

D

ELAN—An expression language

D.1 Introduction

The ELAN expression language is a small language designed for use in inscriptions in MOSES models. Its design goals were the following:

- Simple syntactic structure.
- Simple, side-effect free evaluation semantics, as well as simple constructs for side-effects and iteration.
- Interoperability with the underlying host-language, i.e. Java. Java structures should be usable by ELAN constructions, and ELAN objects should have a straightforward Java representation.
- Support for a number of basic and structured types that allows for the construction of practical models without recourse to Java.
- No static typing.

Evaluation efficiency was not the primary design goal, the rationale being that algorithmically complex activities and runtime-critical parts would be done in Java anyway. We had no intention to duplicate existing facilities, only to merge them into MOSES models in a sufficiently convenient manner.

D.2 Preliminaries

The key concept of the language is an *expression*. There are a variety of expressions in ELAN, but their common characteristic is that all of them can be *evaluated* to produce a *value*. This value in general depends on an *environment*, which is simply a (partial) mapping of variable names to objects.

The evaluation of some expressions may also produce *side-effects*. Side-effects come in different forms—some expression may change the environment by substituting the object a variable is bound to for another object, other expression may change an object's internal state. We will hint at possible side-effects whenever they may occur.

In the following, we will denote general expressions by symbols e and e_i , boolean expressions (conditions) by c and c_i , and variable symbols by v and v_i . Operator symbols will be written as \circ .

D.2.1 Atomic expressions

Atomic expressions are those containing no other expressions. These are variable symbols and a number of constants of different types.

Each identifier is a variable symbol, evaluating to the value it is bound to in the current environment.

ELAN recognizes constants of the following types:

- Integers: any sequence of decimal digits optionally prefixed with a sign.
- Real numbers: any sequence of digits with a period, optionally prefixed with a sign and appended with a decimal exponent in the form of an "E" followed by an optionally signed integer.
- Strings: any sequence of characters enclosed in double quotes.
- Boolean: any of the two keywords **true** and **false**.
- The symbol **null**, denoting the null object.

D.2.2 Unary Operators

There are two unary operators.

$$- e$$

requires that e evaluates to a number and computes the negative value of that number.

$$\sim e$$

requires e to result in a boolean value, and computes the negation of that value.

D.2.3 Binary Operators

The application of a binary operator \circ has the following format:

$$e_1 \circ e_2$$

ELAN currently does not define precedence rules, so one needs to provide proper parentheses to disambiguate expressions.

There are a number of binary operators to manipulate various kinds of objects. For numerical operands, these are the following:

$$+, -, *, /, \text{mod}, \text{div}$$

with the usual interpretations.

For sets, we have the operators

$$+, -, *$$

denoting union, set difference and intersection, respectively. Additionally, the boolean expression

$$e_1 \text{ in } e_2$$

requires e_2 to evaluate to a set, and then it is true if the value of e_1 is an element of that set and false otherwise.

Lists can be concatenated using the $+$ operator.

Maps may be 'added' using the $+$ operator, where the mappings of the right-hand map override those of the left-hand map. Maps may be concatenated using the $*$ operator.

Boolean values may be manipulated using the operators

$$\&\&, ||$$

denoting conjunction and disjunction, respectively.

Finally, objects may be compared to each other. All objects may be compared for equality or inequality using $=$ or $<>$. Objects that have an order defined on them may be compared using the operators

$$<, <=, >, >=$$

For numbers, the order is the numerical one, for strings it is lexical order, and for sets it is inclusion.

D.2.4 Comprehensions and basic structured data objects

ELAN provides direct support for three kinds of structured data objects: sets, lists, and maps. They are constructed by so-called *comprehensions* in the following manner.

A set comprehension has the form:

$$\{e_1, \dots, e_n : \text{for } v_1 \text{ in } e_{1,s}, c_{1,1}, \dots, c_{1,k_1}, \dots, \text{for } v_m \text{ in } e_{m,s}, c_{m,1}, \dots, c_{m,k_m}\}$$

The constructions **for** v_i **in** $e_{i,s}$ are called *generators*, the boolean expressions $c_{i,j}$ are *filters*. The $e_{i,s}$ must evaluate to sets. Such a set comprehension evaluates by iteratively creating a number of bindings for the v_i , viz. all those in which a variable v_i is bound to an element of the set that results from evaluating $e_{i,s}$ so that all $c_{i,j}$ are true. For each of these bindings, the expressions e_i at the head of the comprehension are evaluated and all resulting objects are added to the set. Examples of set comprehensions:

$$\begin{aligned} & \{1, 2, 3, 4\} \\ & \{a : \mathbf{for} \ a \ \mathbf{in} \ s, a \ \mathbf{mod} \ 2 = 0\} \\ & \{a * b : \mathbf{for} \ a \ \mathbf{in} \ s1, \mathbf{for} \ b \ \mathbf{in} \ s2\} \\ & \{a, b : \mathbf{for} \ a \ \mathbf{in} \ s1, \mathbf{for} \ b \ \mathbf{in} \ s2\} \end{aligned}$$

The first is a very simple comprehension without generators resulting in a set of four integers, the second assumes the existence of a variable s (bound to a set of numbers) and computes the set of even number in s , the third computes the products of all combinations from number in the sets (bound to) $s1$ and $s2$, and the last one is a roundabout (and inefficient) way to compute the union of $s1$ and $s2$.

The same mechanism can be used for constructing lists and maps. For lists, the general format is the following:

$$[e_1, \dots, e_n : \mathbf{for} \ v_1 \ \mathbf{in} \ e_{1,s}, c_{1,1}, \dots, c_{1,k_1}, \dots, \mathbf{for} \ v_m \ \mathbf{in} \ e_{m,s}, c_{m,1}, \dots, c_{m,k_m}]$$

This constructs a list containing the enumerated elements, preserving the order among the e_i in each iteration.

A map is a finite mapping of *keys* to *values*. It is constructed in the following manner:

$$\mathbf{map}[e_{k,1} \rightarrow e_{v,1}, \dots, e_{k,n} \rightarrow e_{v,n} : \mathbf{for} \ v_1 \ \mathbf{in} \ e_{1,s}, c_{1,1}, \dots, c_{1,k_1}, \dots, \mathbf{for} \ v_m \ \mathbf{in} \ e_{m,s}, c_{m,1}, \dots, c_{m,k_m}]$$

Here, the $e_{k,i} \rightarrow e_{v,i}$ denote the mapping of the key $e_{k,i}$ to the value $e_{v,i}$. If the comprehension enumerates the same key more than once, the value is chosen non-deterministically.

D.2.5 Closures and closure application

The expression

$$\mathbf{lambda} (v_1, \dots, v_n) e \ \mathbf{end}$$

creates an n -ary closure, i.e. a function object encapsulating the current environment which is valid in the context of the evaluation of this expression. Such an object may be applied to a tuple of n actual parameters in an application expression, which has the following form:

$$e_{clos} (e_1, \dots, e_n)$$

Here, e_{clos} must evaluate to an n-ary closure. The result of this application is that of evaluating the body expression e of the closure in an environment that binds the v_i to the value of e_i .

For example, assume f is bound to the value of **lambda** (x) $2 * x$ **end**, then the expression

$$f(3)$$

will return 6.

D.2.6 Structured expressions

The most basic structured expression is obtained by putting any expression in parentheses:

$$(e)$$

The value of this expression is that of e .

Another fundamental structured expression is the conditional. It has the following form:

if c **then** e_1 **else** e_2 **end**

If the value of c is true, then the value of this expression is the value of e_1 , otherwise it is the value of e_2 .

Another structured expression allows the definition of local variable bindings. The has the form

let $v_1 = e_1, \dots, v_n = e_n : e$ **end**

The value of this expression is the value of e in an environment that binds v_i to the value of e_i .

D.2.7 Quantified expressions

Boolean expressions may be universally quantified over sets in the following manner:

forall v_1 **in** e_1, \dots, v_n **in** $e_n : c$ **end**

Here, the e_i are required to evaluate to sets. The result of this expression is true if c is true in all environments resulting from the binding of the v_i to values from the set resulting from e_i . It is false otherwise.

Likewise, a boolean expression may be existentially quantified as follows:

exists v_1 **in** e_1, \dots, v_n **in** $e_n : c$ **end**

This expression is true if c evaluates to true in at least one of the environments resulting from the binding to the v_i to values from the set resulting from e_i . It is false otherwise.

D.2.8 Iterative constructs

Three constructs support iteration in ELAN: assignment, sequential expressions, and loops.

The assignment expression

$$v := e$$

evaluates e and changes the value of the variable v to the value of e . This is also the value of the assignment expression.

Sequential expressions are a way to evaluate a number of expressions after each other:

$$e_1 \mathbf{; \dots ;} e_n$$

This evaluates the e_i in the order from left to right. The result of this sequential expression is the value of e_n .

Loops provide a way to iterate an expression until a condition is met:

while c : e end

This expression evaluates e as long as c evaluates to true. The value of this expression is the value of the last evaluation of e . If c was false at the very beginning, the value of this expression is null.

E

The MOSES Tool Suite

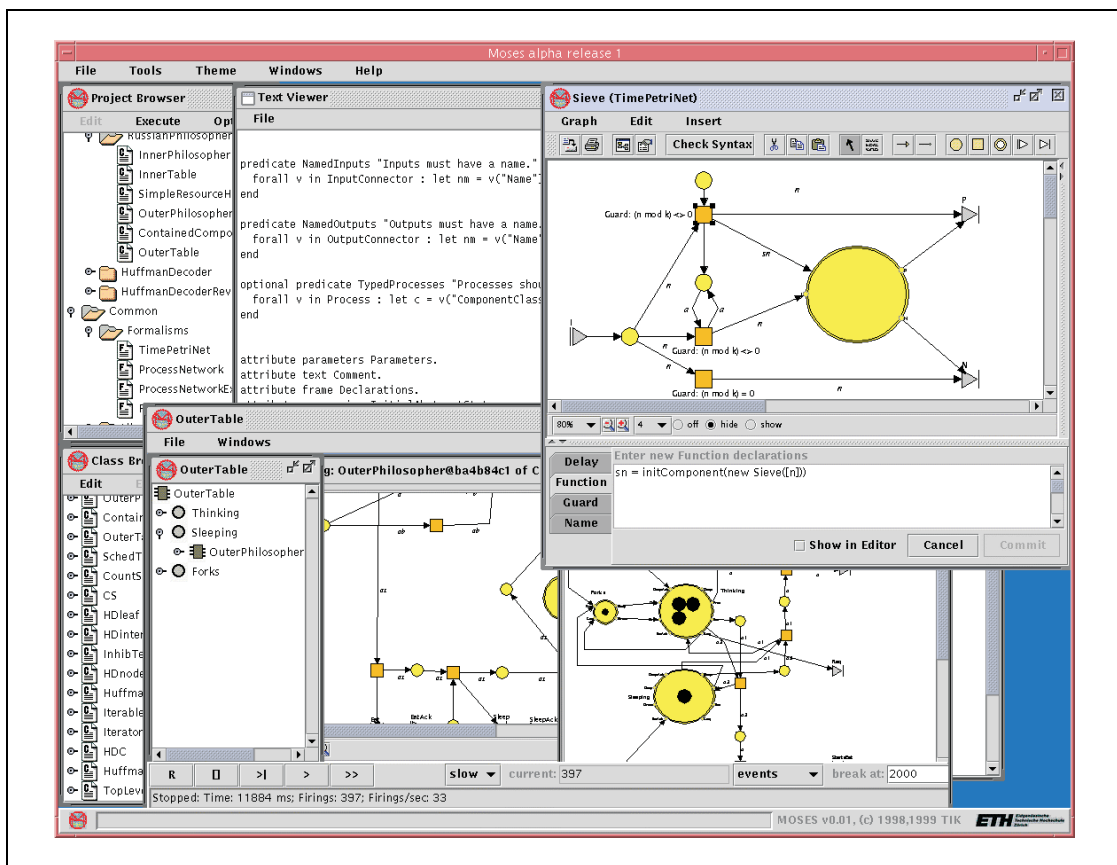


Fig. 41: MOSES Tool Suite, Overview

E.1 Introduction

The MOSES *Tool Suite* is a set of software tools developed in the MOSES Project [1]. It forms the technical environment of this work, and all implementation work has been done in it. The Tool Suite is focused on supporting the following tasks:

- Defining the syntax of new graph-like visual notations, including descriptions of its concrete syntax as well as the definition of syntactical constraints (in the way discussed in Chapter 3).
- Providing an editing environment for these languages which is generated from the syntax descriptions.
- Interpreting graphs on a generic execution platform, so that different visual languages can easily interoperate. Animating visual programs during their run.

In addition to tools directly supporting these tasks (a graph editor, a simulation environment, an animator, a graph translator), there are also management tools, such as a repository for storing and organizing the various files and objects that belong to a modeling and simulation project.

Here we want to give a brief overview of the definition of a visual notation using the Moses Tool Suite and how the resulting specification is used to configure the various parts of the software.

E.2 Syntax definition and editor

The syntax of a visual language is defined textually by enumerating the kinds of vertices and edges that occur in the language and by giving syntax rules for their composition, in the way described in Chapter 3. Section E.4 shows a full specification of a simple Petri net language, in the so-called *Graph-Type Definition Language (GTDL)* [92]. The kind of graph defined by such a description is called a *graph type*.

The specification in E.4 declares the following entities:

- Some kinds of vertices (*vertex types*), including a set of attributes for each (Place, Transition).
- A kind of edge (*edge type*), also along with attributes that describe it (Arc).
- Several syntax predicates that define the well-formedness of a graph (PlaceTransitionConnection, TransitionPlaceConnection, WeightsPositive, NonEmptyPreset).
- The class name of the compiler to be used to translate a graph into executable code (Compiler).

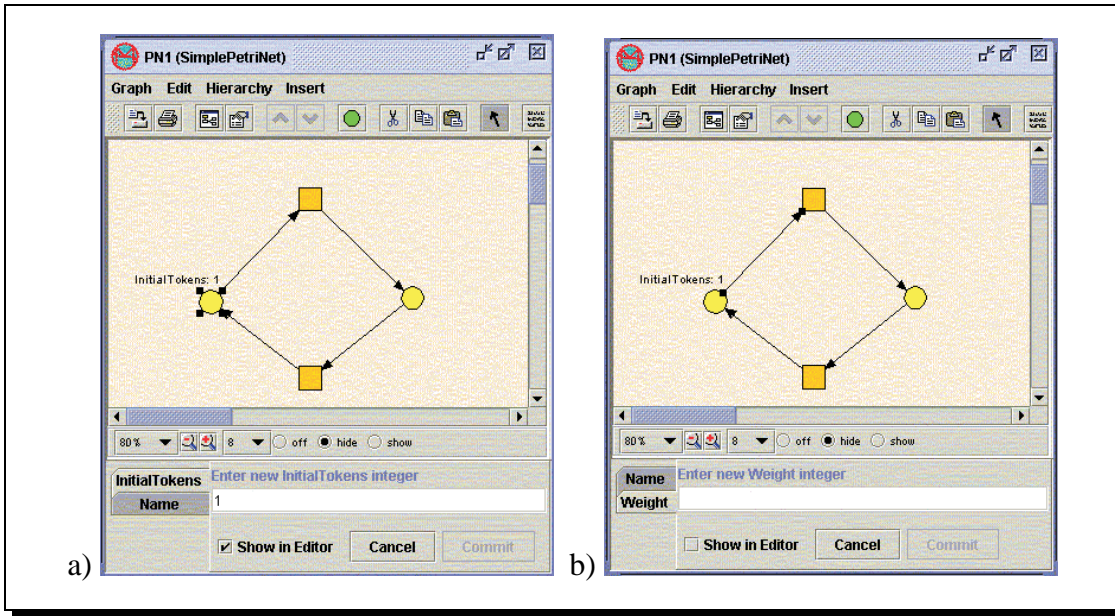


Fig. 42: Object-dependent attribute editors.

The definition of the vertex type (and similarly of an edge type) *Place* looks like this:

```
vertex type Place(string Name, integer InitialTokens)
graphics(hidden string Shape = "Oval",
           hidden color Color = "black",
           hidden color FillColor = "yellow",
           hidden integer ExtentX = 24, hidden integer ExtentY = 24).
```

After the name of the type we have a sequence of *attributes*, defined by a name (*Name*, *InitialTokens*) and their *type* (*string*, *integer*). This type defines the kind of objects that this attribute must contain. Following this is a similar list of graphical attributes, which define the appearance of a vertex when it is displayed. These are usually marked as 'hidden', which advises any tools processing such a description to hide these attributes from the user.

The most important tool that uses these descriptions is the graph editor. This is an interactive direct-manipulation editor that allows the construction of graphs according to such a graph type specification. It makes use of the definition of vertex and edge types in two ways: it displays instances of these types according to their graphical attributes, and it also provides *attribute editors* which are generated from the attribute definitions associated with a vertex or edge type. These attribute editors are dialogs that allow the user to manipulate the (non-hidden) declared attributes of an edge or a vertex. Fig. 42a shows the attribute editor generated from the definition of the *Place* vertex type (which is displayed in the lower part of the editor window whenever a *Place* vertex is

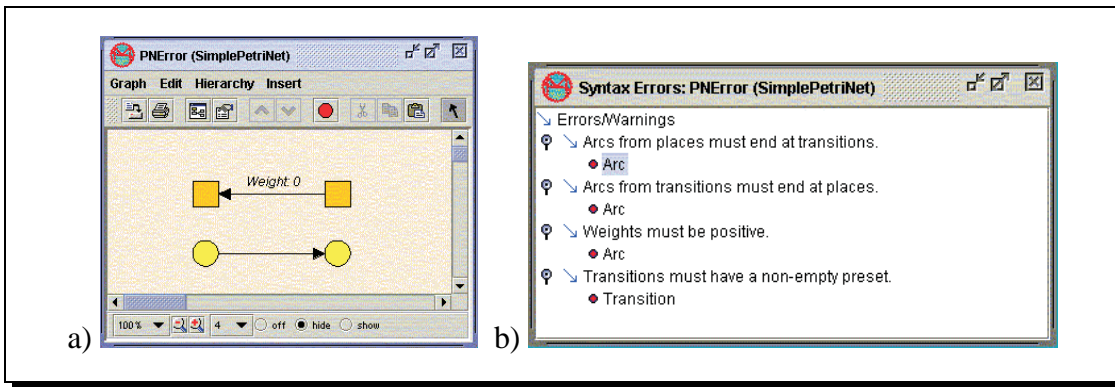


Fig. 43: Error messages generated from syntax predicates.

selected), while Fig. 42b shows the attribute editor displayed when an Arc edge is selected.

The editor also makes use of the syntax predicates in a graph type specification. For example, the syntax predicate that requires all Transition vertices to have a non-empty preset (i.e. there must be at least one Arc edge ending at each Transition vertex) has the following form:

```
predicate NonEmptyPreset
    "Transitions must have a non – empty preset."
forall t in Transition :
    exists e in Arc : dst(e) = t end
end
```

Apart from the predicate name (*NonEmptyPreset*), a predicate has two parts: A message that explains the syntax constraint, and a logical expression that is true if it is fulfilled by the graph. Note that in the above example, variables are quantified over sets called *Transition* and *Arc*—each vertex type and edge type automatically defines a set of that name that contains exactly the vertices/edges of the corresponding type.

Fig. 43a shows a rather non-well-formed Petri net, that violates all four of the predicates in the example in Section E.4. This violation is indicated by the red button just above the main editing area. That button is green for syntactically correct graphs (such as the ones in Fig. 42, and it turns red when an editing operation makes at least one predicate evaluate to false.

In that case, clicking on the button makes a more detailed error report available, as shown in Fig. 43b. This dialog shows the predicates violated (by their clear-text predicate message), and also the vertices or arcs where they failed (clicking on the corresponding red dot would select the culprit).

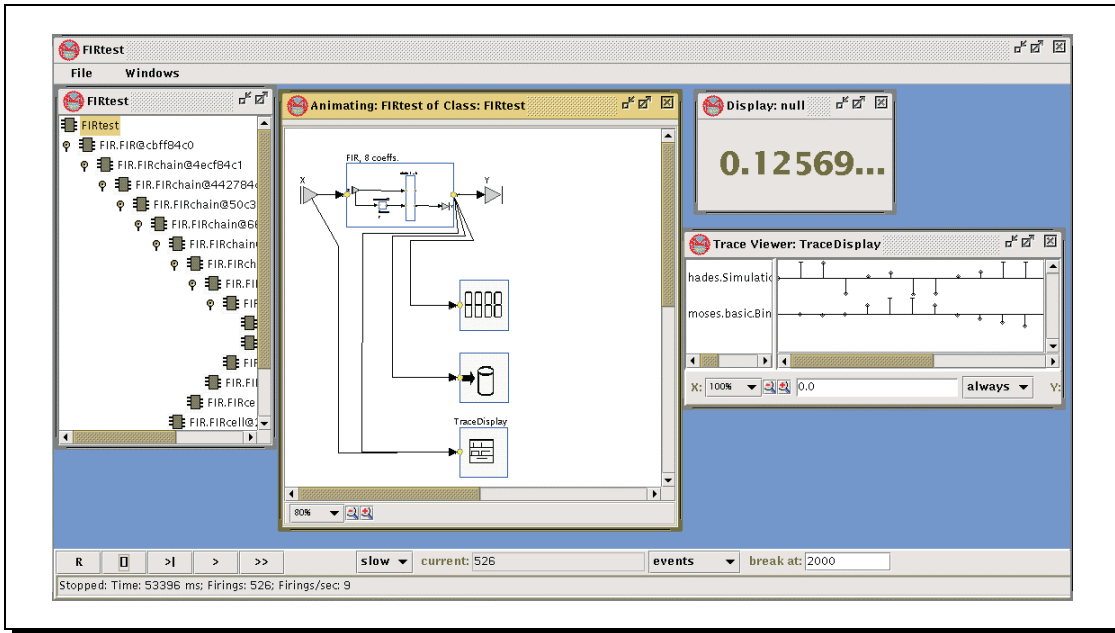


Fig. 44: The simulation environment.

E.3 Execution and animation

Once a graph has been entered and found to be syntactically correct, it can be compiled into executable code and then executed. Although a model may be executed as a standalone program, the Moses Tool Suite provides an environment that facilitates stepwise execution, animation, and inspection of the model state. The interface of this environment is shown in Fig. 44.

The main window features a set of controls at its bottom and a number of internal windows. The controls are used to run and stop the model, a given number of steps or until a certain point in (virtual) time, with or without animation etc. The internal window on the left depicts the containment hierarchy of the model—this is dynamic, so the corresponding tree view may change during the execution of the model.

The big central window shows a top-level view of the model itself—a similar window may be opened for any component in the containment hierarchy. One can also see 'into' contained components by making their container 'transparent' (as shown for the topmost component in that window). Other internal windows may show aspects of the state of the model, show a history of states or output tokens produced etc.

E.4 Sample GTDL Specification

graph type *SimplePetriNet*{

vertex type *Place*(**string** *Name*, **integer** *InitialTokens*)

graphics(**hidden string** *Shape* = "Oval",
hidden color *Color* = "black",
hidden color *FillColor* = "yellow",
hidden integer *ExtentX* = 24, **hidden integer** *ExtentY* = 24).

vertex type *Transition*(**string** *Name*)

graphics(**hidden string** *Shape* = "Rectangle",
hidden color *Color* = "black",
hidden color *FillColor* = "orange",
hidden integer *ExtentX* = 24, **hidden integer** *ExtentY* = 24).

edge type *Arc*(**string** *Name*, **integer** *Weight*)

graphics(**hidden string** *Head* = "ClosedTriangle",
hidden color *Color* = "black",
hidden color *FillColor* = "black").

predicate *PlaceTransitionConnection*

"Arcs from places must end at transitions."

forall *e* **in** *Arc* :
if *src*(*e*) **in** *Place* **then**
dst(*e*) **in** *Transition*
else true end
end

predicate *TransitionPlaceConnection*

"Arcs from transitions must end at places."

forall *e* **in** *Arc* :
if *src*(*e*) **in** *Transition* **then**
dst(*e*) **in** *Place*
else true end
end

predicate *WeightsPositive*

"Weights must be positive."

forall *e* **in** *Arc* :
if *e*("Weight") <> **null** **then** *e*("Weight") > 0
else true end
end

```
predicate NonEmptyPreset
  "Transitions must have a non – empty preset."
  forall t in Transition :
    exists e in Arc : dst(e) = t end
  end

const Compiler = "moses.models.translator.simplepn.PNCompiler".
}
```