Autonomy and Robotics Area
Computational Sciences Division
M/S 269-3
NASA Ames Research Center
Moffett Field, CA 94301

Modeling with Livingstone

Autonomy and Robotics Area
NASA Ames Research Center

http://ace.arc.nasa.gov/postdoc/livingstone

DRAFT ONLY – We are redesigning the Livingstone modeling language. This rough document covers the old Lisp-like modeling language used on Deep Space 1 and will likely not be maintained.

Contact person: kurien@ptolemy.arc.nasa.gov

Table of Contents

1 Introduction to MPL 4

1.1 Well-formed Formula (wffs).....4

1.1.1 Finite Domains 4

1.1.2 Constraining Two Quantities to be Equal 5

1.1.3 Review 6

1.2 Component Definition: Defcomponent.....6

1.2.1 Definition.....6

1.2.2 Type and Args 7

1.2.3 :Attributes, :Inputs, :Outputs or :Ports 7

1.2.4 Modes.....8

1.3 Module Definition: Defmodule.....9

1.4 Macros for generating wffs 10

1.4.1 Relation Definition 10

1.4.2 Forall..... 11

1.4.3 The NEXT Operator 12

1.5 Instantiation..... 12

1.5.1 Relations..... 12

1.5.2 Defmodule..... 13

1.5.3 Defcomponent 13

1.6 MPL Syntax 13

1.6.1 Well formed formulae..... 14

1.6.2 Defcomponent 14

1.6.3 Attributes and Transitions 14

1.6.4 Defmodule 15

2 Commands and Monitors 16

2.1 Background 16

2.2 Monitor Values 16

2.3 Commands 17

2.4 Syntax 17

3 A very simple switch model 19

3.1 The Livingstone model and Lisp test function 19

;; Begin lisp code 19

; Defining the finite domains of our attributes 19

; The switch component 19

; Testing the component 21

;; End lisp code 21

3.2 The output from evaluating (turn-on) 21

4 Tools and Hints for working with models 23

4.1.1 Livingstone Debugging Tools 23

4.1.2 Finding things in the Livingstone Image 23

4.1.3 Common errors encountered in Livingstone 24

Introduction to MPL

MPL is the language with which a modeler describes a system to be diagnosed or controlled by Livingstone. MPL is used to specify what are the components of the system, how they are interconnected, and how they behave both nominally and when failed. Component behavioral models used by Livingstone are described by a set of propositional, well-formed formula (*wff*). An understanding of well-formed formula, primitive component types specified through `defcomponent`, and device structure specified by `defmodule`, is essential to understanding of MPL.

This document describes:

- well-formed formula (wff): The basis for describing the behavior of a component in a system
- defvalues: Specifies the domain (legal values) of a variable
- defcomponent: Defines the modes, behaviors and mode transitions for primitive components
- defmodule: Defines composite devices, consisting of interconnected components
- defrelation: A macro mechanism for expanding a complex wff according to the value of an argument
- forall: An iteration construct used to expand a wff or relation on a set of arguments
- defsymbol-expansion: A mechanism for naming a collection of symbols (eg the name of all valves in the system)

1.1 Well-formed Formula (wffs)

Livingstone uses propositional formulae to describe the behavior of components in particular modes. A proposition is a statement that has a truth assignment of true or false. For example, the proposition `"Palo Alto is in California"` is true, and the proposition `"Tokyo is in California"` is false.

While a proposition can be any statement that can be assigned a truth value, propositions in Livingstone are generally statements about the value of some variable. The variable is in turn some attribute of a device in the system being modeled. The statements "the flow through the valve is low" or "the power switch reports that it is on" are examples. A variable in MPL is of the form:

variable ::= (*attribute component*)

where *component* is the name of some component or module in the model (discussed below) and *attribute* is the name of an attribute of that component or module. If we have a component named `hydrogen-valve` and it has been created to have an attribute called `flow`, we would refer to the variable representing the flow through this valve as:

```
(flow hydrogen-valve)
```

1.1.1 Finite Domains

Livingstone is based upon propositional logic. It does not know about integers, real numbers or arithmetic but instead works with finite domains. A finite domain specifies a finite number of values for a variable. Quite often, each value in the finite domain represents a range of values on the real number line. For example, the flow through the hydrogen valve may be low, nominal or high. Other finite domains may represent truly discrete values. A switch position sensor returns only the values on and off. A new finite domain can be introduced with the form

(`defvalues` *domain-name values*)

where *domain-name* is the name of the new finite domain and *values* is a list of the values within a domain. For example, to model flows as low, nominal or high, one could use the form

```
(defvalues flow-values (low nominal high))
```

This specifies to Livingstone that any variable whose type is flow-values can take on the values low, nominal or high. In fact this will specify that the variable must have exactly one of these values at all times, though at any given time there may not be enough information to determine which value that is. When defining a component, one can specify the name and types of the attributes in a form such as:

```
(defcomponent valve (?name)
(:attributes
((flow ?name) :type flow-values)))
```

This form, which is discussed in detail below, specifies that the valve named *?name* has an attribute named `(flow ?name)` and it's domain is `flow-values`. If we create a valve named `hydrogen-valve`, then the variable `(flow hydrogen-valve)` can take on the values low, nominal and high. In the language of propositions, this means the following propositions have been created, and can be assigned a truth value: "`(flow hydrogen-valve)` is low", "`(flow hydrogen-valve)` is nominal", "`(flow hydrogen-valve)` is high"

Internally, Livingstone represents the proposition "`(flow hydrogen-valve)` is low" as (low (flow hydrogen-valve)). For a number of reasons, including that this is hard to read, one should use the form:

```
(= (flow hydrogen-valve) low)
```

It's critical to keep in mind that this is a proposition, which may be true or false, and not an assignment, which actually changes the value of the variable representing the flow. Propositional formulae are constructed by combining propositions using the logical connectives such as `AND`, `OR` and `NOT`. For example, the logical formula

```
(OR (= (flow hydrogen-valve) nominal) (= (flow hydrogen-valve) high))
```

is true as long as the value of the flow is not low. The actual value of the flow variable is not affected by evaluating the truth of this formula.

Below is the syntax of a propositional, well-formed formula or wff ("woof"),:

```
wff ::= (AND wff+)
          (OR wff+)
          (NOT wff)
          (IMPLIES wff wff)
          (IFF wff wff)
          (WHEN wff wff)
          (UNLESS wff wff)
          (IF wff wff wff)
          proposition
```

Below are some examples of wffs. Since `(flow hydrogen-valve)` must have exactly one value at a time, the following wffs are always true. Once one has introduced more than one variable, substantially more interesting wffs can be created to describe how a device behaves.

```
(OR     (= (flow hydrogen-valve) low)
        (= (flow hydrogen-valve) nominal)
        (= (flow hydrogen-valve) high))

(implies (= (flow hydrogen-valve) low)
        (not (= (flow hydrogen-valve) nominal))
```

1.1.2 Constraining Two Quantities to be Equal

Suppose one has a hydrogen tank, a hydrogen valve and a rocket engine, all connected in series. Neglecting the possibility of leaky pipes for the moment, the flow through all components will be equal. If each of the flows has the domain flow-values, then we could constrain the valve and tank flows to be equal by writing a wff of the following form. The connector iff is a standard abbreviation for "if and only if".

```
(and (iff (= (flow hydrogen-valve) high) (= (flow hydrogen-tank) high)
     (iff (= (flow hydrogen-valve) low)  (= (flow hydrogen-tank) low))
     (iff (= (flow hydrogen-valve) nominal) (= (flow hydrogen-tank)
nominal)))
```

This wff specifies that if the value of one variable is high the other must be high, and so on. If for example the flow of the hydrogen tank was measured from the environment, that would constrain the flow at the hydrogen valve. To avoid the need to type out and maintain such long formulae, Livingstone allows the = operator to be applied to two quantities such as `(flow hydrogen-valve)` and `(flow hydrogen-tank)`. The wff

```
(= (flow hydrogen-valve)(flow hydrogen-tank))
```

expands to the wff shown above, which constrains the two quantities to have the same value. Including =, the full wff syntax is

```
wff ::= (AND wff+)
           (OR wff+)
           (NOT wff)

           (IMPLIES wff wff)
           (IFF wff wff)
           (WHEN wff wff)
           (UNLESS wff wff)
           (IF wff wff wff)
           (= variable value)
           (= variable variable)
```

1.1.3 Review

- The devices in a system are modeled with components. How components are created is discussed below
- A component has attributes
- Each attribute has a domain that specifies the values the attribute may take on
- A new domain is created with defvalues
- The statement that an attribute has a certain value is a proposition, which may be true or false
- Propositions may be combined into formulae using logical connectors such as OR and IMPLIES. These formulae describe the behavior of a device
- = can be used to represent the proposition "this variable has this value"
- = can be used to constrain two quantities to be equal

1.2 Component Definition: Defcomponent

Livingstone models component behavior in terms of a set of modes and transitions between these modes. `defcomponent` is used to provide a specification for the behavior of a class of components that can be instantiated according to the device that is being modeled. This section first defines component definition and follows with component instantiation.

1.2.1 Definition

Defcomponent specifies creates a new type of component. It specifies the *attributes* and attribute types, its *modes*, its *behavior* within modes, and its *transition* behavior between modes for all components that are created of this type. In general the syntax of a DEFCOMPONENT definition is as follows:

```
(defcomponent type (args)

  [(:attributes attribute-spec* )]
  [(:inputs attribute-spec*  )]
  [(:outputs attribute-spec* )]
  [(:ports attribute-spec* )]

  [(:declarations wff)]

  [(:background
     [:model wff]
     [:initial-mode mode-name]
     [:initially wff])]

  (mode-name
     :type <:ok-mode | :fault-mode>
     [:probability number]
     [:value number]
     [:model wff]
     [:transitions <transition-spec>])*
   )
```

See appendix MPL BNF for definitions of [], *, | and +

1.2.2 Type and Args

type is a symbol naming the new component type. For example, this could be valve or power-distribution-unit.

args is a list of arguments. When a component of this type is created, actual values will be provided for the arguments and substituted into the formulas in the component definition. By convention, the names of the arguments begin with "?" and the first argument is "?name", the name of the component being created. For example, the following code

```
(defcomponent heater (?name)
                    ...)
```

defines a component of *type* heater with the argument ?name. When a heater is created, the user will specify a value for ?name which will be substituted into the component definition. The argument ?name in particular will be used in the attributes to ensure the propositions we introduce about this heater's attributes are named for the heater.

1.2.3 :Attributes, :Inputs, :Outputs or :Ports

All of these fields declare the attributes of the component as well as the domain of each attribute. Any combination of *:attributes, :inputs, :outputs* or *:ports* may be used. They are synonymous and the distinction is only provided for clarity. Each attribute field takes a list of attributes definitions. Each definition names the attribute and specifies its domain:

Attribute_def ::= `((`*attname* `?name) :type` *domain-name*`))`

An example is

```
((power-input ?name) :type on-off-values))
```

Note that the name of the attribute is (flow ?name), where ?name will be substituted when a component of this type is created. See section Defvalues for using defvalues to define attribute types

and their values. To define the valve component type to have a flow attribute and a pressure attribute, the following form could be used:

```
(defcomponent heater (?name)
      (:attributes
          ((power-input ?name)
                   :type on-off-values
                   :documentation "Is the heater getting power?")

          ((reset-cmd ?name)
                   :type reset-cmd
                   :documentation
                          "Command to reset the circuit breaker")

          ((thermal-output ?name)
                  :type sign-values
                  :documentation "Is the heater heating?"))
...)
```

this wff will define the expressions that can appear as propositions within the wffs that model the component's behavior.

1.2.3.1 :Declarations

Define declarations section

1.2.3.2 :Background

Define background section

1.2.4 Modes

mode-name is a symbol and must be one of the modes defined for the component. As indicated by the *, a component can have 0 or more modes. The modes of the component will be partitioned into two types of modes. :ok-mode defines the different *nominal* modes of the component. :fault-mode on the other hand defines the *non*-nominal or failure modes of the component. Each failure mode has a prior *probability* associated with failing into that mode.

Each mode has a behavior, which is enforced as long as the component is in that mode. This behavior is specified by a wff in the :model and usually will use some of the constructs identified in section WFF.

For example, the heater component is a component of two modes. The first mode: *ok* is a normal operating mode of the component and is modeled by the following code:

```
(defcomponent heater (?name)
      (:attributes     ...)
      (on
         :type :ok-mode
         :documentation "working -> produces desired heat output"
         :model (if (=(power-input ?name) on)
                       (= (thermal-output ?name) positive)
                       (=(thermal-output ?name) zero)))
          ...)
```

The ok mode can be seen by the keyword `:type`, with value ok-mode. The behavior of this mode is specified by a wff in the `:model` field. For example, in the ok mode the wff:

```
(if (= (power-input ?name) on)
    (= (thermal-output ?name) positive)
    (= (thermal-output ?name) zero))
```

says that if the heater is on it produces positive heat; otherwise, it produces zero heat. The second mode of the heater, a failure mode named *unexpectedly-off* can be seen in the following code:

```
(defcomponent heater (?name)

      (:attributes     ...)

      (ok              ...)

      (blown-circuit-breaker
       :type :fault-mode
       :documentation "Blown internal circuit breaker."
       :model (zero (temperature (thermal-output ?name)))
       :transitions ((reset-cb
                      :when (on (reset-cmd ?name))
                      :next ok
                      :cost 1)))
          ...)
```

The failure mode can be seen by the keyword `:type`, with value :fault-mode has a `:probability` of .00005 which together with the probability of the ok-mode sums to 1. As in the previous example, the behavior of this mode is specified by a wff in the `:model` field. And as can be seen is the simple statement:

```
(zero (temperature (thermal-output ?name)))
```

Finally, a model specifies conditions under which a component may transition from one mode to another. For example, the mode unexpectedly-off, has one transition, which is called on-resets:

```
:transitions ((reset-cb
               :when (on (power-input ?name))
               :next ok
               :cost 1))
```

This transition specifies that if the heater is in the blown-circuit-breaker mode and reset-cmd on, then the heater transitions to the ok mode. Transitions can incur a cost; in this example the cost is 1. In addition to specifying transitions, a component can spontaneously transition to a failure mode. For example, the heater may spontaneously transition from ok to blown-circuit-breaker. However, it can only transition from a failure mode to an okay-mode through one of the explicitly declared transitions.

1.3 Module Definition: Defmodule

The DEFMODULE is a macro used to define the structure of a collection of components. Like DEFCOMPONENT, DEFMODULE is used to define a generic type of a module, i.e., a specification for how a parameterized set of components should be hooked up. The syntax for the DEFMODULE forms is the following:

```
(defmodule <module-type-name> (?NAME <other-parameters>)
  (:structure
   (type . args)*
   )
  [(:facts wff )]
  )
```

where *device-type-name* is an symbol denoting new device type. *parameters* is a list of symbols, each beginning with "?", that parameterize this definition. By convention, the first parameter is always ?NAME. This parameter will be used to specify the name of each module of this type that is instantiate.

The following examples creates a latched-thruster module, which can be turned on or off.

```
(defmodule latched-thruster (?NAME)
  (:structure
   (thrust-valve (thrstr-valve ?NAME))
   (thruster (thrstr ?NAME))
   (flow-connection (fuel-input (thrstr ?NAME))
                    (output (thrstr-vlv ?NAME)))))
```

The module's structure is specified by a wff in the `:structure` field. The latched-thruster is defined by specifying three components defined using defcomponent.

The first two defined components: a thruster-valve and a thruster correspond to primitive components, while the third component flow-connection corresponds to the physical lines that connect the two primitive components. This third component is also defined using defcomponent. The behavior of a module consists of the conjunction of the behaviors (wffs) produced by the components and connections in `:structure`. Added to this conjunction is additional behavior for that module specified by a wff in the `:facts` field.

1.4 Macros for generating wffs

There are three macros that make it easy to specify wffs: defrelation which defines new relations, defvalues which defines new types, and forall, which allows wffs to be constructed through iteration.

1.4.1 Relation Definition

Two constructs are provided to allow the user to construct relations in LMPL. The first construct is the `defvalue` macro which defines a new unary relation that ensures that its argument has exactly one of a discrete set of values. The second construct is the `defrelation` macro which is used to define schemas that instantiate to the well-formed formula in the body of the defrelation. It is used to define binary predicates that order values constructed via defvalues. In addition it used to define procedural constructs such as implication. We believe that a liberal use of defrelation is essential to good modelling.

1.4.1.1 Defvalues

The syntax of defvalues is as follows:

```
(defvalues domain-name values)
```

where *domain-name* is a symbol denoting the name of the relation that defines the domain consisting exactly of values and *values* is a list of symbols, denoting predicates specifying allowed values within that domain. For example,

```
(defvalues sign-values (negative zero positive))
```

defines the unary relation of name sign-values that ensures that any variable of type sign-values will have exactly one of the values positive, negative or zero. The use of these relations to type declare variables required for component behavior definition can be seen in Section \ref{Def:component}. The instantiation of a type declaration can be seen in Section \ref{instantiatedefvalues}.

1.4.1.2 Defrelations

The defrelation macro allows the user to define a macro that expands into a WFF. The syntax of a DEFRELATION definition is as follows:

```
(defrelation relation-name parameters
        wff)
```

where *relation-name* is a symbol denoting the name of the relation, *parameters* is a list of symbols, each beginning with "?", that parameterize this definition and *wff* is a well formed formula as defined above. For example, the implication relation, (A implies B), is equivalent to the wff (not A or B) . Implication is defined using defrelation as follows:

```
(defrelation IMPLIES (wffa wffb)
      (OR (NOT wffa) wffb))
```

The equivalence relation IFF can then be defined from the implication relation as follows:

```
(defrelation IFF (wffa wffb)
      (OR (NOT wffa) wffb))
```

1.4.2 Forall

The above constructs simplify the modeling process and support easy plug and play by providing mechanisms for modularizing pieces of formula (through defrelations and defvalues), as well as structure and data abstraction (through defmodule).

In addition, there are often a variety of devices each substantially different, but which are assembled together through common pieces. For example the behaviors of Cassini's drivers MPD, MEVD, BPLVD, and HeLVD, are similar in that they drive a common class of devices -- valves, heaters, sensors -- although they each drive a different mixture and number of these devices. The foreach construct makes it easy to automatically assemble together these common pieces of behavior into a single device.

For example, to generate the MPD, MEVD, BPLVD, and HeLVD we can define a component type propulsion-driver, whose parameter values specify the different devices being driven, according to type.

```
(defcomponent Propulsion-Driver
    (?name ?latch-valves ?valves ?heaters ?rcs-branches
        ?sensed-temperatures ?sensed-positions)
  (:attributes
       (and (forall ?latch-valve in ?latch-valves
            (driver-latch-valve-attributes ?name ?latch-valve))
        ....)))
```

Here the parameter ?latch-valves binds to a list of valves. The forall construct instantiates the appropriate wffs for each valve specified on the list. To do this ?latch-valve is successively bound to each element of the list denoted by ?latch-valves. The body of the forall is instantiated for each binding of ?latch-valve, producing a wff. What is generated is the conjunction (AND) of these wffs.

Next a specific driver type, such as the MPD, is easily assembled as an instance of propulsion-driver.

```
(defmodule MPD (?name ?lv ?cbh ?rcs-branch ?cbh-temp)
  (:structure
   (propulsion-driver ?driver-name
                      (?lv)
                      ()
                      (?cbh)
                      (?rcs-branch)
                      (?cbh-temp)
                      ()))))
```

The syntax of `forall` is as follows.

```
(forall <parameter> <list-parameter>)
```

wff)

where:

list-parameter is a symbol, beginning with "?", that is bound to a list of expressions.

parameter is a symbol, beginning with "?", that is bound to successive elements of list-parameter.

wff is a well-formed formula as defined above, involving parameter.

1.4.3 The NEXT Operator

Finally the connective NEXT is provided that allows the system to assert propositions in the future.

wff ::= (NEXT *proposition*)

For example the wff,

```
(implies (and (off driver)
              (on (power-command driver)))
    (next (on driver)))
```

states that if the driver is off and an on power command is detected, in the *next* state the driver will be on. Although this connective applies to any propositions it is used primarily to track device state over time.

This is a particularly bad example, because if there were a failure transition at this point which was inconsistent with (on driver) you would be screwed by an inconsistent state. There are better examples of why you'd really need to use next.

This capability is used during mode identification and recovery as Livingstone analyzes the behavior of a system at successive states in time. At each step its analysis encompasses the current state a device is in, and its successor state.

No mention of this capability so far in paper. Is it described in another document that should be referenced?

1.5 Instantiation

As can be seen in Figure 1, once the library has been defined for the domain, models of the domain can be instantiated. Hence for every construct available during library construction there exists a mapping to a set of wffs that are then available to the theorem prover. This section is partitioned by library construct and describes the process of instantiation for each construct from library -> model -> wff.

1.5.1 Relations

Both of these constructs, defvalue and defrelation define a mechanism for encapsulating and reusing wffs, similar to the use of macros in functional programming.

1.5.1.1 defvalues

Given the same variable type declaration example as section \ref{def:defvalues

```
(sign-values (temperature heater))
```

where the unary relation sign-values allows temperature to take on one of the propositions

- (positive (temperature heater))
- (zero (temperature heater))
- (negative (temperature heater))

such that no two of these propositions hold simultaneously, the instantiation of this construct will be:

```
(and  (or  (= positive (temperature heater))
           (= zero (temperature heater))
           (= negative (temperature heater)))
      (not (and (= positive (temperature heater))
                (= zero (temperature heater))))
      (not (and (= negative (temperature heater))
                (= zero (temperature heater))))
      (not (and (= negative (temperature heater))
                (= positive (temperature heater)))))
```

1.5.1.2 defrelations

When the models are instantiated any instances of relations are replaced an instance of the wff in the defrelation body. For example, the procedural constructs supported by MPL provides the following mappings between relations and wffs.:

```
(IMPLIES wffa wffb)  equivalent to  (OR (NOT wffa) wffb)

(IFF wffa wffb)      equivalent to  (AND (IMPLIES wffa wffb)
                                         (IMPLIES wffb wffa))

(WHEN wffa wffb      equivalent to  (IMPLIES wffa wffb

(UNLESS wffa wffb    equivalent to  (IMPLIES (NOT wffa) wffb
(IF wffa wffb wffx)  equivalent to  (AND (IMPLIES wffa wffb)
                                         (IMPLIES (not wffa) wffc))
```

1.5.2 Defmodule

A defined module can be instantiated with specific values for its arguments using the function INSTANTIATE-MODULE.

1.5.3 Defcomponent

An instance of a component is denoted by an expression of the form (*type arg-values*).

For example, the expression

```
(valve-driver rcs-latch-valve branch-a)
```

Creates an instance of component type valve-driver, enforcing the definition specified by the corresponding defcomponent. When instantiating a component each parameter in a wff is replaced with its corresponding value. For example, if the valve-driver defcomponent includes the wff:

```
(implies (= closed (?name ?index))
   (zero (= out-flow (?name ?index))))
```

then the above driver instance produces the wff:

```
(implies (= closed (rcs-latch-valve branch-a))
   (zero (= out-flow (rcs-latch-valve branch-a)))).
```

1.6 MPL Syntax

In the following, items within square brackets are optional, * denotes 0 or more occurrences of the previous regular expression, + denotes 1 or more occurrences of the previous regular expression, and items in angle brackets are explained further.

1.6.1 Well formed formulae

```
proposition ::= (= value variable)
                  (= variable value)
                  (value variable)      ; This is shorthand for (= value variable)
variable ::= (attribute component).
wff ::= (AND wff+)
            (OR wff+)
            (NOT wff)
            proposition
           (IMPLIES wff wff)
           (IFF wff wff)
           (WHEN wffwff)
           (UNLESS wffwff)
           (IF wffwffwff)
           (= variable variable)
           (NEXT proposition)
```

1.6.2 Defcomponent

```
(defcomponent component-type (?NAME remaining-parameters…)

  [(:attributes <att-spec>* )]
  [(:inputs <att-spec>*)]
  [(:outputs <att-spec>*)]
  [(:ports <att-spec>*)]

  [(:declarations <att-spec>*)]

  [(:background
     [:model wff]
     [:initial-mode mode-name]
     [:initially wff])]

  (mode-name
     :type <:ok-mode | :fault-mode>
     [:probability number]
     [:model wff]
     [:transitions <transition-spec>])*
 )
```

1.6.2.1 Additional notes

By convention, the first parameter of a component or module type is ?NAME. The argument supplied for this parameter during instantiation is the name of the resulting component or module instance.

The *wff* associated with :initially is asserted into the theory at the beginning. However, when a transition is executed, these clauses are removed, and new clauses have to be added in using (the infamous!) frame axioms.

1.6.3 Attributes and Transitions

`att-spec` ::= ((?NAME *att-name*) :TYPE *defvalue-type* [:DOCUMENTATION *string*])

transition-spec ::= (*transition*+)
transition ::= (<trans-name> [:when *wff*] :next *mode-name*
[:assert *wff*]
[:cost *number*])

1.6.3.1 Shorthands

- :persist can be used in place of the :next clause. :persist is the same as :next <current-mode>
- :otherwise can be used instead of the <name> and the :when clause :otherwise is the same as :when (not (or <other-conditions>)) The <name> can be omitted only when :otherwise is used
- If the :transitions clause as a whole is omitted from the component definition, it defaults to `((:otherwise :persist))`
- If the :when clause is omitted, the *wff* defaults to true If the :cost clause is omitted, the *number* defaults to 0

1.6.4 Defmodule

The syntax for the DEFMODULE form is the following:

```
(defmodule module-type (?NAME other-parameters…)
  (:structure   (component-type args)*  (module-type args)*     )
  [(:documentation string)]
  [(:facts wff)]
  [(:connections wff)]
  [(:inputs att-spec*)]
  [(:outputs att-spec*)]
  [(:input-declarations att-spec*)]
  [(:output-declarations att-spec*)]
  )
```

where :

- *parameters* is a list of symbols, each beginning with "?", that parameterize this definition
- *component-type* or *module-type* name a previously defined component or module type
- *args* is a list of symbols that define the particular instance of the component/module type that should be instantiated as a part of an instance of this module type. In a module definition, the symbols may also be variables that occur in the *parameters* list

Commands and Monitors

1.7 Background

Every livingstone model has a certain number of attributes that are inputs and a certain number of outputs. An input might represent a command given to a component, such as a command for a switch to turn on, and an output might represent some continuously observable characteristic of the system, such as whether the switch's indicator lamp is on or off. In a physical system a command occurs at a point in time, and a monitored characteristic typically persists for some indefinite interval. Livingstone's model of the world is similar, as illustrated by a switch example.

Consider a switch in some state, for example turned-off, with its observable lamp output persisting in the off condition. A command to turn on is given at a point in time, and the switch most likely transitions to the turned-on state. We might then observe that the lamp now persists in the on condition, reinforcing our belief that the switch transitioned to the turned-on state, rather than failing in some way.

In its propositional representation of a device model Livingstone follows roughly the same sequence of events. The model is in some state, represented by Livingstone's current theory. The theory contains propositions for the device model, the fact that the lamp is observed to be off, and so on. Giving a command is represented by inserting a new proposition of the command value into the theory. When a command is inserted, any old observations are retracted and Livingstone uses the updated theory to determine which mode transitions may have occurred. The command is then retracted as it represents an event, not a persistent condition. The new observations are inserted into the theory, and Livingstone checks whether they are consistent with the inferred nominal transition, or if some failure transition may have occurred.

In order to automate the process of asserting and retracting propositions to simulate commands and monitored outputs during Livingstone diagnosis, monitor and command constructs are provided.

1.8 Monitor Values

The function `do-monitors` is used to conveniently assert that a model attribute has some value and to cause Livingstone to compute the likely diagnosis and mode transitions that result.

A monitor also ensures that no more than one of an output’s mutually exclusive values is asserted into the Livingstone theory at once and that an asserted value persists until explicitly replaced. Thus, if a new output value is asserted with `do-monitors`, any existing value is automatically retracted first to avoid an inconsistency. This new value persists until `do-monitors` or `do-cmd` (see below) is used to replace it. The value UNKNOWN is treated specially, and indicates that no value for the attribute should be asserted

If our model has an output attribute `(indicator-lamp switch1)`, we can specify a monitor as follows:

```
(new-monitor-table)
(new-monitor '(indicator-lamp switch1))
```

`new-monitor-table` sets up the system to create monitors, and should be called before any monitors are created.

`new-monitor` creates a new monitor for an attribute of a model. If no arguments other than the attribute are given, Livingstone will set up a monitor that allows the attribute to be set to any of its legal values and unsets every value when the monitor receives the value UNKNOWN. When the monitor is created, no value of the attribute will be asserted, as unknown will be the default.

We may set the value to on with: `(do-monitors '((on (indicator-lamp switch1))))`

This will assert the proposition `(on (indicator-lamp switch1))` into the theory and cause Livingstone to look for any mode transitions that are indicated. This value will persist until another value is asserted, perhaps with `(do-monitors '((off (indicator-lamp switch1))))`. Note that Livingstone's diagnoses are persistent. That is, once Livingstone diagnoses that a component is in a mode, it will make further diagnoses assuming the component started in the previously diagnosed mode. A "fresh" copy of each component, in its initial mode, can be had by recreating the component or using a checkpoint.

1.9 Commands

`do-cmd` is used to simulate a device receiving a command and then displaying some monitored output. It asserts the command values and monitored output values and asks Livingstone to perform diagnosis, all in the appropriate order. A command ensures that no more than one of an input's values is asserted into the theory at once. However, unlike a monitor, the asserted value is not persistent. To capture that a command is an event rather than a condition, the command value is asserted while Livingstone is computing mode transitions, then reset to a default value. Livingstone assumes the default value is NO-COMMAND, so model attributes that are to be used as commands should have NO-COMMAND in their domains.

If our model has an input attribute `(command-in switch1)` with values turn-on, turn-off and no-command, we can specify a command as follows:

```
(new-command-table)
(new-cmd '(command-in switch1))
```

`new-command-table` sets up the system to create command, and should be called before any commands are created. `new-command` creates a new command for an attribute of a model.

We may set the command to turn-on with: `(do-cmd '(command-in switch1) 'turn-on)`

This will momentarily assert that `(command-in switch1))` has the value `turn-on` and ask ivingstone to look for any mode transitions that are indicated.

However, this is usually **not** how we would like to use `do-cmd`. Note that we have not specified any new monitor values, so Livingstone will perform diagnosis assuming that the command was given and no monitored output values changed. To specify a set of monitor values that change after the command is given, a list of monitors and values can be passed to `do-cmd`:

```
(do-cmd '(command-in switch1) 'turn-on '(on (indicator-lamp switch1)))
```

This asks Livingstone to report the probable mode changes given that the command input `(turn-on (command-in switch1))` was given, and the value of `(on (indicator-lamp switch1))` became `on` as a result. Note that Livingstone's diagnoses are persistent. That is, once Livingstone diagnoses that a component is in a mode, it will make further diagnoses assuming the component started in the previously diagnosed mode. A "fresh" copy of each component, in its initial mode, can be had by recreating the component or using a checkpoint.

1.10 Syntax

Note that this is the syntax for the simplest usage of `do-cmd` and `do-monitors`.

```
(new-monitor-table)
(new-monitor attribute)
(do-monitors new-monitor-proposition-list)

(new-command-table)
(new-command attribute)
(do-cmd attribute value new-monitor-proposition-list)
```

new-monitor-proposition-list = (*monitor-value-proposition**)

monitor-value-proposition = (*value attribute*)

attribute = (*attribute-name object*) generally speaking

If we would like to create a command with a different default value than NO-COMMAND, we can use the following more complex form of new-command:
(new-command *attribute legal-values default-value*)

If we would like to create a monitor with a different initial value than UNKNOWN, we can use the following form:
(new-monitor *attribute legal-values initial-value*)

A very simple switch model

1.11 The Livingstone model and Lisp test function

The switch component models a switch with an indicator lamp. The switch has two nominal modes, on and off. In the off mode, the indicator lamp is off, and in the on mode it is on. When in the on mode, commanding the switch off turns it off When in the off mode, commanding the switch on turns it on If the switch has been commanded on and the lamp is off, or vice versa, the switch must be broken.

If you'd like to run this sample, you can copy everything from **;; Begin lisp code** to **;; End lisp code** out of this window and into a file. Load the file into a Livingstone session, then invoke `(test-switch)`

;; Begin lisp code

; Defining the finite domains of our attributes

```
;; This will allow Livingstone to infer that if an attribute  which is an on-off-value cannot be on in a given
;;  situation, it must be off, and so on.

;;; An on-off-value such as lamp output has one of two values
(defvalues on-off-values (on off))

;;; An on-off-command like switch command has one of three values
(defvalues on-off-command (on off no-command))
```

; The switch component

```
;; This component captures the behavior of the switch given its current
;; mode, the command input and the value of the indicator_lamp observed.
;;
;; To use the model, we will tell Livingstone the command given and the
;; observation at the indicator lamp, and it will infer the most likely
;; consistent mode for the switch. We specify these values by asserting
;; values for (command-in ?name) and (indicator_lamp ?name) The functions
;; make-command and make-monitor will help.
;;
;;*****************************************************************************
(defcomponent switch (?name)
  (:documentation "An example switch  with a failure mode")

  (:inputs
    ((command_in ?name)        :type on-off-command   :documentation "Cmd to turn switch on or off"))

  (:outputs
    ((indicator_lamp ?name)     :type  on-off-values     :documentation "Observed value"))

  (:background
                                ;; The mode starts at OFF and is updated by each mode diagnosis
                               ;; Livingstone makes.
   :initial-mode off)

  (on     :documentation  "The ON mode.  If switch is on, indicator lamp is on.
```

Note that the following ways of saying this are equivalent:
(on (indicator_lamp ?name)) OR (= (indicator_lamp ?name) on)
Use whichever you find less confusing."

```
    :model (= (indicator_lamp ?name) on)
    :type :ok-mode
    :transitions ((turn-off
                          :when (= (command_in ?name) off)
                          :next off)
              (:otherwise :persist)))

 (off    :documentation   "The OFF mode.  If switch is off, indicator lamp is off.
                          If we get an on command, we nominally move to on state."
    :model (= (indicator_lamp ?name)  off)
    :type :ok-mode
    :transitions ((turn-on
                          :when (= (command_in ?name) on)
                          :next on)
               (:otherwise :persist)))

 ;; If no transition above results in a state consistent with
 ;; the value of the command and indicator lamp, we take a
 ;; failure transition to the broken state.

 (broken   :documentation          "This fault mode makes no predictions about the indicator
                                   lamp.  So, it will be used if we turn the switch on and
                                   the lamp is off, or we turn the switch off and the
                                    lamp is on, neither of which is consistent with the
                                   modes above."
  :type :fault-mode
  :probability 0.01
  :transitions    ((:otherwise :persist)))) ;; Once we decide the switch is broken, it stays broken

; A module which will instantiate the component
; A module is a convenient way of instantiating one or more components which are related.
; Instantiating a module will  create all components in its :structure field, here just
; a switch with name ?name
(defmodule switch-module (?name)
   (:structure
    (switch ?name)))

; A function to instantiate the module
; Calling this function will create a switch component called
; switch1. It will also create a command for
; the attribute (command-in switch1) and a monitor
; for the attribute (indicator-lamp switch1).
;
; We will later use the function do-cmd to specify
; the command given and lamp observation for the switch, so
; Livingstone can determine the switch's mode.
;
; See the section on Commands and Monitors

(defun create-switch()
  ; create a switch module, which will create a switch
```

```
  ; component named switch1
  (instantiate-module '(switch-module switch1))

  (new-command-table)
  (new-monitor-table)

  ;; Create a command which will assert a value
  ;; for (command-switch switch1), allow the switch
  ;; transition to be inferred, then reset to
  ;; (command-switch switch1) to no-command, the default

  (new-command '(command-in switch1))

  ;; Create a monitor which will assert persistent
  ;; values for (indicator-switch switch1)

  (new-monitor '(indicator-lamp switch1)))
```

; Testing the component

```
;We turn the switch on, and the lamp goes on. We turn the switch
;off and the lamp goes off. Finally, we turn the switch on and the
;lamp stays off, a failure, then we turn the switch off

(defun test-switch ()
  (format t "Creating switch...~%")
  (create-switch)
  (format t "Giving command on, with indicator lamp turning on~%")
  (do-cmd  '(command-in switch1) 'on '( (on (indicator-lamp switch1))))

  (format t "~%~%Giving command off, with indicator lamp turning off~%")
  (do-cmd  '(command-in switch1) 'off '( (off (indicator-lamp switch1))))

  (format t "~%~%Giving command on, with indicator lamp still off~%")
  ;; Note we do not need to specify monitor value, since old value persists
  (do-cmd  '(command-in switch1) 'on )

  (format t "~%~%A broken switch stays broken in our model~%")
  (format t "Giving command off, with indicator lamp still off~%")
  ;; Note we do not need to specify monitor value, since old value persists
  (do-cmd  '(command-in switch1) 'off))
```

;; End lisp code

1.12 The output from evaluating (turn-on)

```
TP(33): (turn-on)
Creating switch...
Giving command on, with indicator lamp turning on

Commanding (COMMAND-IN SWITCH1) to ON,
 Consequently the following modes just transitioned:
   SWITCH1 : OFF - ON
```

Giving command off, with indicator lamp turning off

Commanding (COMMAND-IN SWITCH1) to OFF,
Consequently the following modes just transitioned:
SWITCH1 : ON - OFF

Giving command on, with indicator lamp still off

Commanding (COMMAND-IN SWITCH1) to ON,.
(<(BROKEN)>)
Consequently the following modes just transitioned:
SWITCH1 : OFF -> BROKEN

>A broken switch stays broken in our model
>Giving command off, with indicator lamp still off
Commanding (COMMAND-IN SWITCH1) to OFF,
Consequently the following modes just transitioned: none
NIL

Tools and Hints for working with models

1.12.1 Livingstone Debugging Tools

- why
  The function `(why proposition)` explains why a proposition has a truth value or why it has no truth value. This function is preferred to explain-current (see below) because it is interactive. After invoking the function type :h for help or :q to quit.

- *debug-level*
  The *debug-level* can be set to 0,1,2 or 3 and determines the amount of debugging output Livingstone gives while determining the state of the system. At level 0, only the diagnosed mode transitions are reported. At level 3, additional information such as which modes were examined and rejected is printed. *debug-level* may be modified at any time with `setf`

- pvalues ("parameter values")
  The function `(pvalues attribute)` queries the Livingstone current propositional theory for the value of a parameter, or attribute in the MPL language. For example, if you have instantiated a component named switch1 with an attribute `(command-in switch1)`, then `(pvalue '(command-in switch1))` returns the current value of that attribute.

- cprops ("see props")
  The function `(cprops devicename)` will display the propositions involving the device without requiring you to specifically enumerate them as pvalues does.

- explain-current
  The function `(explain-current)` can be used to explain the current contradiction in Livingstone's propositional theory, as in the case where a command inconsistent with the current state has been asserted. This is extremely useful in tracking down an inconsistency to the portion of a model which is responsible.

  When `(explain-current)` is invoked, it displays a proposition that could not be satisfied, such as the inconsistent command that was asserted. Then, it displays the contradiction with the unsatisfied clause, along with its support, or reasons why it must be true. The support of the support is displayed, until all support is reduced to true propositions, such as the current mode of the model.

  If there is no inconsistency, then `(explain-current)` simply sets up the current candidate diagnosis for more detailed explanation via `explain-prop`

- explain-prop
  The function `(explain-prop proposition)` displays the current truth value of a proposition and an explanation for why the proposition has that truth value.

  For example, if you have a component named switch1 with an attribute `(command-in switch1)`, then `(explain-prop '(off (command-in switch1)))` will report if (off (command-in switch1) is true or false, that is, whether or not (command-in switch1) has the value off. In either case, an explanation for the truth value is also printed.

  Note that `explain-prop` will not produce explanations unless `explain-current` has been used to set up the explanations for the current diagnosis first.

1.12.2 Finding things in the Livingstone Image

1.12.2.1 Printing all items of a given type:

- print-all-modes (&key (system *system*))

- print-all-cmds (&key (system *system*))
- print-all-monitors (&key (system *system*))
- print-monitor-values ()
- print-waiting-monitors ()

1.12.2.2 Finding specific items by name

- find-check-point (name &optional (ht *check-point-ht*))
- find-command (name &optional (system *system*))
- find-component (name &optional (system *system*))
- find-component-by-short-name (name &optional (system *system*))
- find-component-mode (mode-name component)
- find-instance (instance-name &optional (system *system*))
- find-mode (name &optional (system *system*))
- find-module (name &optional (system *system*))
- find-module-by-short-name (name &optional (system *system*))
- find-monitor (name &optional (system *system*))
- find-proposition (name &optional (theory *theory*))
- find-proposition-value (pname &optional (theory *theory*))
- find-relation-definition (relation)
- find-variable-legal-values (variable)

1.12.3 Common errors encountered in Livingstone

- **Component X in mode Y has no next state**
  This indicates Livingstone does not have enough information to determine the next mode of this component. This could mean there is a problem in the `:transitions` of the mode Y.

  However, it usually means you have forgotten to specify a command or monitor that matches the inputs or outputs of the component, and therefore transitions determined by these values cannot be taken.

  If you think you have specified all the necessary commands and montiors, check that the component input or output attribute is spelled correctly when you create the command or monitor. Creating switch1 with command input `(command-in switch1)` and then creating and using a command `(command-in switch-1)` will result in this error.

- **Command is inconsistent with initial state**
  The current state of a model is represented by a set of propositions. When do-cmd is used to give a command, the first step is for Livingstone to assert the command into the model as a proposition. This may allow new inferences to be made with contradict the original set of propositions.

  For example, consider a component which has a mode with the following model, where `(command-in ?name)` is a command to be asserted:

```
:model  (and  (= on a)
        (implies (= off (command-in ?name))
                 (= off a)))
```

In this mode, `(= on a)` is true. However, if the off command is given in this mode, `(= off (command-in ?name))` is asserted. Thus, `(= off a)` is implied, which contradicts `(= on a)`, presuming `(= off a)` and `(= on a)` are mutually exclusive. Thus, asserting the off command is inconsistent with the current mode of a component. The Livingstone function `explain-current` is useful here.

- **Operator `next` seems to have no effect**
  The `next` operator in Livingstone specifies that a proposition must be true in the next state after a transition. Note that `next` only applies to a single proposition or negated proposition, such as `(next (not (= off (command-in ?name))))`.

  You **cannot** use `next` to directly state that a more complex clause such as `(or (on a) (= off (command-in ?name))))` will be true in the next state. Unfortunately, Livingstone will parse this without error and simply ignore it, potentially creating a debugging problem.

  You can easily represent the same effect with `next` by using the logical equivalent of a flag to determine if some complex clause should be true. For example, if a component has the background model `(implies or-is-true (or (= on a) (= off (command-in ?name)))))`, then `(next or-is-true)` has the same effect as the **incorrect** `(next (or (= on a) (= off (command-in ?name))))`.