Formal Derivation of Concurrent Executable Specifications

Dan Marius Dinescu

Office Automation and Basic Software for PCs Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71 316 Bucharest
ROMANIA

Ileana Valentina Rabega

Artificial Intelligence Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71 316 Bucharest
ROMANIA

Abstract: The paper presents a software development environment- MIE- (Multispel Integrated Environment)based on formal methods, which assists in both sequential and concurrent system development. MIE uses its own formal, concurrent executable specification language, named MULTISPEL (MULTI level SPEcification Language). MULTISPEL main characteristics are presented and illustrated.

Keywords: software development environments, concurrent systems, client/server model, executable specifications, formal methods, formal specification language, formal verification rules, correctness proof.

Dan Marius Dinescu was born in Romania, in 1954. He received his M.Sc. degree in informatics from the University of Bucharest, Faculty of Mathematics, in 1978. Since graduation, he has been working at the Research Institute for Informatics. His topics of interest include formal specification/programming languages design and use, CASE systems, human interfaces, artificial intelligence, information and data management. His programming experience combines several computer types and operating systems and more than 4o programming languages and DBMSs. He authored and co-authored papers which were published in the country or included in Proceedings of international symposia.

Ileana Valentina Rabega was born in Romania, in 1955. She graduated in mathematics from the University of Bucharest, Faculty of Mathematics, in 1979. Since that year she has been working at the Research Institute for Informatics in Bucharest. Her fields of interest include formal specification/design and correctness proving techniques, CASE systems, artificial intelligence methods applied to software engineering. She is now a senior research worker and her published papers, in Romanian journals or in Proceedings of international symposia, are more than thirty.

Introduction

Use of formal methods in building software development environments has been defined as the main research topic of several projects launched by the EC under their ESPRIT Programme, such as LOTOSPHERE [2, 14], RAISE (LACOS) [15, 16], ATMOSPHERE.It can also be retrieved in some projects to be carried out by American universities, such as : Larch [7] and Unity [9].

A prototype environment used to specify and design sequential and concurrent systems-MIE-has been devised for converging the research topic defined above [4]. MIE uses its own formal, concurrent, executable specification language, named MULTISPEL [3].

MULTISPEL will serve for:

- sequential and concurrent system specification and design
- executable specifications correctness formal verification.

Any problem is liable to being approached by a formal specification language, as MULTISPEL [1, 2, 6]. The only difference proves to be in the extent to which correctness formal verification of a problem can be completed. Critical systems need an in-depth formal verification all through the phases of their life-cycle, while less critical systems need just a correct formal specification to start with.

Several aspects need be defined for critical systems development and MULTISPEL can help as to:

- executable specifications correctness: provability of MULTISPEL source code;
- executable specifications re-use: MULTISPEL source code abstraction
- concurrency of MULTISPEL executable specifications: acceptance of non-determinism and capability of detecting deadlock situations.

Many modern concepts of such programming languages as: Unity [9], Larch [7] [USA], Aspik [2] [Germany], Ada, occam [13] are used by MULTISPEL.

MULTISPEL's original characteristics are the following:

- a framework within which system and process are specified and designed uniformly ;
- protocol definition on faces;
- the way of defining theories .

MULTISPEL stands for a broad spectrum specification language [14, 15, 16] ,as:

- it supports system specification;
- it supports several specification levels (specification refinement);
- it allows specification correctness formal verification, i.e. system development as a result of formal verification rules.

A concurrent system may be developed at three different levels:

Level 1 consists in identifying global system purposes, as a set of state spaces;

Level 2 consists in each formerly developed state space being assigned a hierarchy of abstract machines (the way a MULTISPEL process may be represented). An abstract machine may derive executable specifications, starting from abstract data representation as theories;

Level 3 consists in source code generation with several programming languages, based on the MULTISPEL executable specifications.

A client-server model, using services (the way actions and processes are described), and data to be read/modified by these services, will incorporate a concurrent system specification.

1. MULTISPEL Characteristics

An executable MULTISPEL specification of a concurrent system includes one or several doers. A doer controls resource (data) access; resources (data) are represented by theories.

A sequential or concurrent system specification will be possible by means of the following specification units :

- theories
- processes
- face (types)
- doer (types)

which are finally instantiated as one or several doers, that communicate by sending and receiving messages.

1.1. Theories

A theory can be viewed as:

1) a sentence set, where a sentence represents a relation without free variables;
2) a metapredicate, as far as its signature is concerned: in this sense, theory axioms may be divided into axioms and assumptions; axioms decide on whether the predicate is true or false, whereas assumptions must be true even though the predicate proves false, otherwise a semantic error will be reported.

The theory symbols name (value) sets, values, relations. A theory parameterization will use notions named sorts, operators, abstract relations. Parameters form a signature which , for each notion (i.e. sort, operator, relation), contains sorts manifest in their arities (generically arity names a relation domain or an operator/reducer < domain, range > pair).

A signature, therefore, contains symbols for distinct:

- (nullary) identifiers/numbers;
- relators
- infix operators
- postfix operators.

Variables represent sort arbitrary values (homonymous identities).

Predicates represent elementary relations (i.e. they do not contain other relations). They are sorted, that is their domain is a sort. A special predicate example is equality (" = "), which is a congruence, and the only one of the system. In some of their variables, relations (including predicates) can be univoque . If so, and given a restriction, they can be functional, i.e. they can admit inverses in those variables.The relation properties may be expressed as follows:

- there is a solution satisfying the relation:

  :e: $\in$ D i.e. for a known "D", an "e" can be found so that e $\in$ D

- all solutions satisfying the relation may be found

  SEQ :e: $\in$ D i.e. for a known "D" all "e" can be found (ennumerated) so that e $\in$ D.

Reducers represent recursive closures of binary functions on enumerations.

The theory object consists of:

1) sorts: an ordered set, S, based on which an order relation " < " is defined; order extends to the Cartesian product S^n, $n \geq 2$, by components; sorts should be assigned distinct identifiers;
2) operators, possibly indexed (optionally restricted) form set Σ, which defines value names space, consisting of identifiers, (infix or prefix) operators, numbers; operators can be homonymous, but if they are,they will be of different arities;
3) an indexed predicate set, P, (optionally restricted) containing identifiers, relators: predicates can be homonymous, but if they are, they will be of different arities;
4) a reducer set, R, (optionally restricted) containing identifiers, possibly homonymous , if a restriction is made on the argument enumeration elements. Domain restrictions are relations on indexes, true, if in the notion domain and undefined, if not; restrictions need not dynamical,but statical proofs .

Operators and relations must verify the regularity condition:

for each $\sigma \in \sum \left(w_1, s_1\right)$ and each w_0, with

$w_0 \leq w_1$ the set

$\left\{(w, s) \mid \sigma \in \sum (w, s) \text{ and } w \geq w_0\right\}$ has a least element (where w, w_0, w_1, s_1 are sorts in S);

for each $p \in P_{w1}$ and each w_0, with $w_0 \leq w_1$:

$\left\{w \mid p \in P_w \text{ and } w \geq w_0\right\}$ has a least element

(where $p \in P_{w1}$ shows that predicate p has arity w1).

The regularity condition makes:

- a homonymous operators domain set, including reducers, be closed to infimum, extended by ϕ;
- homonymous operators ranges, including reducers with the same domain, be disjunctive;
- an operator of which arity is inferior to another homonymous operator be a restriction of the latter.

The theory body is composed of theorems and definitions which will be referred to as methods (as in object-oriented programming). The theory body can contain possibly (confined) indexed distinct sorted free variables;the variable names should differ from the operator and predicate symbols.

Theorems (sentences) contain either metapredicates (equivalent to axioms conjunction in the theory) or first-order predicates.The theorems (as first-order predicates) where all the variables looked upon as not annotated are universally quantified will contain the definition annotation (": :") just in the following cases:

a') A predicate (or relation "V = Term") of which arguments are only variables and some of them definitionally annotated, asserts that there is a method for enumerating all definitionally annotated variable valuations, that verify the relation yielded by overlooking annotations; this is an enumerating inverse theorem. E.g. :e: $\in$ S means that there is a method determining all values of "e", which, for given "S", solves relation "e $\in$ S";

a'') A predicate (or relation "V = Term") having either distinct variables or definitionally annotated operators applied to the variables and asserting that the relation is functional in annotated positions and the operators represent or name these functions. This is a functional inverse theorem. For example,

n + :m - n: = m

says that relation "n + x = m" is functional in "x" and "m - n" is this very function.

b) Decomposition/analysis/induction/classification/ partitioning theorems, as in the following example

$<> V = T_1 <> V = T_2 <> \ldots <> V = T_n$

where either "T_i" is a nullary operator or "$V = T_i$" is a functional inverse.

E.g. < > n = 0 < > n = succ(:pred(n):)

says that not annotated operators range values cause a partition on variable "n" sort and that the operators do admit partial inverse functions (as in a").

Methods are solution-producing,but not sentence form-producing. The types of methods go from methods determining predicates and producing functions evaluation to inversing methods.

In the following example the theory of natural numbers is built.

The entire signature < Nat > [+,-,pred, ≤,0,1,succ] is definitionally annotated. The theory is constructive i.e. all functors are defined within it. Functors, as it is easily observable, are defined in (inductive) def clauses, i.e. they can be executed. Theorems are non-executable and are used but for specification correctness proofs. The proofs can be included in the specification body, as the following example does. An automated theorem prover can also be used. MIE uses OTTER prover, as illustrated by the Chapter 5 example.

```
THEORY:  < Nat >  [ +,-,pred,≤,0,1,succ]:
         Natural
   SET   Nat; VAR n,m,p: Nat;
   VAL   0,n + m,1,succ(n): Nat,
         pred(p) | p /=0 : Nat;
            -- pred(p) (excepted for p = 0) is
            --a natural number
INDUCTIVE DEF :(:n: + 0): = n,
            :(:n: + succ(:m:)): = succ(n + m);
-- . + .
   DEF :1: = succ(0);
   REL n≤m;
INDUCTIVE DEF :(0≤:n:):,:succ(:m:)≤
                  succ(:): ⇔m≤n;
TH. n≤n + m;
   VAL n-m | m≤n : Nat;


   DEF :pred(succ(:n:)): = n;
   TH. < > n=0 < > n=succ(:pred(n):);
   TH. pred(:succ(n):) = n;
   TH. < > m/≤n< > m + :n-m: = n; DEM
   INDUCTIVE DEF :n: + 0 = :n:,
                 :n: + succ(m) = succ(n + :m:);-- . + :
END DEM;
DEF m + :(:n:-:m:): = n;
   TH. n + m = m + n;-- in fact
     -- < Nat,0, + > abelianGroup
   TH. < > m/≤n < > :n-m: + m = n;
TH. :n + m:-m = n;
TH.  < > m/≤n < > n-:n-m: = m;


 END THEORY;
```

1.2 Processes, doer (types), face (types)

The "dynamic" specification units (processes, doer (types), face (types)) describe system behaviour as state machines, in which data are expressed as theories; doers or doer types are real (parameterized) machines, while faces or face types are virtual (parameterized) ones. MULTISPEL processes need exist since, during the specification process of real problems, there are moments when the machine nature (whether real or virtual) cannot be defined ; clearly a MULTISPEL process evolves as a face (type) or doer (type) .

An execution process in a dynamic specification unit is described by means of services. They are defined on faces or face types; first they are declared and then their behaviour is given using abstract data expressed in theories. LET clause can be used in importing current theories or special-purpose theories can be defined within all dynamic specification units. Service order on faces or face types indicates the protocol.

MULTISPEL processes and doer (types) communicate asynchronously. They are either clients or servers, in respect of their way of running services.

Services are described on faces or face types and are implemented on doer or doer types. The distance between a service description and its implementation is solved by the specification enrichment at doer (type) level. A machine behaviour, given as a state sequence and starting from an initial state, will do this. Machine invariants, consisting of a temporal assertion, are user's initiatives. Anyhow,they must remain unaltered by any transition execution. Invariants serve to prove doer (type) behaviour correctness.

Therefore, a doer (type) provides some services. When ordered to execute some service, a doer (type) acts like a server towards other doers (doer types) acting like clients (they give the orders).Given input arguments, an order is asking this time for some services to be executed, and certain results to be obtained . On accepting an order the state of the system gets altered, and enables a transition.

Therefore, a transitions set describes a (real or virtual) machine behaviour.

A transition can be described as a condition-action relation type, where

- the condition may be
  - a (list of) Boolean(s)
  - service
  - return
- the action represents a list of MULTISPEL actions.

A MULTISPEL action may be either primitive or composed. A primitive action will include service, return, stop or a temporal predicate next (') expressed by means of output definition (e.g. :a': = 1;).

The operators for composing MULTISPEL actions are: (seq) and, (seq) or, ⇒ (meaning "then"), < > (for several or-s).

The following example presents a register specification:

process <Data> Reg

set Data;

is var d: Data;

ex action store(d);

ex val exp_value: Data;

state val stored: Data;

initial not :def stored'::;

< > store(:stored:) => :stored': = d

def :exp_value: = stored;

end process;

An external action "store" depending on a variable from a "Data" set is declared in the process specification . A state value is also stored in the same "Data" set. Initially, a next value for state val "stored", i.e. stored' is undefined. "Reg" process considers an one-transition machine which changes the "stored" value. The exported value "exp_value" is assigned the current value of "stored".

1.3 MULTISPEL grammar

MULTISPEL grammar is given below.

CompUnit = LibUnit* [Doer]

LibUnit = Theory | Process

Theory = (**theory** Genericity TheoryEl*)+ **end theory** ";"

| **theory** Genericity **is** MetaPredicate ";"
renaming.

Genericity = *Object*IOSignature
MetaPred Ident *Param*Signature*

IOSignature = Signature | ":" Signature ":"

Signature = "<" IOParam+, ">" ["[" IOSymbol+, "]"]

IOParam = IOSignature | IOSymbol

IOSymbol = *Inp*Symbol | ":" *Out*Symbol ":"

Symbol = Ident | Number | Operator ["."] | DefinedRelator

TheoryEl = VarDecl | Ex (SetDecl | ValDecl | RelDecl | RedDecl)
| Assume | Theorem | Let | Definition

Ex = [**ex**]

Assume = **assume** *Prop*Relation+, ";"

Theorem = (**ax** | **th**) [Title] "." *Prop*Relation+, ";" Dem*

Dem = [**inductive**] **dem** TheoryEl+ **end dem;**

Title = String | Ident | Number

Let = **let** MetaPredicate+, ";"

Definition = [**inductive**] **def** *Prop*Relation+, ";"

SetDecl = Ex **set** SetIdDecl+, ";"

SetIdDecl = *Set*Ident
*Set*Ident ">" *Set*Ident ## extension.
*Set*Ident "<" *Set*Ident ## abs. subset.
*Set*Ident "=" SubSet

Subset = *SubSet*Ident ## rename.

XPath ArgStru*
FACE by procotol.

"{" *Var*Ident "|" Relation "}" ## subset.

[*Set*Ident] "{" Elem $^{+,}$ "}" [Succ]

Elem = Term ["÷" Term]
orig. SUCC range.

Succ = "<" | "o" ## liniar, circular.

OptRestrictions = ["|" Relation $^{+,}$]

ParList = ["(" *Var*Ident $^{+,}$ ")"]

VarDecl = **var** VarList $^{+,}$ OptRestrictions ";"
##no overloading.

VarList = *Var*Ident $^{+,}$ ":" SubSet

ValDecl = **val** ValSList $^{+,}$ OptRestrictions ";"

ValSList = Val $^{+,}$ ":" SubSet

Val = *Op*Ident ParList

| *Un*Operator *Var*Ident | *Var*Ident
*Bin*Operator *Var*Ident

RelDecl = **rel** PredDecl $^{+,}$ OptRestrictions ";"

PredDecl = *Pred*Ident ParList

*Var*Ident DefinedRelator *Var*Ident

RedDecl = **red** RedList $^{+,}$ ";"

RedList = *Red*Ident $^{+,}$ ":" Subset

XPath = XId $^{+.}$

XId = Ident ["(" Term $^{+,}$ ")"]

Path = Ident $^{+.}$

DemoPragma = "{" (Title $^{+.}$) $^{+,}$ "}"

Implicatie = "←" | "→" | "⇔"

Relation = ("< >" Relation) $^{+}$ ## partition.

-- rightassoc Relation "⇒" Relation

-- Relation (Implicatie Relation) $^{+}$

-- Relation **or** Relation

-- Relation **and** Relation

-- **not** Relation | "(" Relation ")" | ":" Pred1 ":"
DemoPragma Relation

def StateVar

ValuePred | MetaPred

Term
(implicit assignment) TRUE.

Pred1 = Term Relator Term | Predicat

ValuePred = Term (Relator Term) $^{+}$ | Predicat

Relator = "/" + DefinedRelator | "/="

DefinedRelator | "="

DefinedRelator = "≈" | "≡" | "∈" | "?" | "¿" |
"■" | "<" | ">" | "≤" | "≥" | "<<" | ">>"

DefinedRelator + Sufix

Sufix = "∘" | "’" | "‘" | "∩" | "2" | "a̲" | "o̲"

MetaPred = IOArgStru *MetaPred*Ident ArgStru*

IOArgStru = ArgStru | ":" ArgStru ":"

ArgStru = "<" IOArg $^{+,}$ ">" ["[" OptArg $^{+,}$ "]"]

IOArg = IOArgStru | IOSymbol

OptArg = [Symbol "⇒"] IOSymbol

Predicat = *Pred*XPath | *StatePred*XPath

StateVar = *StateVar*XId

Term = Term AddOp Term
left-associative.

-- Term MulOp Term
left-associative.

-- Term ":" *Sort*Ident
disambiguation.

-- *Un*Operator Term | "(" Term ")" |
DemoPragma Term

*Val*XPath | *Var*Ident

StateVar ["’"]

*Red*XPath "[" Enumerate "]"

"[" Relation "]" ## (implicit assignment)
##VALUE OF

":" Term ":" ## defining occurence

"|" Term "|" ## only in canonic
##definitions

"{" Enumerate "}" ## implicit OR regarding
##contained pred.

makes relator negation
##necessary

AddOp = "+" | "-" | "\" | "\/" | " ?" | "#" | "@" |
"!" | "−" | "±"

AddOp + Sufix

MulOp = "*" | "/" | "%" | "/\" | "&" | "^" | "~" |
"¡" | "¬" | "•"

MulOp + Sufix

Operator = AddOp | MulOp

Enumerate = Term "|" Relation

(Term ["÷" Term])*,
SUCC to be used

Asynch = [**asynch**]

Process = Asynch **process** Type **is** XPath
ArgStru* ";"

| Asynch **process** Type **is** ProcessDesc **end process** ";"

ProcessDesc = ProcessAux* Procedure

Type = Ident ["(" Param +, ")"] *Param*Signature*
ParamDesc*

ParamDesc = TheoryEl

Doer = Asynch **doer** XIdList ":" XPath ArgStru*
OptRestrictions ";"

| Asynch **doer** XIdList ":" ProcessDesc **end doer** ";"

iterable.

ProcessAux = Face | Initial | Invariant | Server

|Ex (Process | Doer | State | ActionDecl
| Service)

| Ex (ValDecl | RelDecl | RedDecl)

Face = **face** XIdList ":" XPath ArgStru*
OptRestrictions ";"

| **face** XIdList ":" ProcessDesc **end face** ";"

iterable.

ServiceDecl = **service** Message+,
OptRestrictions ";"

Message = *Service*Ident ParList (**return** ParList) +

State = **state var** SVarTList +, OptRestrictions ";"
iterable.

state ValDecl

state RelDecl

state RedDecl

SVarTList = XIdD +, ":" Subset

ActionDecl = **action** Actiune1 +, OptRestrictions
";"

Actiune1 = Ident ["(" ActPar +, ")"]

ActPar = "‘" Ident | Ident | ":" Ident ":"

Server = **server** FaceId +, OptRestrictions ";"

FaceId = [Ident "="] XPath

Initial = **initial** Action ";"
except I/O & STOP, iter.

Invariant = **invar** Relation +, ";"

Procedure = ("< >" Transition) +

Transition = Guard +, "=>" Action ";"
Trans. iter by guard.
Action iter by itself.

Guard = Relation | Service | Return

Action = Action "," Action

-- Action **or** Action

-- Action **and** Action

-- "(" Action ")"

Service | Return | **stop** | LocalAction

LocalAction = Relation
temporal

[Xpath "."] Ident ["(" ActArg +, ")"]

ActArg = Term | "‘" StateVar

Number = Digit | Number + Digit

Ident = Ident + AN | Letter

AN = Letter | Digit

Digit = "0"÷"9"

Letter = "α" | "ß" | "Γ" | "π" | "Σ" | "σ" | "μ" | "τ" | "Φ" | "Ω" | "δ" | "∞" | "Ø" | "A" ÷ "Z" | "a" ÷ "z" | "_"

2. MULTISPEL Basic Model

MULTISPEL basic model is composed of an algebraic model and an operational model.

2.1. MULTISPEL Algebraic Model

Based on the associated MULTISPEL signature an algebra of terms (AT) is built, which is the least family of expressions obtained by means of the operators defined in the signature. The equations set (E) is then considered. It contains the well-formed formulae which have "equality" as unique predicate. The AT/E quotient algebra creates the set of equivalence classes. The value set in MULTISPEL algebraic model is built by selecting a representative value out of each equivalence class.

This model is a framework within which correctness of MULTISPEL theories may be proved. A MULTISPEL algebraic model is associated with any theory; it can be enriched (a theory and, subsequently, its model) by adding new MULTISPEL sentences, corresponding to the already defined operators, or by adding new operators; that is, a theory can import another theory.

2.2. MULTISPEL Operational Model

MULTISPEL operational model refers the dynamic, temporal aspect of a system, whereas the algebraic model refers the static, immutable part of it. MULTISPEL operational model appears to be a state machine

$(S, Inp, Out, g, s_0, S_{FIN})$

where

S = a (possibly infinite) state set,

Inp = an input set,

Out = an output set,

s_0 = an initial state,

S_{FIN} = a (possibly void) final state set,

g : (Inp*, S)→(Out*, S) a non-deterministic transition function

$$g\left(<i_1, \dots, i_n>, s_{ante}\right) = \left(<o_1, \dots, o_m>, s_{post}\right)$$

A MULTISPEL doer is modelled as

- a state machine

or

- a communication process among other (sub)doers, to be in turn modelled as a state-machine or a communication process a.s.o. recursively

or

- a mixed way, both a state-machine and a communication process.

A doer (type) state at one particular moment is given by a pair

< Values, Relations >

where

Values = state variables values (including undefined values);

Relations = other doers (doer types) (which form a bag or multiset) with whom the current doer (type) is involved as sender or receiver.

A transition is presented as a pair of such states, thought of as being successive.

3. MULTISPEL Specification Style

Example 1

A well-known bounded buffer problem is going to be specified in MULTISPEL.

Viewed as a buffer for smoothing speed variations in the outputs of a producer process and in the inputs of a consumer process , a process will be specified and implemented under MULTISPEL terms (where M represents the maximum number of inputs/outputs), as follows:

asynch doer type BoundedBuffer < Elem > < Nat > (M)

as. < Nat > [NonNul⇒:PositiveNat:] Natural;

-- NonNul becomes PositiveNat, which is
--obtained from Nat theory

val M: PositiveNat;

set Elem;

is var e: Elem;

```
-- face In declaration is introduced
face In:
ex service write( e) return, close return;
   -- a virtual machine is defined within face In;
   -- when
   -- service write(:e:) produces, e is obtained
   --before
   --    (:.: = definition annotation)
      < >service write(:e:) => return write;
      < >service close => return close, stop;
 end face;


 face Out:
  ex service read return (e) return;
    -- a theory instance is introduced by let; this
    --theory is
    -- built within string (to see definition anno-
    -- tation)
    let :<String>[null,&]: string <Elem>;
    state var B: String;
     -- this virtual machine is composed of
--transitions with guards; their truth value "true"
--permit transition firing
      < >B/=null, service read => B=:e:&:B':,
        return read(e);
      < >B=null, service read => return read,
        stop;
 end face;


  val L:Nat; def :L:= 0;
  -- a new set is built by restricting Nat to the
  --interval [L÷L+M-1]
  var i: Nat [L÷ L+M−1];
  -- within the defined restriction set, a new set,
  --Index is
  -- built, by means of a constructor applied to
  -- variable i
  -- ranging over the mentioned set and having a
  --circular SUCC functor defined on it
  set Index= [^i] o; var p: Index;
  doer F, T:
    ex state val val: Index;
    ex action next;
    state var crt: Index; initial :crt':= ^L;
    def :next:⇔:crt':= succ( crt), :val:= crt;
  end doer;
  -- the let clause instantiates
--additiveExtension<Nat>
  -- to set Index, where "+" operator symbol is
  -- marked
  -- as output (i.e. is defined within the theory)
  --and
  -- can be used with this particular symbol or a
  -- user-defined one, having the same meaning


   let <Index> [ :+:] additiveExtension
<Nat>;
  state var B(p): Elem, c: Nat,
        input: {closed, open};
  state rel full, empty;
  state val f,t: Index;
  def :full: ⇔ c = M,
     :empty: ⇔ c =0,
     :f:= F.val,
     :t:= T.val;
  invar f=t+c, c ≤ M,
  -- pred(f) means predecessor, that is
  -- pred(f)=f-1 or undef
  -- and is supposed to be imported from Nat
      < >t/=f = p={t÷pred(f)}⇔ def B(p)
      < >t=f ⇒ (< > empty< > full)
          and (< > empty< > def B(p));
  --the real machine is introduced: it has an initial
  -- clause,
  --in which variables are assigned initial values
  -- and a
  --transition set, that describes machine
```

```
-- behaviour;
initial :c':=0, :input':=open, not :def B(p)':;
 < >not full, service In.write(:B(f)':) =>
    :c':= c+1, F.next,  return In.write;
 < >not empty, service Out.read =>
    :c':= c-1, T.next, not :def B(t)':,
    return Out.read( B( t));
 < >service In.close =>:input':=closed,
     return In.close;
 < >empty, input=closed,
      service Out.read=>
    return Out.read, stop;
end doer type;
```

BoundedBuffer is specified as an asynchronous doer type, in which two faces, In and Out, are introduced and two internal doers, F and T, expressing the machine invariant, are also considered. It is worth mentioning that MULTISPEL let temporal assertions be used (**invar**) for verifying formal machine correctness. Another way of specifying this problem by using MULTISPEL is to define a theory which should contain the forementioned services as operations/functions. In such a case correctness proof falls under the theory's resposibility (by proving theorems and definitions), and not under machine's.

```
theory: <String>: StringOf <Elem> <Nat>
  as. <Nat> Natural; -- :succ:.
  set Elem, String;
  var S,T: String, E,F: Elem;
ex val E&S, S&E, null: String, #S: Nat;
  val first(S) | S/=null: Elem,
     rest(S) | S/=null: String;
  ax. < >S = null < > S = :first(S):&:rest(S)::;
  inductive def :#null:=0, :#(:E:&:S:):=
                                  succ(#S),
     :(:E:&:S:)&:F::= E & (S & F), :null &:E::=
                                  E & null;
  -- the vertical bar attached to a value declaration
  -- means
  -- "with exception of that particular value", the
  -- declaration stands
  val last(S) | S/=null: Elem,
     leading(S) | S/=null: String;
  th. < >S = null < >S = :leading(S):&:last(S):;
  inductive def (E & S) & F =:E:&(:S:&:F:),
     null&F=:F:&null;
end theory;
asynch doer type    <Elem>  <Nat>
BoundedBuffer( M) | M>0
  def: <String>: StringOf <Elem> <Nat>;
 -- :&, &, null, #.:;
-- face set Elem; & as. <Nat> Natural; :≤:.
  val M: Nat;
is var e: Elem;
  face In:

   ex service write( e) return, close return;
      < > service write(:e:)=> return write;
      < >service close => return close, stop;
  end face;
  face Out:
   ex service read return (e) return;
     state val B: String;
      < >B/=null, service read=> B=:e:&:B':,
         return read(e);
      < >B=null, service read=>return read,
         stop;
  end face;

  state var B: String, input: {open, closed};
  state rel empty; def :empty: ⇔ B=null;
  initial :B':=null, :input':=open;
   < >#B ≤M, input=open,
      service In.write(:e:)=>
     :B':= B&e,  return In.write;
   < >input=open, service In.close =>
```

:input':= closed, **return** In.close;

< >**not** empty, **service** Out.read =>

B =:e:&:B':, **return** Out.read(e);

< >input = closed, empty, **service** Out.read

=> **return** Out.read, **stop**;

end doer type;

Example 2

Here is an example of building a rainbow specification using MULTISPEL. A starting theory is defined, followed by more and more specific refining theories. It is refinement that helps in the process of formulating definitions (def clauses) which are part of a theory which is indeed executable.

theory <RAINBOW>[red, orange, yellow, green, blue, indigo, violet, succ, pred]

Colours

set RAINBOW; **var** C: RAINBOW;

val red, orange, yellow, green, blue, indigo, violet,

succ(C) | C/= violet : RAINBOW;

ax. < >C = red < >C = orange < >C = yellow < >C = green

< >C = blue < >C = indigo < >C = violet;

ax. succ(red) = orange,

succ(orange) = yellow,

succ(yellow) = green,

succ(green) = blue,

succ(blue) = indigo,

succ(indigo) = violet;

val pred(C) | C/= red : RAINBOW;

ax. < >C = red < >C = succ(:pred(C):),

< >C = violet < >C = pred(:succ(C):);

theory <RAINBOW>[red, orange, yellow, green, blue, indigo, violet, succ, :pred:]

Colours

-- in this theory, functor pred is a constructor,

-- serving to define the other operators

def :pred(orange): = red,

-- definition (unnormalized)

:pred(yellow): = orange,

:pred(green): = yellow,

:pred(blue): = green,

:pred(indigo): = blue,

:pred(violet): = indigo;

theory <RAINBOW>[red, succ, :orange:, :yellow:, :green:,

:blue:, :indigo:, :violet:, :pred:]

Colours -- RED & SUCC are still input

-- parameters.

-- orange, yellow, green, blue, indigo, violet, pred

-- are constructors

def succ(red)

:orange:,

succ(orange) =:yellow:,

succ(yellow) =:green:,

succ(green) =:blue:,

succ(blue) =:indigo:,

succ(indigo) =:violet:;

theory :<RAINBOW>[red, orange, yellow, green, blue,

indigo, violet, succ, pred]:

Colours

-- the whole theory is constructed in this

-- refinement

def :pred(succ(:C:)):= C; -- more specific

-- redefinition

-- because SUCC is constructor.

end theory;

theory <RAINBOW>[red, orange, yellow, green, blue,

indigo, violet, succ, :pred:]

```
Colours -- enriched (with subordinated
                    -- theories)
theory <RAINBOW>[ :succ:, red, orange,
        yellow, green,
        blue, indigo, violet, :pred:]
  Colours

def :succ( red):     = orange,
  :succ( orange):  = yellow,
  :succ( yellow):  = green,
  :succ( green):   = blue,
  :succ( blue):    = indigo,
  :succ( indigo):  = violet,
end theory;
```

```
theory : <RAINBOW>[ red, orange, yellow,
         green, blue,
         indigo, violet, succ, pred]:
  Colours  <Nat,k>
 as. <Nat> Natural; -- inherits :succ:,: + :.
 val k: Nat;
 -- NumColor is a restriction of Nat to interval
 -- [k ÷ k+6]
 -- which inherits Nat successor functor
  set NumColor = Nat [k÷k+6]; var  x:
NumColor;
 -- succ(x) is undefined for x = k+6, with given k
 val succ( x) | x/=k+6 : NumColor;
 th. <NumColor> [ red⇒k, :orange:, :yellow:,
        :green:,
        :blue:,:indigo:,:violet:, :pred:] Colours;
  -- another (noncanonical) interpretation is
  -- asserted
  -- present operator symbols may be changed at
  -- user discretion
  -- ( the meaning being preserved anyway)
end theory;
```

The signature is parameterized by the operators:

- red, orange, yellow, green, blue, indigo, violet

as well as

- succ, giving next colour
- pred, giving previous colour.

Example 3

The well-known problem of dining philosophers (due to E. W. Dijkstra) is going to be specified in MULTISPEL.

Problem: Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a large bowl of spaghetti, and the table is laid with five forks. When feeling hungry, a philosopher enters the dining room, sits in his chair, and picks up the fork on the left of his place. Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. When he finishes, he puts down both forks, and leaves the room. The room should keep a count of the number of philosophers in it.

The problem is specified by means of a process, depending on a philosopher's number (here N) in which 2*N + 1 doers asynchronously run : N asynch doers phil(i), N asynch doers fork(i), and one asynch doer room.

```
process  <Nat> ThemPhilosophers (N)
as. <Nat> Natural;
val N:Nat; as. N>2;
-- two new sets are built Nat [1÷N], which restrict
-- Nat
-- to interval [1÷N] and [^b] o, which transforms
-- this
-- interval into a set with circular successor functor
-- (the successor functor is inherited from Nat)
is var b:Nat [1÷N] ;
  var i:[^b] o;
```

```
-- for running this example N asynch doers fork(i)
-- must be active

asynch doer fork(i):
  ex service pickUp return, putDown return;
    state val mis: {held, free}; initial: mis': = free;
  < >mis = free, service pickUp=>
       :mis': = held , return pickUp;
  < >mis = held,serviceputDown=>
      :mis': = free, return putDown;
end doer;
asynch doer room:
face door:
  ex service enter return, exit return;
    state val mis:{in,out}; initial :mis': = out;

    < >mis = out , service enter => :mis': = in ,
                         return enter;
    < >mis = in , service exit =>:mis': = out ,
                        return exit;
end face;
  var occupants: Nat; initial :occupants': = 0;
  -- service names can be prefixed by face names;
  -- this
  -- notation becomes necessary for homonymous
  -- service names
  invar occupants  <N;
  < >occupants  <N-1 , service door.enter =>
       :occupants': = occupants + 1 ,
        return door.enter;
   < >occupants >0 , service door.exit =>
       :occupants': = occupants - 1 ,
        return door.exit;
end doer;
-- for running this example, N asynch doers phil(i)
must
-- be active, and totally 2*N + 1 asynch doers
must be active.

asynch doer phil(i):
  server room = room.door, left = fork(i),
         right = fork(succ(i));
   state val mis:{outside, hungry, moving, eating};
  initial :mis': = outside;
  -- variable mis is regarded differently when
  -- appearing on
  -- the two different sides of the "fence" (=>);
   -- e.g. in
  -- the first transition, departing from
  -- mis = outside,
  -- its next value becomes hungry, i.e.
  -- :mis': = hungry; but
  -- next attached to the variable has a meaning
  -- just
  -- within this first transition; for the machine,
   -- as a whole, the current value of the variable
--(here mis) is assigned its next value, i.e.
--mis = hungry;
 < >mis = outside=>THINK or:mis': = hungry;
 < >mis = hungry=> :mis': = moving ,
      service room.enter;
 < >return room.enter=> service left.pickUp,
               service right.pickUp;
 < >return left.pickUp, return right.pickUp=>
               :mis': = eating;
 < >mis = eating =>  EAT or :mis': = moving ,
                    service left.putDown ,
                    service right.putDown;
 < > return left.putDown, return right.putDown
      => service room.exit;
 < > return room.exit => :mis': = outside;
 end doer;
end process;
```

4. Specification Steps in MULTISPEL

Step 1. A concurrent system is specified as a doer. A process description will consist of:

a) a doer (a machine)

b) communicating doers (a system)

c) a combination of a) and b).

Step 2. Faces are specified: names are given to faces, actions, services, pre- and post-conditions are associated with services. Back to step 1, invariants are associated with doer(s) (types).

Step 3. Services defined at step 1 introduce data types, functions, predicates, theorems. All of these are theory-abiding. Theory properties (theorems) are proved. Back to step 1, doer (type) behaviour correctness may be proved (using invariants). Step 3 is repeated for theory refinement.

5. M I E Formal Specification Environment

MIE environment assists user in

- specifying and designing a sequential or concurrent system;
- verifying executable specifications correctness.

MIE aims at making users get accustomed to formal methods and to the way they are used in system development. A formal method-based environment, as for instance MIE, is useful when software production technology and critical projects are involved [8, 12]. Software environments based on formal methods can assist the development of a great deal of critical and semi-critical systems as:

- error detection is enabled at an early stage of the software life -cycle;
- maintenance costs get down drastically because it is no longer the source code which should be maintained but the executable specifications ;
- executable specifications are abstract, so that they should be reused for similar project developments.

MIE can be used for specifying/designing:

- transaction processing-based systems (e.g. components of a telephonic system, a ticket-reservation system, etc.);
- control systems (e.g. systems monitoring aerial communication, road traffic, rail freightage, etc.).

The MULTISPEL executable specification library must be enriched and domain-adapted in order to develop such applications.

User interface with an MIE environment is represented by a central menu with several options:

- File
- Compile
- Run
- Help index
- Quit.

File option covers file generation, editing, and loading of the existing files. The user may temporarily or definitely quit MIE by selecting two other suboptions of File option; he (she) may also quit MIE with Quit option.

Compile option covers lexical and syntactical analysis, and static type-checking of MULTISPEL source code.

Run option covers two aspects:

- formal verification of predicates defined in MULTISPEL executable specifications; therefore, MIE includes OTTER prover [10, 11] of the ARGONNE NATIONAL LABORATORY, Illinois, USA (first developed in 1990 and then extended in 1991);
- simulation of MULTISPEL executable specifications concurrent run.

Here is an example of specifying a MULTISPEL theory defining the "set" notion by means of a lists-defining theory:

theory <EgSet>[ϕ,&,:∈:] listBasicSet <Elem>

-- no unique representation !!!

set Elem, EgSet; **var** E, F: Elem, S: EgSet;

val ϕ, S&E: EgSet;

ax. S&E&E = S&E, S&E&F = S&F&E;

rel E∈S; **inductive def not** :E:, ∈ϕ:E ∈S&F:⇔ E = F **or** E∈S;

th. :E:∈S -- demo "∈" = Member in OTTER input

--file

dem inductive def E∈:S:&:F:⇔ :E: = F **or** :E:∈S, **not** E∈ϕ;

end dem;

let <EgSet, Elem, :rset:, &, ϕ> LRed;

-- rset is a reducer

th. S = rset [rset(E) | :E:∈S]

inductive dem

th. rset [rset(E) | :E:∈∅] = rset [] = ϕ ;

th.

rset [rset(E) | :E:∈S&F] =

rset [rset(E) | :E: = F **or** :E:∈S] =

rset (F) V rset [rset(E) | :E:∈S] =

= rset(F) V S = S&F;

end dem;

end theory;

An example of a small proof session is offered. The first file represents the input file for OTTER hereby a proof of the predicate Member (or "∈" in the above example) is obtained.

```
% Input file for OTTER


set(para_into).
set(para_from).
set(free_all_mem).


list(sos).


(x = EMPTY) | (MakList(First(x),Rest(x)) = x).
(MakList(x, y) != EMPTY).
(Append(EMPTY, y) = y).
(Append(MakList(x,z), y) = MakList(x, Append(z, y))).
-Member(e, EMPTY).
(Member(e, x) = $IF($ID(e, First(x)), $T, Member(e, Rest(x)))).
(x = x).
-Member(e, x).% denial
-Member(e, y).% of
Member(e, Append(x, y)).% conclusion
end_of_list.
list(demodulators).
(Append(EMPTY, y) = y).
(Append(MakList(x,z), y) = MakList(x, Append(z, y))).
(First(MakList(x, y)) = x).
(Rest(MakList(x, y)) = y).
end_of_list.
```

An output file is obtained , a fragment of which is given:

```
---------------- PROOF ----------------
3 [] (Append(EMPTY,y) = y).
9 [] -Member(e,y).
10 [] Member(e,Append(x,y)).
15 [para_into,10,3] Member(e,x).
16 [binary,15,9] .
----------- end of proof -------------
```

Help index option covers some of the most important topics related with formal methods use in specifying concurrent and distributed systems , as well as some MULTISPEL language syntactic and semantic characteristics, as follows:

Communication

Communication in MULTISPEL

Communication structures

Concurrency

Concurrency in MULTISPEL

Concurrent versus sequential programs

Configuration description

Distributed versus concurrent systems

Exception handling

Finite state machine

Finite state machine versus hierarchy of functions

Functional view of a distributed system

Graph model

Hierarchy of functions model

Interleaving view of a distributed system

MULTISPEL actions

MULTISPEL algebraic model

MULTISPEL axioms

MULTISPEL doer

MULTISPEL face

MULTISPEL interpretation

MULTISPEL invariants

MULTISPEL messages

MULTISPEL operational model

MULTISPEL process

MULTISPEL semantic model

MULTISPEL sentence

MULTISPEL service

MULTISPEL signature

MULTISPEL states

MULTISPEL theorems

MULTISPEL theory

LOTOSPHERE Methodology

Mathematical function model

Modularity

Module classes

Module classes in MULTISPEL

Parallel blocks

Petri nets

Process

Process declarations

Real-time concepts

Space - time view of a distributed system

Specification models for concurrent and distributed systems

Synchronization properties

Views of describing a distributed system.

The above listed topics and their corresponding contents text form a (Prolog) database, which can be updated at user's wish, by means of a separate component, and then automatically reintegrated into MIE.

MIE operates as a framework prototype to demonstrate some of the MULTISPEL language characteristics. It has been implemented on IBM-PC AT compatible computers, in Prolog, using MS-DOS operating system and Turbo Prolog 2.0 environment. The so-far developed several versions of MULTISPEL language, needed for facing the complexity of any formal specification language, will be added a new extended version. MIE only supports the specification/design part of a system development. The authors also intend to extend MIE to supporting C programs generation from MULTISPEL source code.

Conclusions

The paper presents a formal specification environment, named MIE, using its own formal, concurrent, executable specification language, named MULTISPEL.

MULTISPEL is a broad spectrum specification language, having both provability and efficiency as specification objectives. The user can specify sequential and distributed systems and verify executable specifications correctness. Some verification steps are built-in and more complex ones (proving steps, as in Unity [9]) must be specified by the user. MULTISPEL differs from LOTOS [5,14] in that it is a logical language; that is, automated verification steps are difficult to impose.

An extended MULTISPEL version is currently under development, in order to better cope with formal correctness proof at doer (type) level and to improve communication structure so as to permit several communication forms (not only client-server, as possible in the current MULTISPEL version).

MIE operates as a framework prototype which demonstrates some of the MULTISPEL language characteristics. MIE does not support requirements specification (as done in lite, LOTOS environment). For the time being the authors will not go further in dealing with this aspect.

A new version of MIE is going to be developed using Top Speed Multilanguage Environment, with built-in concurrency primitives in MS-DOS.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their helpful criticism of the preliminary version of this paper.

REFERENCES

1. BAETEN, J.C.M. and VAN GLABBECK,R.J., **Another Look at Abstraction in Process Algebra**, in Th. Ottmann (Ed.) ICALP'87, LNCS 267, SPRINGER-VERLAG, Berlin, 1987.
2. BEIERLE, C. and VOSS,A., **Theory and Practice of Canonical Term Functors in Abstract Data Type Specifications**, in H.Ehrig et al (Eds.) TAPSOFT'87, Vol. II, LNCS 250, SPRINGER-VERLAG,Berlin, 1987, pp. 320-334.
3. DINESCU, D. M., **A Formal Executable Specification Language**, Proceed. of the Intl.Symp.Ec.Comp. Sci. IE'93, Bucharest, May 1993, pp. 120 - 122.
4. DINESCU, D. M. and RABEGA,I.V., **Software Development Environment for Sequential and Concurrent Systems, Based on Formal Methods**, Proceed. of the Intl.Symp.Ec. Comp.Sci.IE'93, Bucharest, May 1993, pp. 43 - 46.
5. FEREIRA PIRES, L. and VISSERS,C.A., **Overview of the Lotosphere Design Methodology**, Memoranda Informatica 90-64, The Netherlands, October 1990.
6. GOGUEN, J.A. and MESEGUER, J., **Models and Equality for Logical Programming**, in H.Ehrig et al (Eds.) TAPSOFT'87, Vol. II, LNCS 250, SPRINGER-VERLAG, Berlin, 1987, pp. 1-22.
7. GUTTAG, J. V., HORNING,J.J. and WING, J., **Larch in Five Easy Pieces**, DEC Systems Research Center, Palo Alto, CA., July 24, 1985.
8. JONES CLIFF, B., **Software Development, A Rigorous Approach**, PRENTICE-HALL International Inc., London, 1980.
9. KNAPP, E., **An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs**, ACM TRANS. PLS, Vol. 12, No .2, April 1990, pp. 203-223.
10. MCCUNE, W., **OTTER 2.0 Users Guide**, ARGONNE NATIONAL LABORATORY, Math. & Computer Sci. Div. March, 1990
11. MCCUNE, W., **What's New in Otter 2.2**, ARGONNE NATIONAL LABORATORY, Math. & Computer Sci. Div., Tech. Memo. No. 153, July, 1991.
12. PIZZARELLO, A. **Development and Maintenance of Large Software Systems**, Lifetime Learning Publications, Belmont, CA., 1984.
13. POUNTAIN, D. and MAY,D., **A Tutorial Introduction to Occam Programming**, MCGRAW-HILL Book Company, London, 1987.
14. VISSERS, C.A., FEREIRA PIRES,L. and VAN LAGEMAAT,J., - **Lotosphere, an Attempt Towards a Design Culture**, Memoranda Informatica 92-56, TIOS 92-24, Univ. Twente, The Netherlands, August 1992.
15. * * * **RAISE Overview**, CRI A/S, Denmark, 1991.
16. * * ***RAISE, Newsletter** No. 1, CRI A/S, Denmark, Spring 1992.