NAME

seq, SEQ – CADP common format for execution sequences (i.e., traces)

DESCRIPTION

The SEQ format (where *SEQ* stands for *SEQuence*) is used to specify a finite set (possibly empty) of execution sequences belonging to a Labelled Transition System (LTS). Each of these sequences is finite and starts from the initial state of the LTS. Thus, a SEQ file specifies a subgraph of the LTS; in this subgraph, only the initial state may have more than one successor state (namely, if there are several sequences).

The SEQ format has been carefully designed so as to be easily readable and writable by both humans and computer programs. For this reason, it is a character-based format. Files in the SEQ format are expected to have a **.seq** extension.

There are two versions of the SEQ format:

- In the *simple format*, execution sequences are merely specified as lists of transition labels; labels are specified as character strings.

- In the *full format*, execution sequences are specified in a more sophisticated way, using regular expressions; labels are represented either as character strings or regular expressions.

Both versions of the SEQ format are compatible in the sense that the simple format is a subset of the full format. Therefore, the simple format can be used at every place where the full format is allowed.

BNF-LIKE NOTATION

The syntax of the SEQ format is described below using a notation similar to the BNF (Backus-Naur Form) notation. However, as the angle brackets **<** and **>** used in BNF are also meaningful in the SEQ format, there are some differences with respect to the standard BNF notation:

- Terminal symbols are enclosed in simple quotes, whether they are one or several characters long (for instance: **'+'**, **'"'**, **'<while>'**, etc.). In particular, **'\n'** and **'\t'** denote the newline and tabulation characters.

- Non-terminal symbols are written as alphabetic identifiers (for instance, **sequence**, **sequence_list**, etc.). Contrary to the standard BNF notation, non-terminal symbols are not enclosed within angle brackets.

- The following meta symbols are used, with their usual meaning: parentheses for grouping, star (∗) for zero or more repeated occurrences, and vertical bar (|) for alternates.

Note: it should be understood that ∗ and **'∗'** do not have the same meaning: the former is the meta-symbol denoting repeated occurrences, whereas the latter denotes the terminal character "star".

LEXICAL DEFINITIONS

STRINGS

A **string** is a sequence of characters, enclosed between double quotes characters **'"'**, that denotes a label of the LTS:

string ::= '"' valid_character∗ '"'

where **valid_character** denotes any character different from double quote (**'"'**) and from end-of-line (**'\n'**). Consequently, a **string** cannot encompass several lines; however, there can be several **string**s on the same line (see, e.g., **label** and **label_group** below).

The **string**s can be used in both the simple and full SEQ format, with the same lexical conventions.

REGULAR EXPRESSIONS

A **regular_expression** is a notation for a set of labels. The **regular_expression**s of the SEQ format are based upon UNIX basic regular expressions (see the **regexp**(5) and **regexp**(LOCAL) manual page for a detailed description of UNIX basic regular expressions). Syntactically, a **regular_expression** of the SEQ format is a UNIX basic regular expression enclosed between square brackets **'['** and **']'**:

```
regular_expression ::= '[' UNIX_basic_regular_expression ']'
```

Unlike standard UNIX basic regular expressions, each **regular_expression** must satisfy two additional restrictions:

- First, it should not be empty (otherwise, the **regular_expression** would be confused with another meaningful token **'[]'**).

- Second, there should be exactly the same numbers of **'['** and **']'** characters in **UNIX_basic_regular_expression**. This precludes the use of very particular regular expressions such as **'a[^]]b'** or **'a[]bc]d'**. This restriction should not be a problem for OPEN/CAESAR users.

Like **string**s, **regular_expression**s cannot encompass several lines; however, there can be several **regular_expression**s on the same line (see, e.g., **label** and **label_group** below).

Note: the choice operator **'|'** is not supported in UNIX basic regular expressions. For instance,

```
[PUT.*|GET.*]
```

will search for a label of the form **"PUT.*|GET.*"** rather than for either **"PUT.*"** or **"GET.*"**. However, the intended meaning can be obtained using the choice operator available in the syntax of **label**s (see below):

```
[PUT.*] | [GET.*]
```

The **regular expression**s can only be used in the full SEQ format.

BLANKS

A **blank** is a (possibly empty) sequence of space characters **' '** and/or tab **'\t'** characters:

```
blank ::= ( ' ' | '\t' )*
```

Blanks can appear anywhere, at the beginning of a line, at the end of a line, or between two tokens. They are ignored (except, of course, in **string**s and **regular expression**s).

Note: end-of-line characters (**'\n'**) are not part of **blank**s. On the contrary, they are meaningful in the SEQ format as they are used in the definition of many non-terminal symbols.

Both versions of the SEQ format share the same lexical conventions for **blank**s.

COMMENTS

A **comment** is a sequence of characters that is meaningless and ignored. There are two kinds of **comment**s:

- Any sequence of characters that begins with the special character **'\001'** (control-A) and that ends with the special character **'\002'** (control-B) is a **comment**. A **comment** of this form may encompass several lines of text. The characters **'\001'** and **'\002'** have been selected because they are not visible by the user.

- Any line whose first non-blank character does not belong to the following list of reserved characters: **'['**, **'('**, **'<'**, **'"'**, **'~'**, **'\001'** is a **comment**. This **comment** extends up to the end-of-line. This definition includes the case of lines that contain nothing but **blanks**.

Both versions of the SEQ format share the same lexical conventions for **comment**s.

SYNTAX OF THE SIMPLE FORMAT

The following BNF-like grammar defines the syntax of the simple SEQ format. The axiom of the grammar is **sequence_list**.

```
sequence_list      ::= ''
                     |  sequence
                     |  sequence '[]' '\n' sequence_list

sequence           ::= string '\n'
                     |  string '\n' sequence
                     |  '<deadlock>' '\n'
```

Note: this grammar defines a regular language.

SYNTAX OF THE FULL FORMAT

The following BNF-like grammar defines the syntax of the full SEQ format. The axiom of the grammar is **sequence_list**.

```
sequence_list      ::= ''
                     |  sequence
                     |  sequence '[]' '\n' sequence_list

sequence           ::= label_group '\n'
                     |  label_group '\n' sequence
                     |  '<deadlock>' '\n'

label_group        ::= label
                     |  label '*'
                     |  label '+'
                     |  '<while>' label
                     |  '<until>' label
                     |  '<while>' label '<until>' label

label              ::= simple_label
                     |  label '&' simple_label
                     |  label '|' simple_label
                     |  label '^' simple_label
```

```
simple_label      ::= '<any>'
                    |  string
                    |  regular_expression
                    |  '~' simple_label
                    |  '(' label ')'
```

Note: each **label_group** (and consequently each **label** and **simple_label**) appears on a single line of text.

Note: from the grammar, the postfix operators **'+'** and **'*'**, and the **'<while>'** and **'<until>'** operators have the lowest priority. Then, the binary operators **'&'**, **'|'**, and **'^'** have the same, intermediate priority. Finally, the prefix operator **'~'** has the highest priority.

Note: the simple SEQ format is the subset of the full SEQ format in which each **label_group** is constrained to be simply a **string**.

SEMANTICS OF THE FULL FORMAT

The semantics of the full format is defined by induction on its syntax; the semantics of the simple format can be derived as a special case.

Let (S0 T1 ... Tn Sn) be an execution sequence that starts from some state S0 (not necessarily the initial state of the LTS) and that reaches some state Sn by applying n successive transitions T1, ..., Tn. The number n of transitions can be null.

A SEQ file contains a finite list of execution **sequence**s, separated by the **'[]'** keyword. This list can be empty, as specified by the **''** token in the BNF-like grammar. The semantics of the full format is only defined for a given **sequence**:

- If this list is empty, then the SEQ file only matches the empty execution sequence (S0), where S0 is equal to the initial state of the LTS.

- If this list contains more than one element, then a particular **sequence** must be selected; for instance, the **-seqno** of the **exhibitor**(LOCAL) tool enables the user to indicate which **sequence** is to be considered. In such case, the semantics of the full format is the set of all execution sequences (S0 T1 ... Tn Sn) such that S0 is equal to the initial state of the LTS and such that the sequence-matching relation "(S0 T1 ... Tn Sn) |==== **sequence**" defined hereafter is satisfied.

DEFINITION OF SIMPLE LABEL MATCHING

For any transition T of the LTS, let L(T) denote the character string generated from the label of transition T.

Let "T |= **simple_label**" be a relation expressing that the transition T "matches" **simple_label**. This relation is defined by induction on the syntax of **simple_label** and it is mutually recursive with the relation "T |== **label**" defined in the next subsection.

T |= **'<any>'**
: is always true

T |= **string**
: iff L(T) is equal to **string**

T |= **regular_expression**
iff L(T) matches **regular_expression**

T |= **'~' simple_label**
iff not T |= **simple_label**

T |= **'(' label ')'**
iff T |== **label**

DEFINITION OF LABEL MATCHING

Let "T |== **label**" be a relation expressing that the transition T "matches" **label**. This relation is defined by induction on the syntax of **label**.

T |== **simple_label**
iff T |= **simple_label**

T |== **label '&' simple_label**
iff T |== **label** and T |= **simple_label**

T |== **label '|' simple_label**
iff T |== **label** or T |= **simple_label**

T |== **label '^' simple_label**
iff T |== **label** exclusive-or T |= **simple_label**

Note: **regular_expression**s apply to *entire* **label** strings, from the first character to the last one, and not to substrings. For instance, the **label 'PUT !0'** will match the regular expression **'PUT.*'**, but not **'PUT'**. Consequently, the special characters **'^'** and **'$'** of UNIX basic regular expressions are useless in the SEQ format, and should not be used.

DEFINITION OF LABEL GROUP MATCHING

Let "(S0 T1 ... Tn Sn) |=== **label_group**" be a relation expressing that the execution sequence (S0 T1 ... Tn Sn) matches **label_group**. This relation is defined by induction on the syntax of **label_group**.

(S0 T1 ... Tn Sn) |=== **label**
iff n = 1 and T1 |== **label**

(S0 T1 ... Tn Sn) |=== **label '*'**
iff for all i in {1 ... n} Ti |== **label**

The remaining constructs **'+'**, **'<while>'**, and **'<until>'** used in the syntactic definition of **label_group** are merely shorthand notations introduced for user convenience. They are defined as follows:

- The construct:
  label '+'
  is equivalent to:
  label '\n'
  label '*'

It denotes a sequence of one or more transitions matching **label**.

- The construct:
  '<while>' label
  is equivalent to:
  label '∗'
  It denotes a sequence of zero or more transitions matching **label**.

- The construct:
  '<until>' label
  is equivalent to:
  '(~' label ')∗' '\n'
  label
  It denotes a sequence of zero or more transitions that do not match **label**, followed by a transition matching **label**.

- The construct:
  '<while>' label1 '<until>' label2
  is equivalent to:
  '(' label1 '& ~' label2 ')∗' '\n'
  label2
  It denotes a sequence of zero or more transitions that match **label1** and do not match **label2**, followed by a transition matching **label2**.

DEFINITION OF SEQUENCE MATCHING

Let "(S0 T1 ... Tn Sn) |==== **sequence**" be a relation expressing that the execution sequence (S0 T1 ... Tn Sn) matches **sequence**. This relation is defined by induction on the syntax of **sequence**.

(S0 T1 ... Tn Sn) |==== **label_group '\n'**
iff (S0 T1 ... Tn Sn) |=== **label_group**

(S0 T1 ... Tn Sn) |==== **label_group '\n' sequence**
iff there exists some state Sm in the sequence (S0 T1 ... Tn Sn) such that:
(S0 T1 ... Tm Sm) |=== **label_group** and
(Sm Tm+1 ... Tn Sn) |==== **sequence**

(S0 T1 ... Tn Sn) |==== **'<deadlock>' '\n'**
iff n = 0 (the sequence is reduced to a single state) and
S0 is a sink state (no transition goes out from S0)

EXHIBITOR'S SEMANTIC CONVENTIONS

The current version of **exhibitor**(LOCAL) interprets the full SEQ format in particular ways, described hereafter.

TRANSITION LABELS

Since **exhibitor**(LOCAL) operates on the fly and is based on the OPEN/CAESAR's graph module, it implements the aforementioned T(L) notation by invoking the **CAESAR_STRING_LABEL()** function (see the **caesar_graph**(LOCAL) manual page).

CASE INSENSITIVITY

In order to be compatible with the conventions used by **caesar**(LOCAL) when printing labels as character strings, all lower-case letters contained in **`string`**s and **`regular_expression`**s are turned to upper case. However, the **`string`**s and **`regular_expression`**s (case-insensitively) equal to **`"i"`** or **`"exit"`** are recognized as special values (denoting the internal gate and the termination gate) and turned to lower case.

This is the default option, but it can be overriden using the **-case** option of **exhibitor**(LOCAL) if case sensitivity needs to be preserved.

DETERMINIZATION STRATEGY

Given a **`sequence`**, **exhibitor** will search for execution sequences (S0 T1 ... Tn Sn) such that S0 is equal to the initial state of the LTS and such that (S0 T1 ... Tn Sn) |==== **`sequence`**.

In the above second semantic rule defining sequence matching (namely, "(S0 T1 ... Tn Sn) |==== **`label_group '\n' sequence`**"), if there exist several states Sm, the one with the greatest index m is selected. By doing so, **exhibitor** reduces potentially non-deterministic sequences into deterministic ones. Intuitively, every time that **exhibitor** has the choice between remaining in a '∗'-group or leaving it, it will remain in the '∗'-group. For instance, if the label **`"B"`** has to be matched against the sequence:

```
(~ "A") *
    "B"
```

there is a conflict, since **`"B"`** matches both lines of the sequence. In such case, the sequence will not be recognized successfully, since the label **`"B"`** will be used to match the first line of the sequence instead of the second line. Therefore, the determinization strategy gives priority to the longest match.

The **-conflict** option of **exhibitor** (see the **exhibitor**(LOCAL) manual page for a detailed decription of this option) can be used to display the list of all conflicts which have been solved using this determinization strategy.

The solution to this problem consists in avoiding the conflict by making the sequence more precise:

```
(~ "A" & ~ "B") *
    "B"
```

Similarly, the sequence:

```
<any>*
    "A"
```

will never be recognized, because of the conflict between **`<any>`** and **`"A"`**. It should be written instead:

```
(~ "A") *
    "A"
```

Note: translating the **`label_group`** construct:

```
label '+'
```

to:

```
label '*' '\n'
    label
```

would not be correct because, due to the determinization strategy, this sequence is never recognized (one always remains in the '∗'-group).

SEQUENCE REDUCTION

exhibitor removes all trailing '∗'-groups at the end of the sequence to be searched, because these groups are meaningless. For instance, the following sequence:

```
"A"
    "B"*
```

```
      "C"*
```
is reduced to:
```
   "A"
```

If the sequence becomes empty due to this reduction, **exhibitor** emits a warning and stops.

EXAMPLES OF PATTERNS

The following sequence:
```
   "i" *
      "PUT"
      "i" *
      "GET"
```
searches for an action **"PUT"**, followed by an action **"GET"**, with any number of invisible actions **"i"** before and between.

The following sequence:
```
   <until> [PUT !TRUE !.*]
      <until> [GET !FALSE !.*]
```
searches for an action of the form **"PUT !TRUE !.*"**, followed by an action of the form **"GET !FALSE !.*"**, with any number of visible or invisible actions before and between.

The following sequence:
```
    <until> ([SEND !.*] & ~ "SEND !NULL")
```
searches for an action of the form **"SEND !.*"** such that the offer associated with gate **"SEND"** is different from **"NULL"**.

The following sequence:
```
   <until> "OPEN !1"
      <while> ~ "CLOSE !1" <until> "OPEN !2"
```
searches for an action **"OPEN !1"**, followed by an action **"OPEN !2"** without any **"CLOSE !1"** action between them.

The following sequence:
```
   <any>*
       <deadlock>
```
searches for deadlocks. Thus, **exhibitor** can be used as an alternative to **terminator**(LOCAL), although it implements totally different algorithms.

HOW TO CREATE A SEQ FILE

It is easy to create a SEQ file manually, using a text editor. It is also possible to produce a SEQ file automatically, using the **bcg_io**(LOCAL) tool, which converts to the simple SEQ format a graph (encoded in various other formats) consisting of a set of sequences all starting from the initial state. Finally, many CADP tools for simulation, model checking, equivalence checking, etc. generate their output in SEQ format when such output denotes an execution sequence or a set of execution sequences (as opposed to more general labelled transition systems).

HOW TO READ A SEQ FILE

The tool **seq.open**(LOCAL) reads a SEQ file in the simple SEQ format.

The tool **exhibitor**(LOCAL) reads a SEQ file in the full SEQ format.

SEQ files can be converted to many other graph formats using the **bcg_io**(LOCAL) tool.

AUTHORS

The SEQ format was developed by Hubert Garavel (INRIA Rhone-Alpes).

SEE ALSO

bcg_io(LOCAL), **exhibitor**(LOCAL), **seq.open**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_***.*

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Please report any bug to cadp@inria.fr