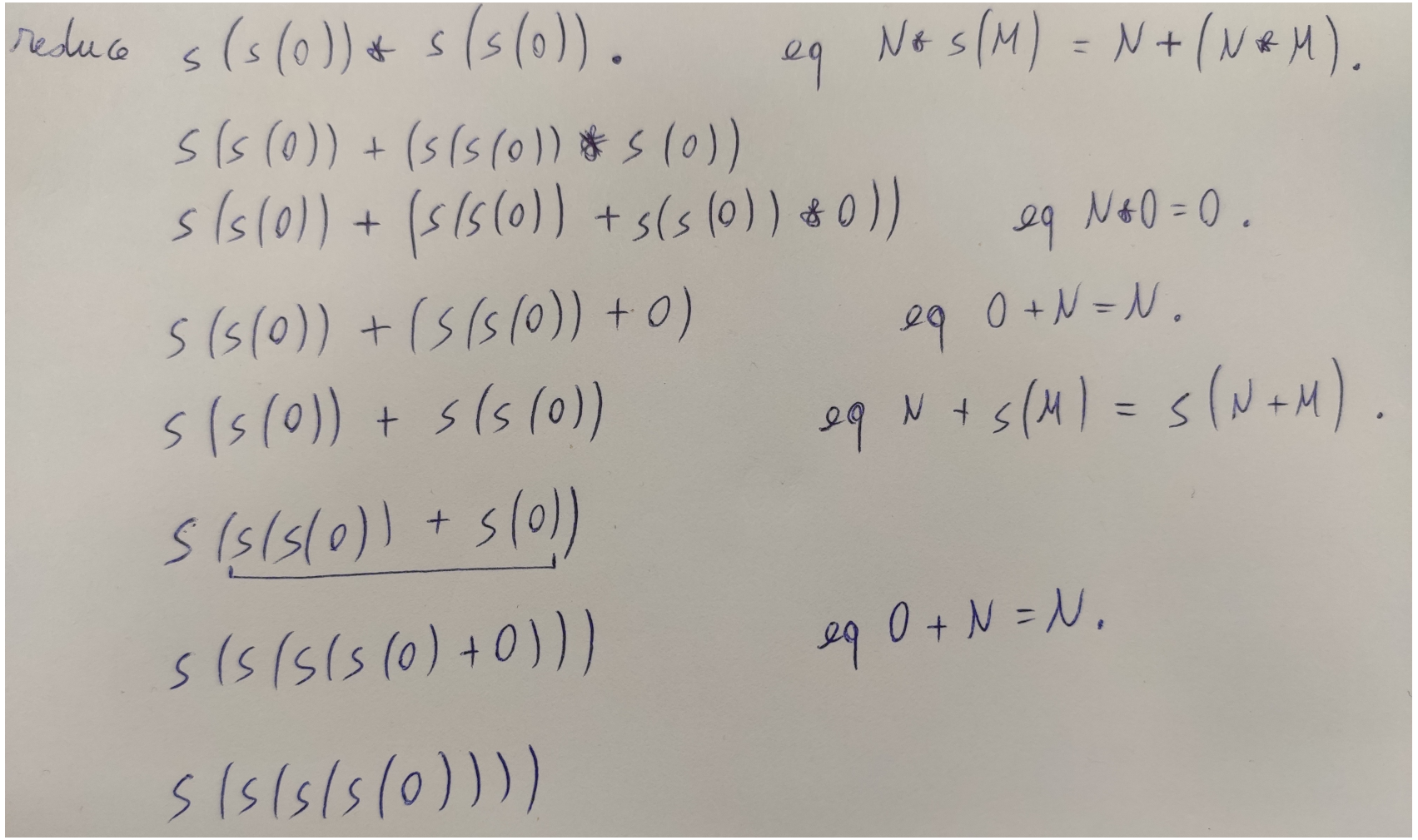
The Maude interpreter evaluates an expression in an equational Maude program by applying the **equations** “from left to right” until no equation can be applied, thereby computing the normal form (or “value”) of the expression.

```
Maude> reduce s(s(s(0))) * s(s(s(s(s(0))))) .
reduce in PEANO-NAT-MULT : s(s(s(0))) * s(s(s(s(s(0))))) .
rewrites: 26 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))
```

Equation Strategy

reduce s(s(0)) * s(s(0)) .

Analyze the result and write the execution trace as we did before



reduce s(s(0)) * s(s(0)) .
rewrites: 7 in 0ms cpu (0ms real) (~ rewrites/second)
result Nat: s(s(s(s(0))))

Unconditional Equations

The equations we have seen until now are called **Unconditional Equations**

eq ⟨Term-1⟩ = ⟨Term-2⟩ [⟨StatementAttributes⟩] .

E.g., we can have equations axiomatizing the addition operation, where we distinguish two cases for the second argument, according to whether it is zero or not:

```
vars N M : Nat .
eq N + zero = N .
eq N + s M = s (N + M) .
```

Unconditional Membership

We can also have **Unconditional Membership**

mb ⟨Term⟩ : ⟨Sort⟩ [⟨StatementAttributes⟩] .

Consider the module 3*NAT with the basic Peano number declarations as in the Peano-Nat-Extra module and a new sort 3*Nat.

```
fmod 3*NAT is
   sort Zero Nat .
   subsort Zero < Nat .
   op zero : -> Zero .
   op s_ : Nat -> Nat .
   sort 3*Nat .
   subsorts Zero < 3*Nat < Nat .
   var M3 : 3*Nat .
   mb(s s s M3) : 3*Nat .
endfm
```

Conditional equations and memberships

Conditional Membership

cmb ⟨Term⟩ : ⟨Sort⟩
if ⟨EqCondition-1⟩ /\ ... /\ ⟨EqCondition-k⟩ [⟨StatementAttributes⟩] .

Conditional Equations

ceq ⟨Term-1⟩ = ⟨Term-2⟩
if ⟨EqCondition-1⟩ /\ ... /\ ⟨EqCondition-k⟩ [⟨StatementAttributes⟩] .

An extract of the PATH module

```
sort Path .
subsort Edge < Path .
op _;_ : [Path] [Path] -> [Path]

var E:Edge.
vars P Q R S : Path .
cmb E ; P : Path if target(E) = source(P) .
```

The conditional membership axiom (introduced by the keyword cmb), in this example, states that an edge concatenated with a path is also a path when the target node of the edge coincides with the source node of the path

Conditional equations are equations that depend on a Boolean statement.

```
ceq different?( N , M ) = true if N =/= M .
ceq bothzero?( N , M ) = true if N == M /\ M == 0 .
ceq N - M = 0 if M > N .
```

Rewrite Laws

The real power of Maude is about transitions that occur within and between structures. These transitions are mapped out in rewrite laws.

Rewriting logic consists of two key ideas: **states** and **transitions**.

States are situations that, alone, are static, and **transitions** are the transformations that map one state to another. **Defined in mod (system module).**

A rewrite law declares the relationship between the states and the transitions between them.

```
rl [<Label>] : <Term-1> => <Term-2> [<StatementAttributes>] .
```

```
rl [raincloud] : sunnyday => rainyday .
```
----irreversibility; “one-way equations”

Rewrite Laws

```
mod CLIMATE is
sort weathercondition .
op sunnyday : -> weathercondition .
op rainyday : -> weathercondition .
rl [raincloud] : sunnyday => rainyday .
endm

rew [100] sunnyday .
```

```
Maude> rew [100] sunnyday .

rewrite [100] in CLIMATE : sunnyday .
********** rule
rl sunnyday => rainyday [label raincloud] .
empty substitution
sunnyday
--->
rainyday
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result weathercondition: rainyday
```

Example

```
mod CIGARETTES is
    sort State .
    op c : -> State [ctor] . *** cigarette
    op b : -> State [ctor] . *** butt
    op __ : State State -> State [ctor assoc comm] .
    rl [smoke] : c => b .
    rl [makenew] : b b b b => c .
endm
rew [100] c c c c c c c c c c c c c c c c .
```

Example

```
set trace on .
rew [100] c c c .
```

```
Maude> rew [100] c c c .
rewrite [100] in CIGARETTES : c c c .
********** rule
rl c => b [label smoke] .
empty substitution
c
--->
b
********** rule
rl c => b [label smoke] .
empty substitution
c
--->
b
********** rule
rl c => b [label smoke] .
empty substitution
c
--->
b
rewrites: 3 in 0ms cpu (0ms real) (~
rewrites/second)
result State: b b b
```

Example

```
mod COUNTING-CIGARETTES is
    protecting NAT .
    sort State .
    op c : Nat -> State [ctor] .
    op b : Nat -> State [ctor] .
    op __ : State State -> State [ctor assoc comm] .
    vars W X Y Z : Nat .
    rl [smoke] : c(X) => b(X + 1) .
    rl [makenew] : b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .
endm
rew c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0) c(0)
c(0) c(0) c(0) c(0) c(0) .
```

we see rewriting laws drawing transitions from complex states to simpler states, and *this does not have to be the case*.

Search command

Maude provides a search command which searches through all behaviors from a given initial state and returns all—or a user-given number of—states which can be reached from the initial state and which satisfy the given search condition. The search may be restricted to analyze all behaviors up to n rewrite steps.

The basic forms of the search command are

```
search t0 arrow pattern .

search t0 arrow pattern such that cond .
```

A term t satisfies the search condition if pattern matches t and cond holds for the matching substitution.

Search command

=>1: states which can be reached in exactly one step from the initial state t0;

=>*: states reachable in zero or more steps;

=>+: states reachable in one or more steps; and

=>!: states that cannot be further rewritten.

Arcade-Crane

```
mod ARCADE-CRANE is
protecting QID .
sorts ToyID State .
subsort Qid < ToyID .

op floor : ToyID -> State [ctor] .
op on : ToyID ToyID -> State [ctor] .
op clear : ToyID -> State [ctor] .
op hold : ToyID -> State [ctor] .
op empty : -> State [ctor] .
op 1 : -> State [ctor] .
*** this is the identity State; it's just good to have one.
op _&_ : State State -> State [ctor assoc comm id: 1] .
vars X Y : ToyID .

rl [pickup] : empty & clear(X) & floor(X) => hold(X) .
rl [putdown] : hold(X) => empty & clear(X) & floor(X) .
rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
rl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) .
endm
```

The search command searches for paths of rewrite laws between a beginning and end state supplied by the user.

Maude> frew [1] empty & floor('mothergoose) & on('teddybear,'mothergoose) & on('soccerball,'teddybear) & clear('soccerball) & floor('dragondude) & clear('dragondude) .

Maude> continue 1 .

The search command searches for paths of rewrite laws between a beginning and end state supplied by the user.

search in ARCADE-CRANE : empty & floor('mothergoose) &
on('teddybear, 'mothergoose) & on('soccerball, 'teddybear) &
clear('soccerball) & floor('dragondude) & clear('dragondude) =>* S:State .

We can see all the reachable states

Show Path

From Box A state, we could switch Teddy Bear and Mother Goose so that instead of on('teddybear, 'mothergoose) we have on('mothergoose, 'teddybear).

search in ARCADE-CRANE : empty & floor('mothergoose) & on('teddybear, 'mothergoose) & on('soccerball, 'teddybear) & clear('soccerball) & floor('dragondude) & clear('dragondude) =>+ empty & floor('teddybear) & on ('mothergoose,'teddybear) & on('soccerball,'mothergoose) & clear('soccerball) & floor('dragondude) & clear('dragondude) .

```
Maude> show path 69 .
   state 0, State: empty & ...
   ===[ rl empty & clear(X) & on(X, Y) => clear(Y) & hold(X) [label

        unstack] . ]===>
   state 2, State: floor(‘mothergoose) & ...
   ===[... [label stack] ...
   state 4, State: ...
   ===[... [label unstack] ...
   ...
   state 69, State: empty & floor('teddybear) & floor('dragondude) &
                clear('soccerball) & clear('dragondude) & on('mothergoose,
          'teddybear) & on('soccerball, 'mothergoose)
```

Conditional Rewrite Laws

Imagine to extend the Arcade-Crane with the weight concept. For which an object can be hooked only if its weight is under a certain value

Conditional rewrite laws use the keyword crl, and the rest is a rewrite law with an if statement at the end.

```
crl [pickup] : empty & clear(X) & floor(X) => hold(X)
                                        if weight(X) < 10 .
```

In general, conditional rewrite laws are used when there's a condition that can't be easily expressed as a state.

```
crl [equation1] : a(X) => b(X - 1) if X > 0 .
crl [equation2] : a(X) => b(X - 1) if X > 0 == true .
crl [equation3] : a(X) => b(X - 1) if X > 0 = true .
crl [membership1] : a(X) => b(X - 1) if X :: NzNat .
crl [membership2] : a(X) => b(X - 1) if X : NzNat .
crl [pattern] : a(X) => b(X - 1) if s(N:Nat) := X .
```

Conditional Rewrite Laws

The condition of a rewrite rule may also be another rewrite rule.

```
crl [rewrite] : b(X) => c(X * 2) if a(X) => b(Y) .
```

It means that the rewrite law may be executed on b(X) only if a(X) could transition to a some state of b. Think of the => not so much as the rewrite symbol but as the search symbol: the idea is, the condition is fulfilled if it's possible to rewrite the left-hand side into the right-hand side.

Conditional Rewrite Laws

Equations and rewrite laws solve very different problems.

Equations are much better than rewrite laws at simplification; however, rewrite laws are better at expressing problems with just one level of simplicity.

Rewriting laws also can illustrate constitutional changes that equations can't; though an equation may simplify an expression, the expression is still mathematically equal to its predecessor.

In general, **equational logic creates a framework through which rewriting laws trace transitions**.

System modules often include equations; when they do**, they set up and define all the operations that become states, and then the rewrite laws deal with these states**. For example, in the slightly expanded arcade crane example, we would have to define the operation weight using an equation.

We also use equations to define numbers (creating our own notation, such as the Peano notation, or using the library module INT, which is also defined with equations) that can later be used in rewrite laws.

Equations are the nuts and bolts and gears and girders; rewrite laws create the machine.

Maude analysis and specifications

To analyze *all* possible behaviors *from a given initial state* one can use Maude's high-performance ***search* capabilities** to investigate whether certain (un)desired states can be reached from the initial state.

We typically have ***two* levels of specification**:

- a ***system specification*** describing which actions the system can perform,

- a ***requirement specification*** describing the requirements that the system should satisfy.

The requirements the system should satisfy in Maude can be defined as **linear temporal logic** formulas.

Maude's high-performance ***model checker*** can then be used to decide whether all possible behaviors from a given initial state satisfy the requirements, provided that the set of states reachable from the initial state is a **finite set**.

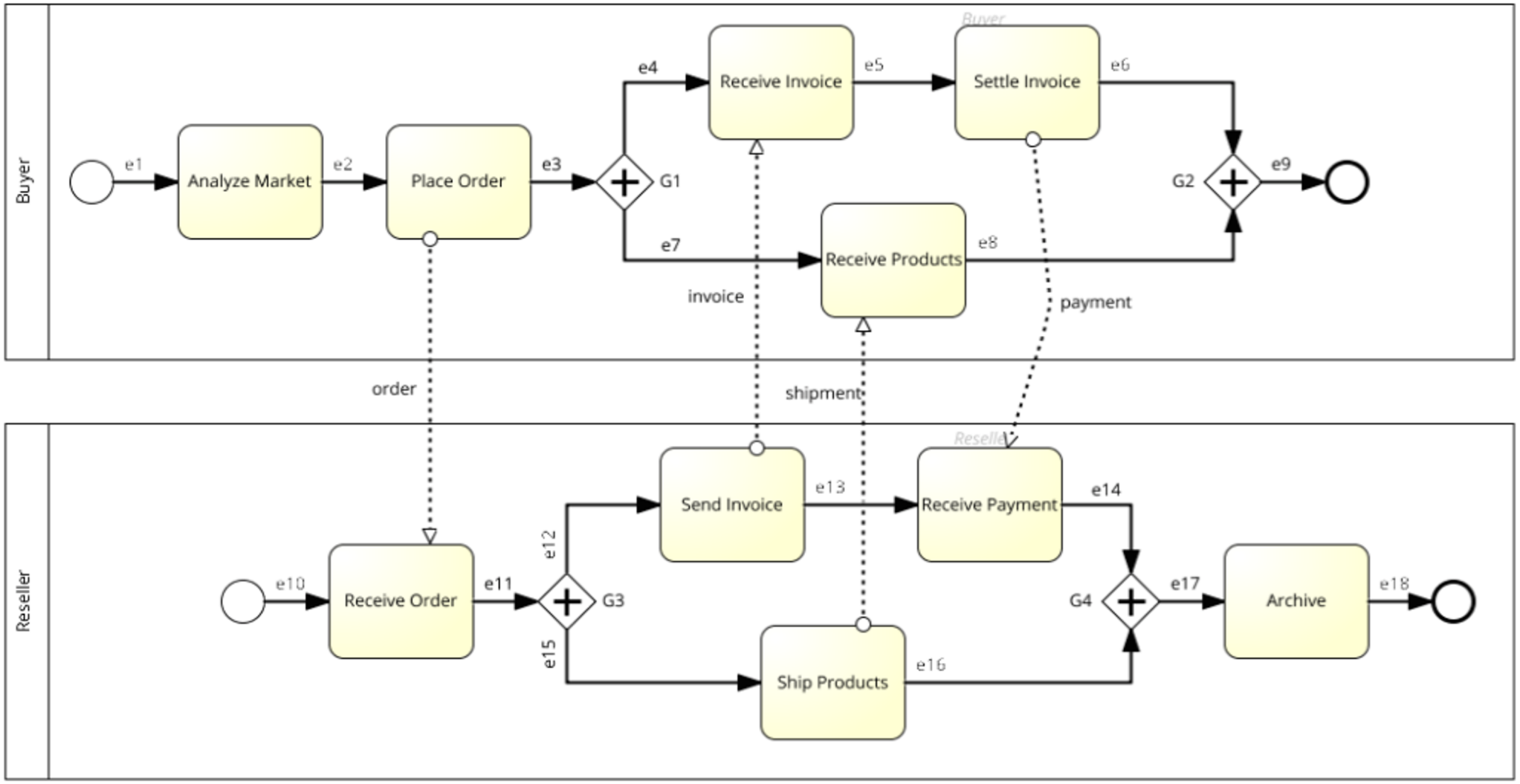
How I Applied Maude

Business Process

A Business Process is a collection of **activities** that are performed in coordination in an **organizational** and **technical environment**. These activities jointly realize a business goal taking one or more kinds of **inputs** and generating a product, or a service, as **output**. Each business process is enacted by a **single organization**, but it may interact with business processes performed by **other organizations**.

Revisited version of: *“Mathias Weske. Business process management : concepts, methods, technology. Springer, 2007”.*

Business Process



BPMN Operational Semantics

Corradini, F., Polini, A., Re, B., & Tiezzi, F. (2015, October). **An Operational Semantics of BPMN Collaboration***. In Pre-proceedings (p. 113).

FACS2015 Formal Aspects of Component Software http://facs2015.ic.uff.br/

- It is a **native semantics**, rather than a mapping
- It provides a compositional approach **based on Labelled Transition Systems**, which allows to use consolidated analysis techniques and related software tools
- It is suitable to model business processes with **arbitrary topology**, without imposing syntactic restrictions (Well-structured, SESE…)
- Not only Core elements but it takes into account **collaborations** and message exchange, which are often overlooked by other formalisations

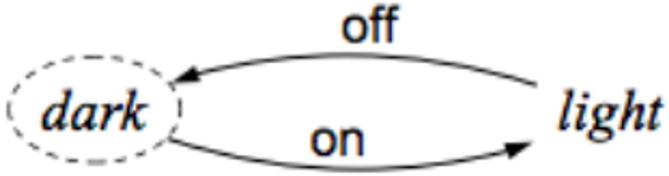
Labelled Transition System (LTS)

Definition: A **labelled transition system** (S, s_0, L, δ) consists of

- a set S of *states*
- an *initial state* $s_0 : S$
- a set L of *action labels*
- a *transition relation* $\delta : \mathbb{P}\,(S \times L \times S)$.

Example:

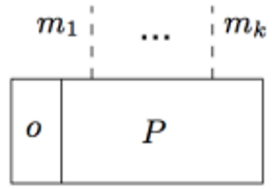
$$LightSwitch_1 = (\{dark, light\}, dark, \{\text{on}, \text{off}\}, \{(dark, \text{on}, light), (light, \text{off}, dark)\})$$



Backus Normal Form (BNF) Syntax

(Collaborations) C ::=

(Pool, $k \geqslant 0$)



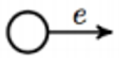
(Pool collection) | C_1 C_2

(Gateways) G ::=

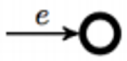
(XOR/AND/OR)

(Processes) P ::=

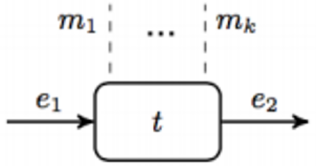
(Start event)



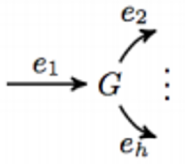
(End event) |



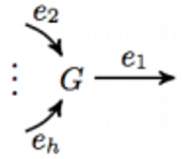
(Task, $k \geqslant 0$) |



(Split gateway, $h > 2$) |

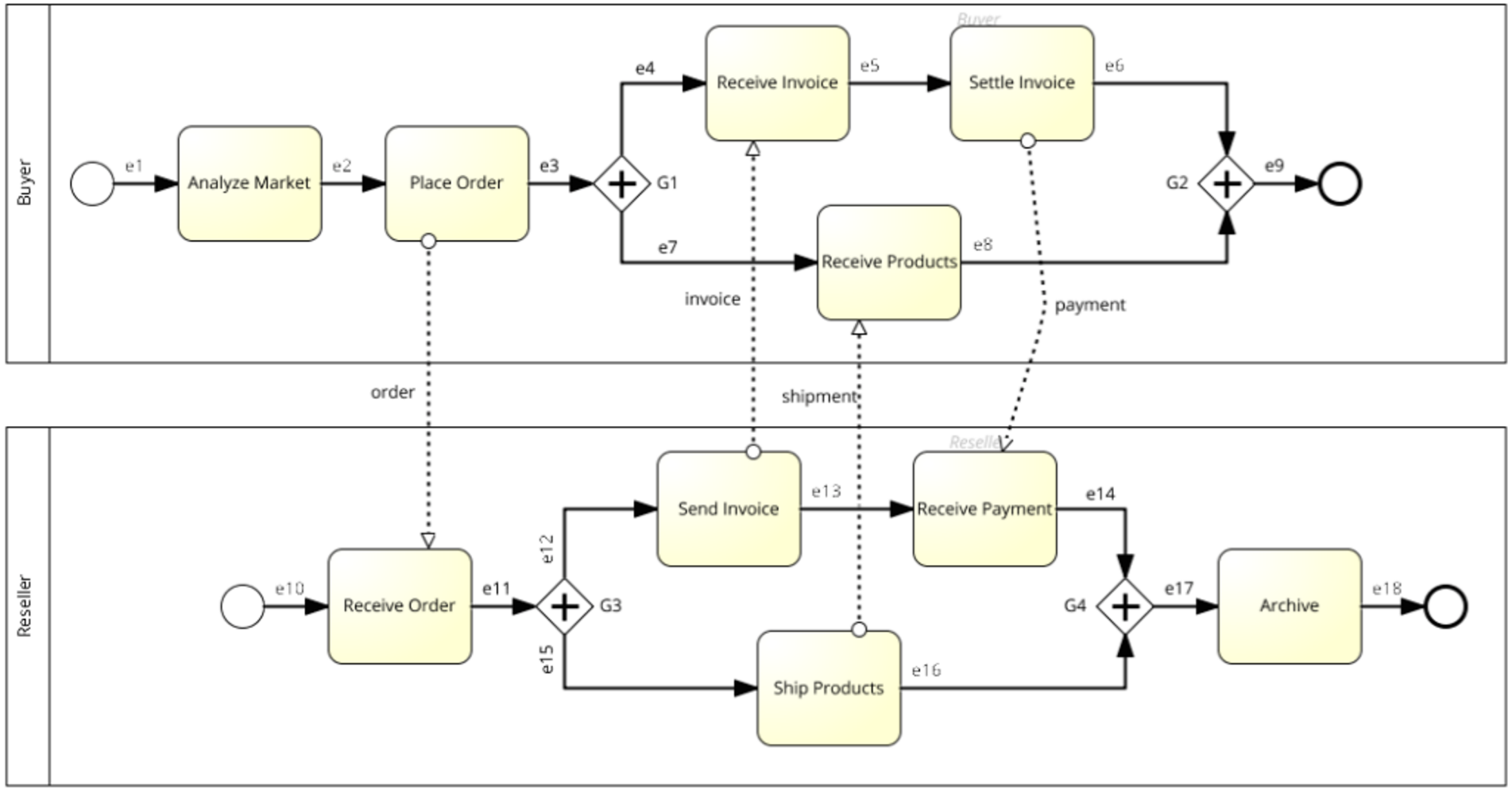


(Join gateway, $h > 2$) |

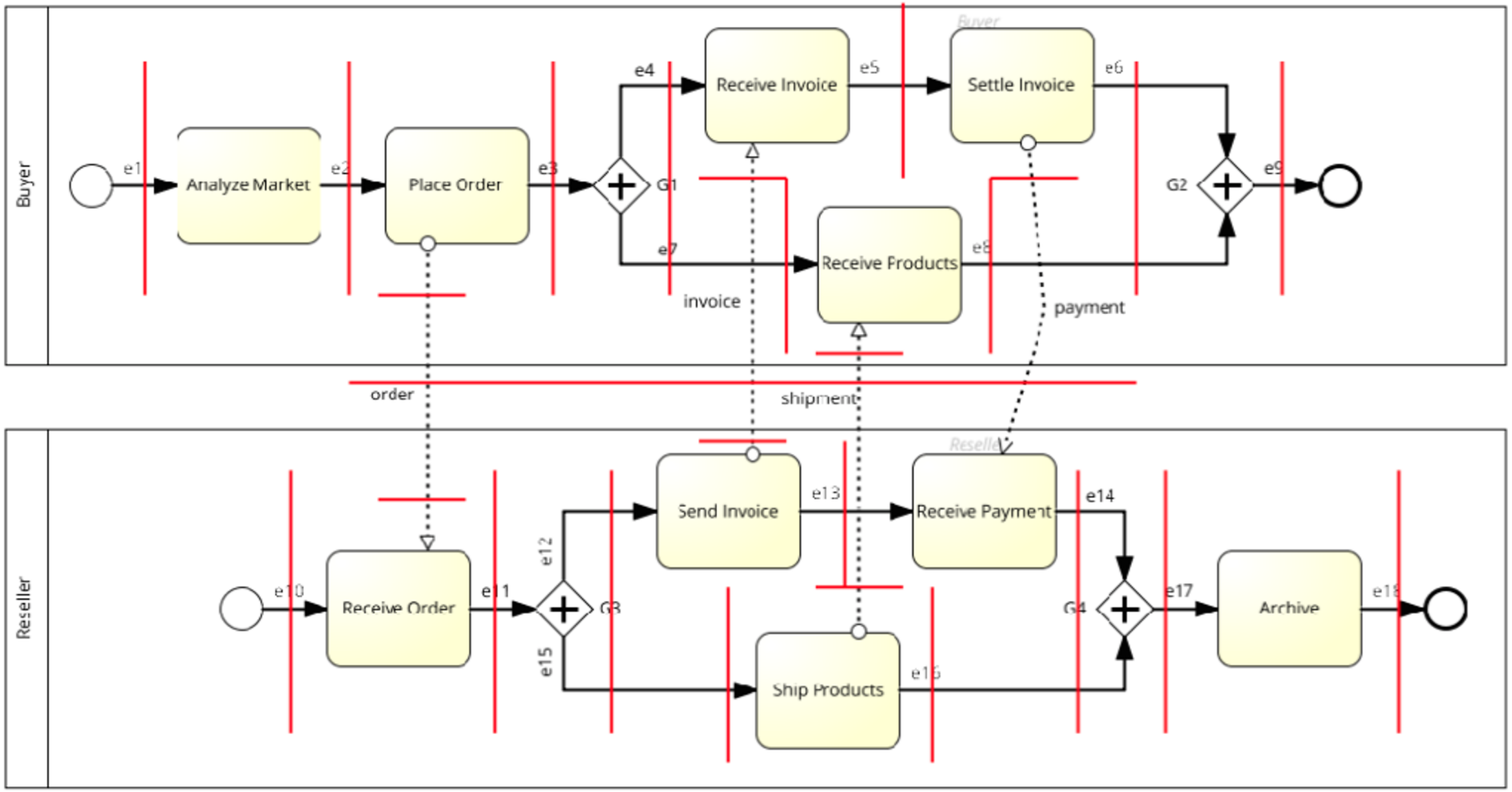


(Node collection) | P_1 P_2

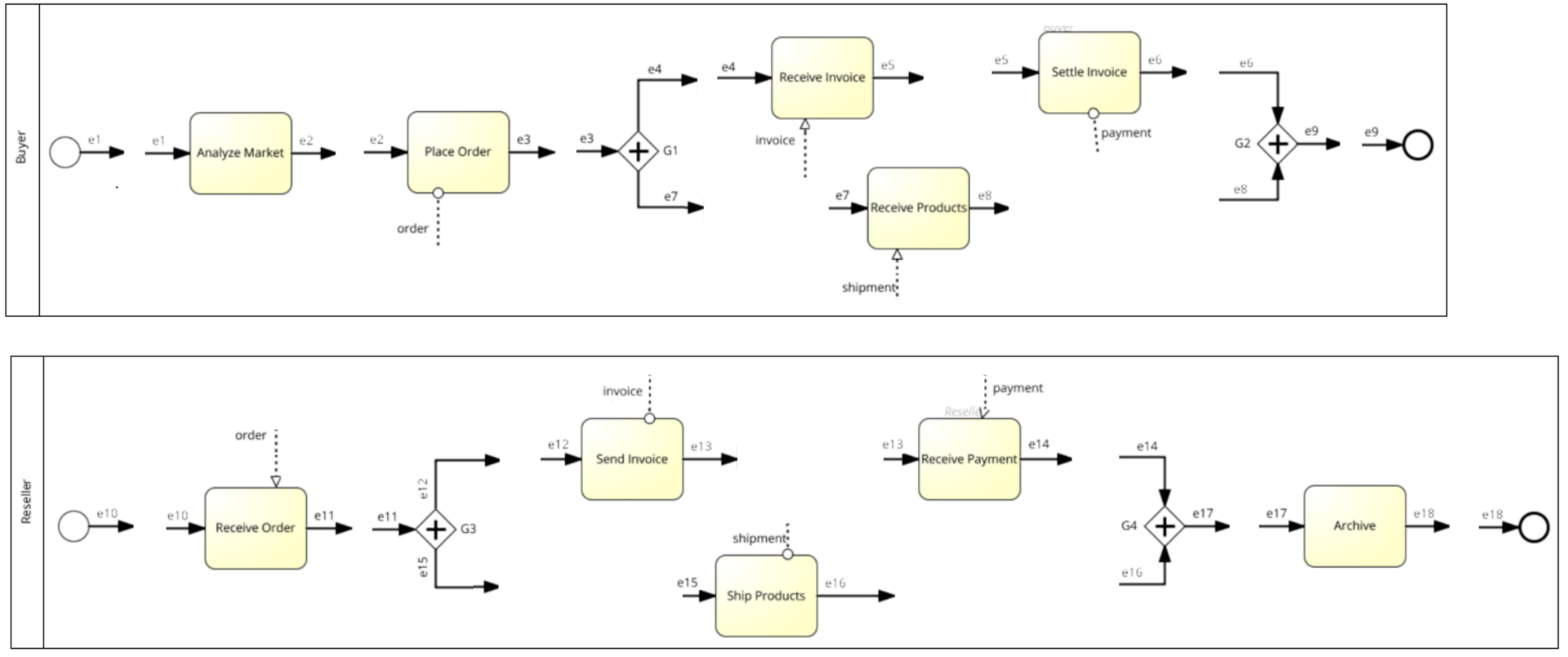
Buyer-Reseller Example



Buyer-Reseller “cuts”



Buyer-Reseller post “cuts”



Operational Semantics

The Operational semantics is based on a set of **inference rules** defined at different layers:

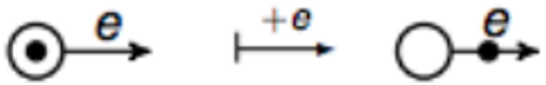
- Collaboration Layer
- Process Layer
  - Control Flow Constructs
  - Task Constructs
  - Node Collections

Marking

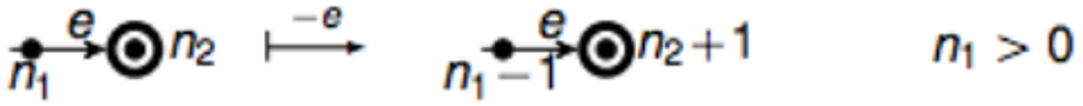
A marking is a distribution of tokens over pool, messages, edges, and process elements, that indicates message arrivals and the process nodes that are active or not in a given step of the execution.

Process Layer - Control Flow Constructs (1)

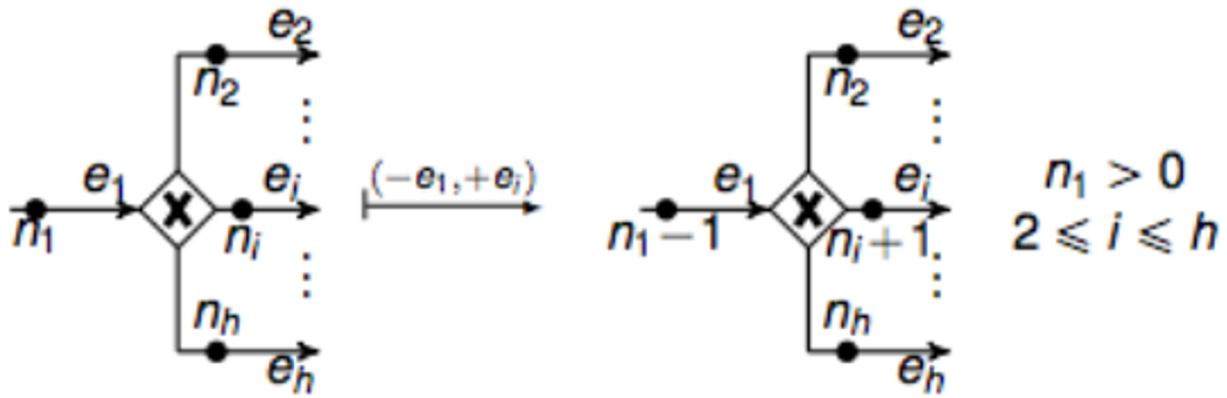
Start event:



End event:

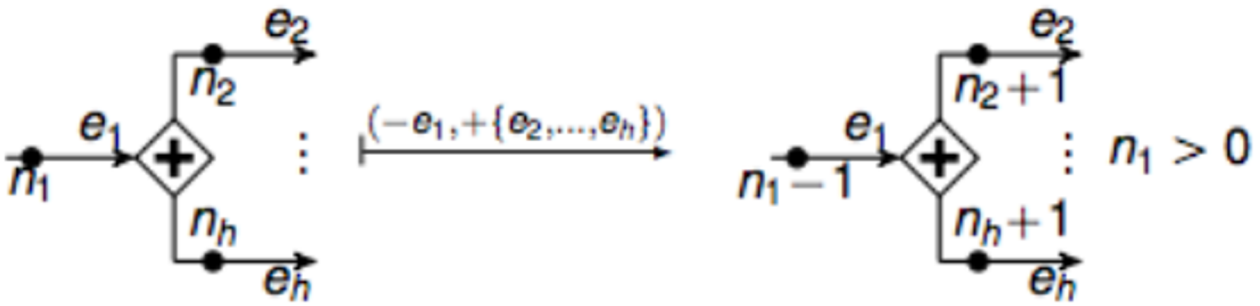


XOR Split:

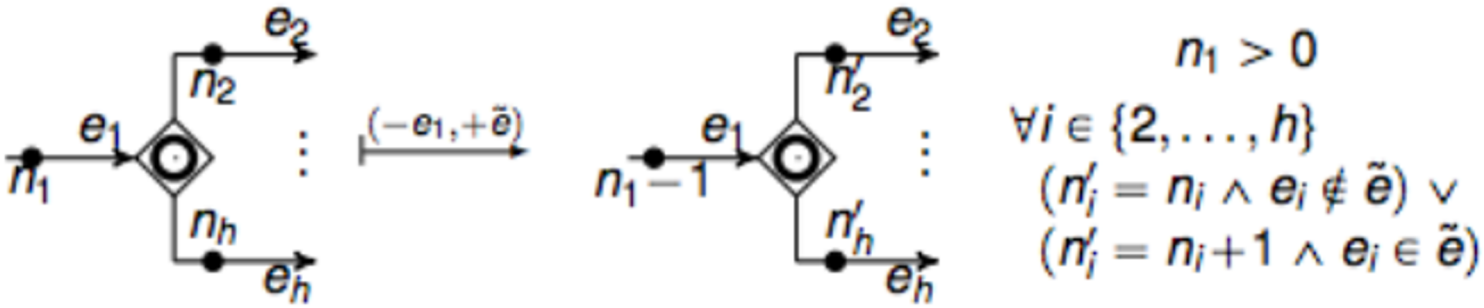


Process Layer - Control Flow Constructs (2)

AND Split:

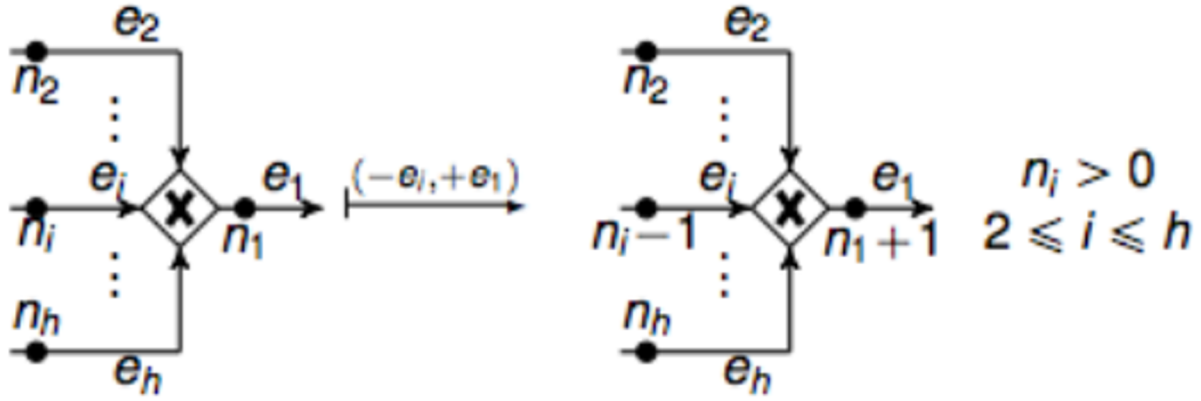


OR Split:

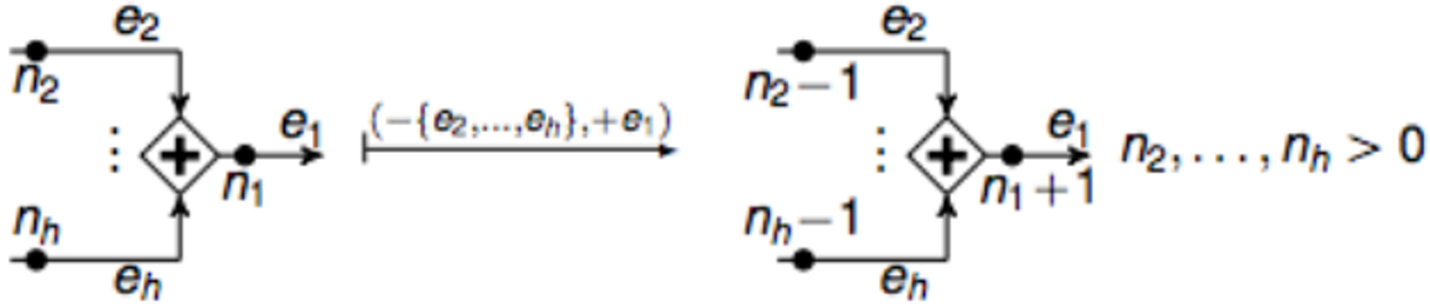


Process Layer - Control Flow Constructs (3)

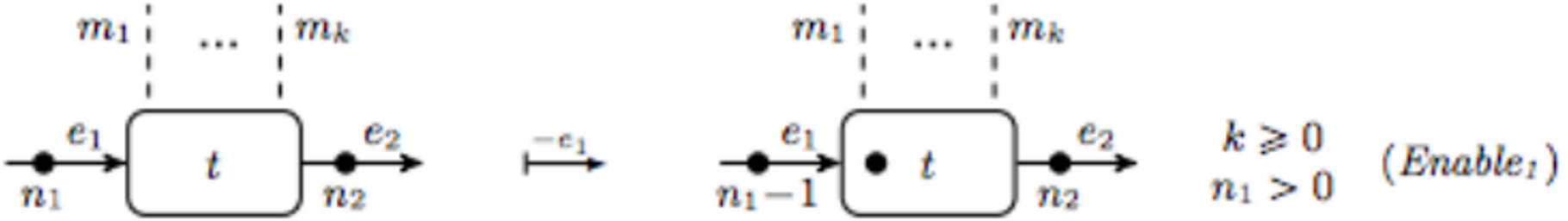
XOR Join:

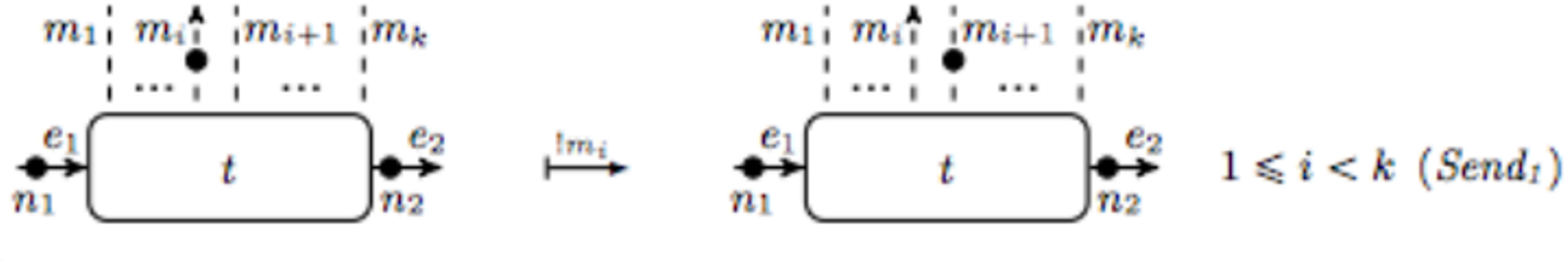


AND Join:



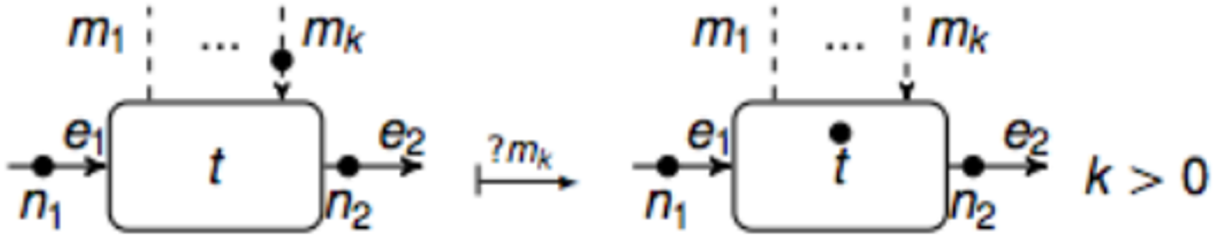
Process Layer - Task Constructs (1)



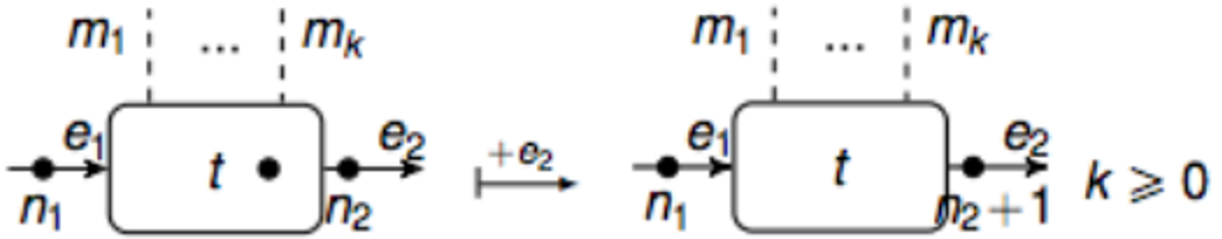


Process Layer - Task Constructs (2)

Receive Task (2):



Complete Task (2):



Process Layer - Propagation

Connected processes:

$$\frac{P_1 \overset{(-\tilde{e}_1,+\tilde{e}_2)}{\longmapsto} P_1'}{P_1 \quad P_2 \overset{(-\tilde{e}_1,+\tilde{e}_2)}{\longmapsto} P_1' \quad P_2 \pm_{\tilde{e}_1,\tilde{e}_2}} \qquad \frac{P_2 \overset{(-\tilde{e}_1,+\tilde{e}_2)}{\longmapsto} P_2'}{P_1 \quad P_2 \overset{(-\tilde{e}_1,+\tilde{e}_2)}{\longmapsto} P_1 \pm_{\tilde{e}_1,\tilde{e}_2} \quad P_2'}$$

Independent processes:

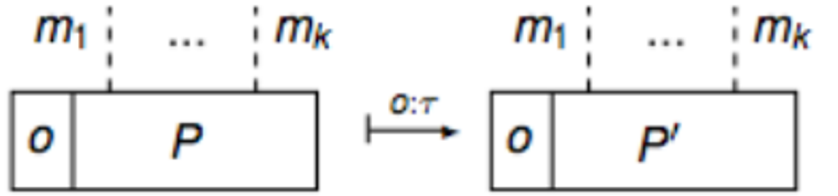
$$\frac{P_1 \overset{\alpha}{\longmapsto} P_1' \qquad \alpha \neq (-\tilde{e}_1, +\tilde{e}_2)}{P_1 \quad P_2 \overset{\alpha}{\longmapsto} P_1' \quad P_2} \qquad \frac{P_2 \overset{\alpha}{\longmapsto} P_2' \qquad \alpha \neq (-\tilde{e}_1, +\tilde{e}_2)}{P_1 \quad P_2 \overset{\alpha}{\longmapsto} P_1 \quad P_2'}$$

Example on collaboration (1)

Internal Action:

Premises

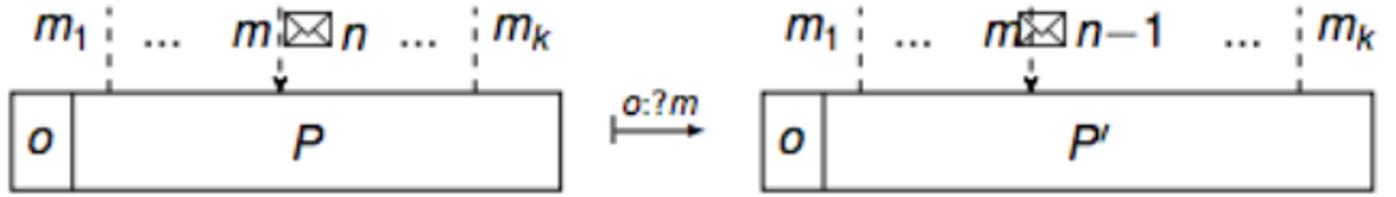
$$P \xrightarrow{\tau} P'$$



Conclusions

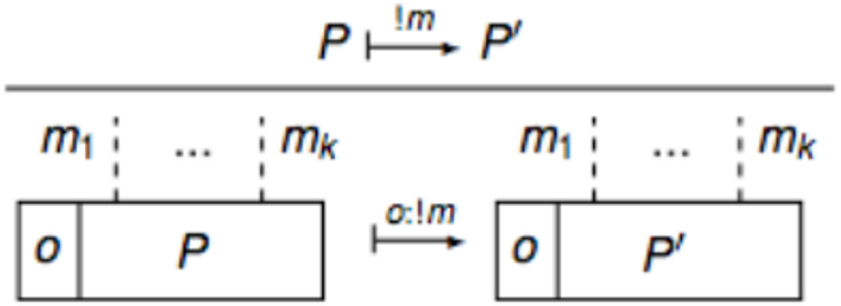
Receive action:

$$P \xrightarrow{?m} P' \qquad n > 0$$

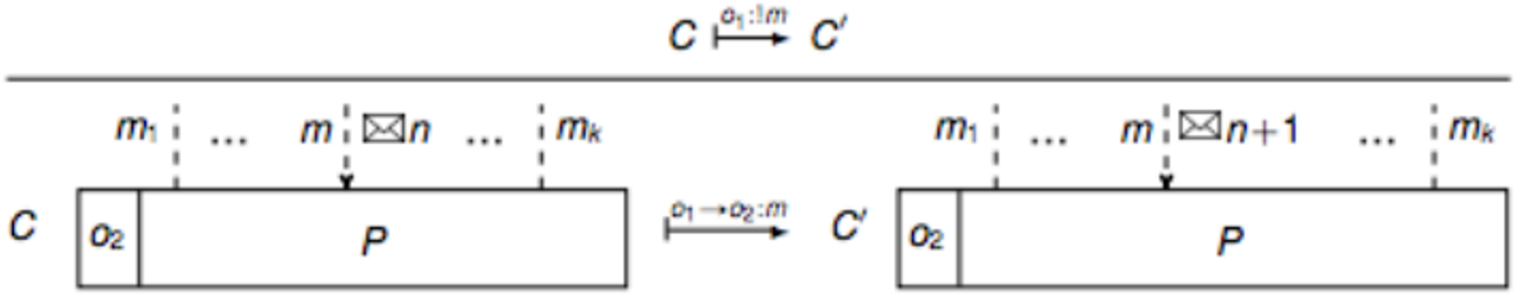


Example on collaboration (2)

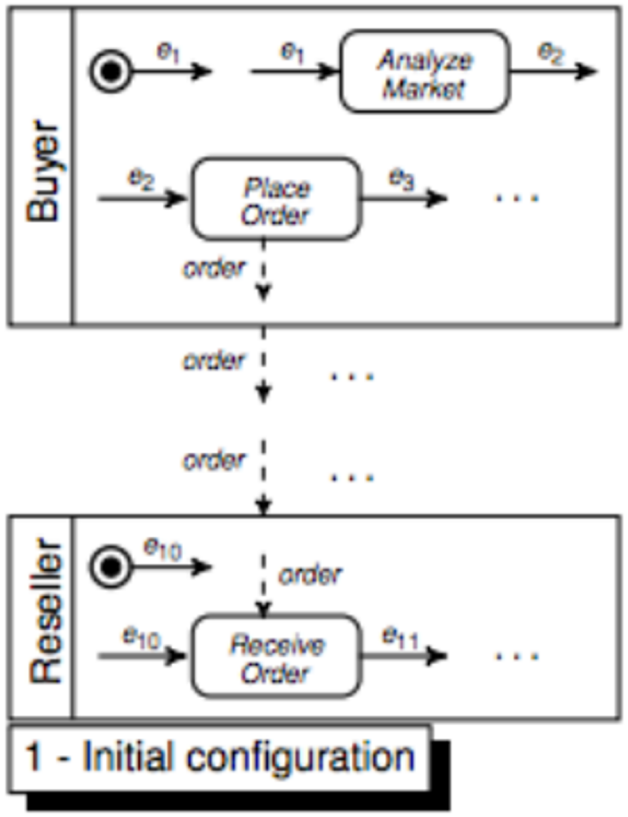
Send Action:

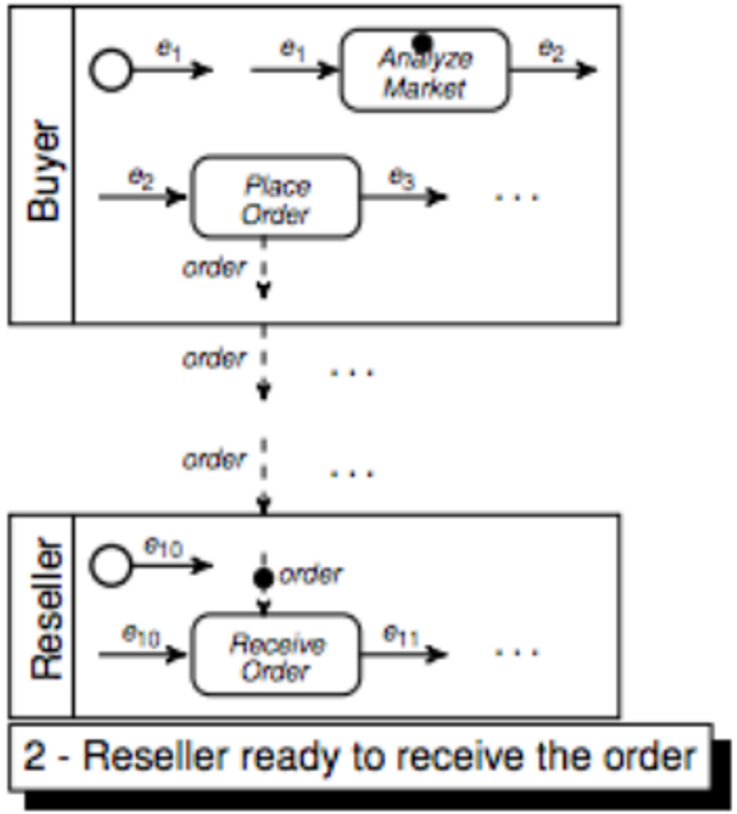


Delivery Action:

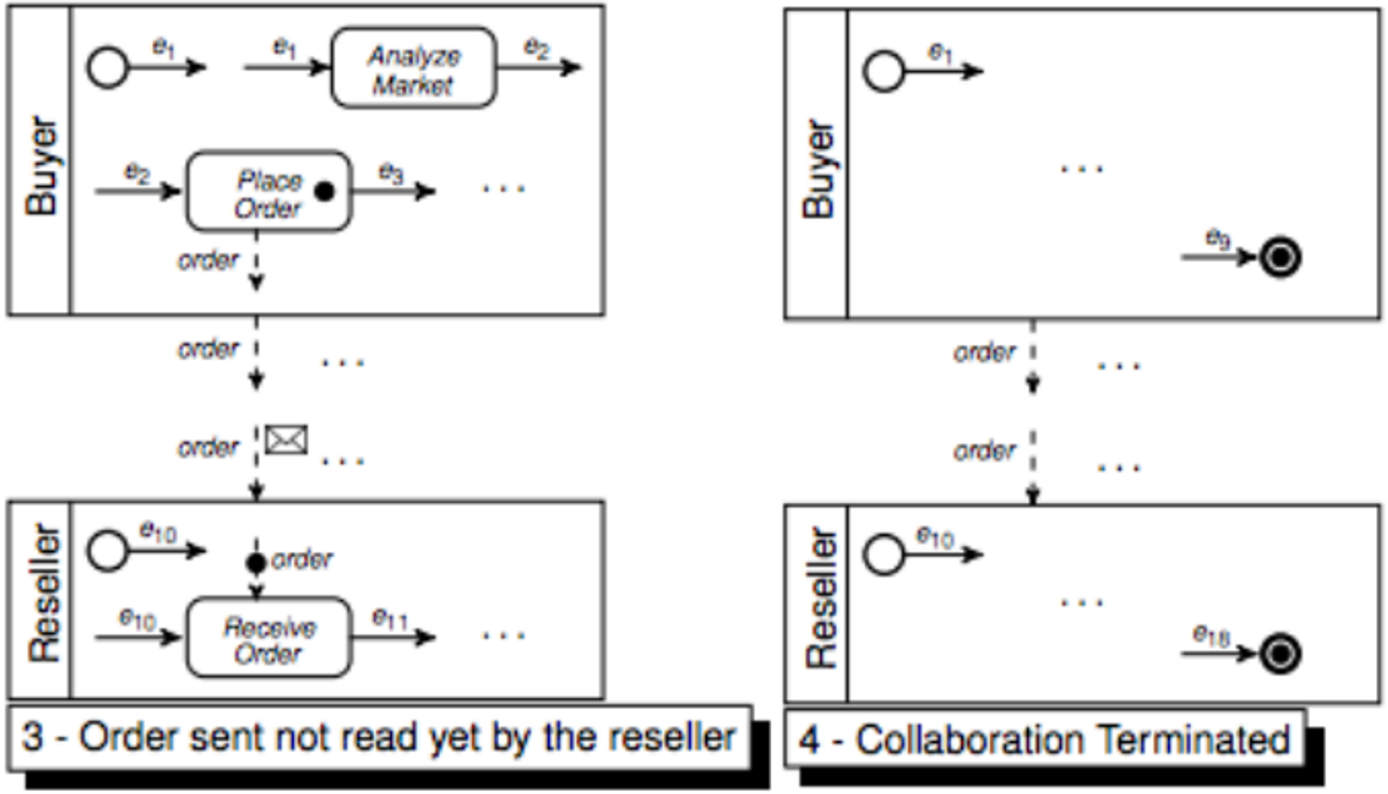


Semantics of the running example (1)





Semantics of the running example (2)



Semantics in Maude

BPMN Operational Semantics in Maude

https://github.com/PROSLab/BPMNOS-Maude

BProVe

http://pros.unicam.it/bprove/bprove-web-interface

Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., & Vandin, A. (2021). A formal approach for the analysis of BPMN collaboration models. *Journal of Systems and Software*, *180*, 111007.

Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., & Vandin, A. (2017, October). BProVe: tool support for business process verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 937-942). IEEE.

Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., & Vandin, A. (2017, October). BProVe: a formal verification framework for business process models. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 217-228). IEEE.

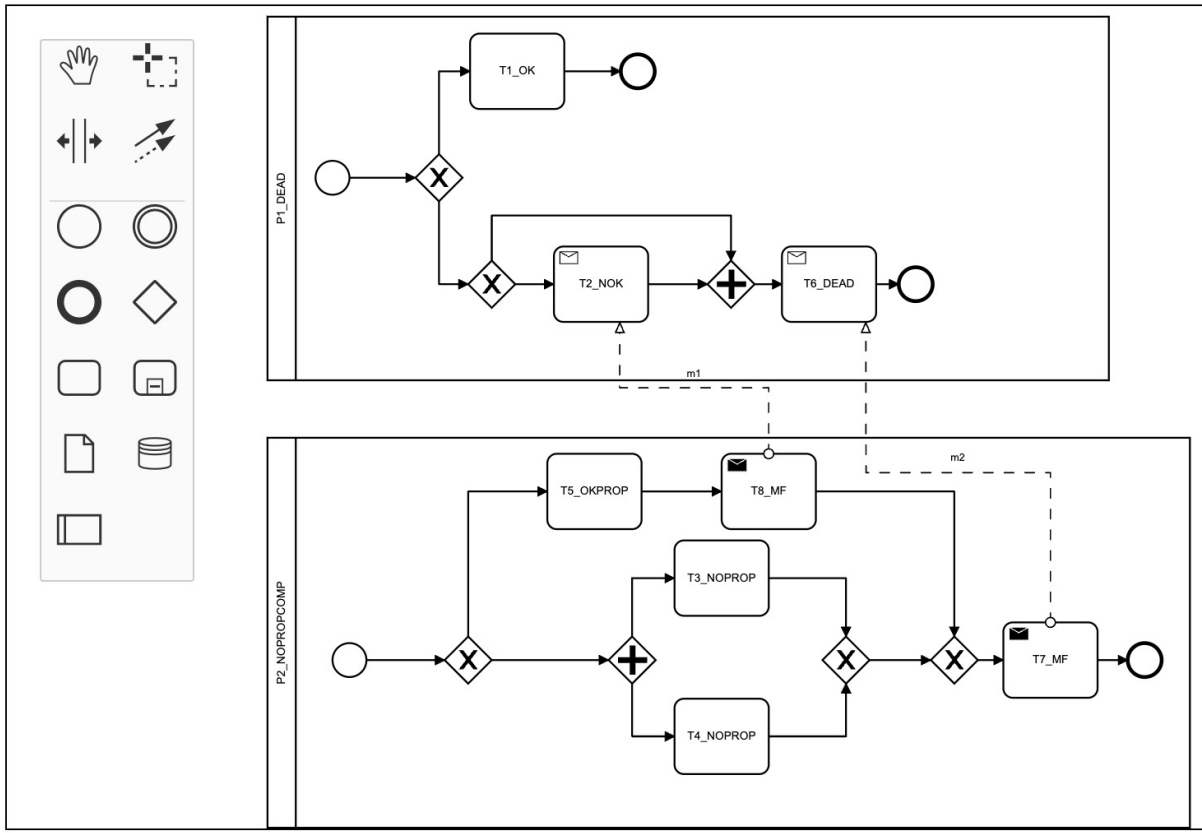
Corradini, F., Fornari, F., Polini, A., Re, B., & Tiezzi, F. (2018). A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming*, *166*, 35-70.

Corradini, F., Fornari, F., Polini, A., Re, B., Tiezzi, F., & Vandin, A. (2017, October). BProVe: a formal verification framework for business process models. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 217-228). IEEE.

BProVe

Welcome to the BProVe Web Interface!

Create a New BPMN Model | Load a BPMN Model | Download Displayed Model | Parse a Model | Open Verification Menu | Zoom to Fit



Property verification result:

```
True in time 11834ms
```

BProVe Verification Menu

Non-Domain dependent properties

Can a Process End? | Check Property On Collaboration

Domain dependent properties

Choose pool... | Choose property to verify... | Check Property On Pool

Choose pool... | Choose message... | Check Property On Rcv Msg

Choose pool... | Choose message... | Check Property On Snd Msg

Choose task... | Choose property to verify... | Check Property On Task

LTL Property Builder

[] ((aBPstartsParameterized("Customers") -> <>
aBPoolendsParameterized("Customers"))

Check your LTL property

Select Verification Tool

LTL Maude Model Checker ☐ MultiVeStA Statistical Model Checker: ☐

Display elements id on click Start