THE INCA QUERY LANGUAGE

STEPHEN F. SIEGEL

Abstract. INCA is a tool for analysis of concurrent systems. The system to be analyzed is modeled in the S-Expression Design Language (SEDL), and the properties of the system to be verified are written in the INCA Query Language. INCA takes these two items as input, and produces Integer Linear Programming problems, which can be analyzed by standard linear programming tools, such as CPLEX. In this document, we give a precise description of the syntax and semantics of the INCA Query Language, as well as several examples of INCA queries.

1. Introduction

The *INCA Query Language* is a language for describing sets of executions of a system of communicating finite state automata. (The precise definitions of these terms will be given in Section 3.) A single INCA query describes a set of executions of a given system of communicating FSAs; it is analogous to a single regular expression, which describes a set of strings over a given alphabet. The situation for queries, however, is more complicated, due to the more complex structure of communicating FSAs.

While in theory the INCA Query Language may be used to describe executions for an arbitrary system of communicating FSAs, it was designed, naturally, with INCA in mind. For this reason, it will help to have a basic understanding of what INCA does. Briefly, INCA works in two stages. The INCA front end takes as input a description of a concurrent system written in the S-Expression Design Language (or SEDL; see [1]) and produces from that a set of communicating FSAs. (There is one FSA for each task in the concurrent system.) The INCA back end begins with this set of communicating FSAs and a query. The query describes the set of all executions which violate the property of the system one wishes to verify. From these, the back end produces an Integer Linear Programming (ILP) problem in a standard format which can be read by most linear programming tools, such as the commercial package CPLEX. The linear programming tool determines if the system of equations and inequalities produced by INCA has a solution: if it does not, this means that there are no executions of the concurrent system which satisfy the query, i.e., the property holds. If, on the other hand, there is a solution to the system of equations and inequalities, this may or may not represent an actual

Date: July 7, 2003.

This research was partially supported by the National Science Foundation under grant CCR-9708184 and by the U.S. Army Research Laboratory and the U.S. Army Research Office under Agreement DAAD190110564. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the U.S. Army Research Laboratory, the U.S. Army Research Office, or the U.S. Government.

violating execution of the concurrent system, and some further work must be done to determine which is the case.

With this in mind, it is not surprising that the INCA Query Language uses many constructs which lend themselves naturally to an easy translation into the language of Integer Linear Programming. For example, it is quite easy to write a query which specifies all executions of a system of communicating FSAs in which the number of occurences of event a is greater than or equal to 3 times the number of occurences of event b.

There are also some elements of the language which have no effect at all on the set of executions determined by the query, but are instead used in some way to modify the final Integer Linear Programing problem generated by INCA. (An example of this is the "`cost`" parameter, which is used to define the objective function; see Section 5.) Moreover, since INCA analyzes programs written in a specific language, SEDL, there are a few constructs in the INCA Query Language for describing sets of events which arise naturally from SEDL programs (e.g., "`rend`" and "`rendezvous`"; see Section 7). But despite these few particulars, the INCA Query Language stands largely on its own, and one really does not have to know anything about INCA to understand it.

The purpose of this document is to give a precise definition of the syntax and semantics of the INCA Query Language. In Secion 3 we define precisely what we mean by a *system of communicating FSAs* and a (finite) *execution*. Section 4 gives the highest-level description of a query: in brief, the query consists of a list of *sequences*, each of which consists of one or more *intervals*. The most substantial part of the document is Section 5, where the various parameters for describing intervals are explained. Infinite executions are discussed in Section 6. The mechanisms for defining sets of symbols are explained in Section 7, and Section 8 concludes with several examples of INCA queries and a thorough discussion of each.

2. Tokens

The following productions will be used throughout in describing the grammar of the Inca Query Language:

```
<boolean>  ::= 't' | 'nil'
<digit>    ::= '0' | '1' | '2' | '3' | '4'
             | '5' | '6' | '7' | '8' | '9'
<sign>     ::= '+' | '-'
<number>   ::= [ <sign> ] <digit> { <digit> }
<letter>   ::= 'a' .. 'z' | 'A' .. 'Z'
<alpha>    ::= <letter> | <digit> | '_' | '-'
<name>     ::= '"' <letter> { <alpha> } '"'
<char>     ::= <alpha> | ';' | '(' | ')' | '.'
<string>   ::= '"' { <char> } '"'
<symbol>   :: = <string>
```

Although technically a `<symbol>` and a `<string>` are the same thing in the grammar we are describing, we will always use the former to represent an event in one of the FSAs, and this will help to clarify the presentation.

3. The Model

By a *system of communicating FSAs* we mean a set of FSAs $F_1, ..., F_n$ together with some additional structure. The additional structure is given as follows. Let A_i be the alphabet of F_i, and let T_i be the set of transitions of F_i. (We are assuming that $A_i \cap A_j$ and $T_i \cap T_j$ are empty for $i \neq j$.) For each i, we are given a subset A_i^c of A_i, the elements of which are called *communication* events. (The elements of $A_i \setminus A_i^c$ are called *local* events.) Let A^c be the (disjoint) union of the A_i^c, and let

$$f\colon A^c \to A^c$$

be a map such that (i) $f \circ f = \mathrm{id}_{A^c}$ and (ii) for all i, if $a \in A_i^c$ then $f(a) \notin A_i$. In other words, f establishes a pairing between the communication events in such a way that every communication event is paired with another communication event in a different FSA. The specification of the subsets A_i^c together with the function f constitute the additional structure.

Each FSA in the system has a unique *name*, which is represented in the query language by a `<name>`. Each state in each FSA has an *identifcation number*, unique within that FSA, which is represented in the language by a `<number>`. The events are all represented uniquely as `<symbol>`s in the query language. We will use the words *event* and *symbol* interchangably.

Although the language we are about to describe may be applied to any system of communicating FSAs, its main application is to INCA. The events in the communicating FSAs generated by INCA have a particular form; some examples are as follows:

- `"call(t1;t2.e)"`
- `"accept(t1;t2.e)"`
- `"internal(t1;x)"`.

The first two are paired communication events: the "call" is in the alphabet of the FSA named `t1`, and the "accept" is in the alphabet of `t2`. From the point of view of INCA and the query language, however, it does not matter which is the call and which is the accept. The third event is an example of a local event in task `t1`.

Suppose we have a finite sequence of sets of transitions

$$S_1, S_2, \ldots, S_m$$

for which

(i) $|S_j| \in \{1, 2\}$ $(1 \leq j \leq m)$.
(ii) If $|S_j| = 1$ then the element of S_j is labeled by a local event $(1 \leq j \leq m)$.
(iii) If $|S_j| = 2$ then the labels of the elements of S_j are a pair of communication events $x, f(x)$ $(1 \leq j \leq m)$.

For any $i \in \{1, \ldots, n\}$, we can "project" this sequence onto T_i as follows: we remove any of the S_j which do not contain an element of T_i and we replace any S_j which contains an element t of T_i with t.

Now we say that the finite sequence is an *execution-prefix* of the system of communicating FSAs if in addition:

(iv) For all $i \in \{1, \ldots, n\}$, the projection of the sequence onto T_i is an execution-prefix of F_i.

When we say a sequence of transitions is an execution-prefix of an FSA we just mean that it is a valid sequence of transition firings starting from the start-state,

but the FSA does not necessarily have to be in an accepting state after the last transition.

The idea is of course that the executions of the FSAs may be interleaved in any way possible, with the sole proviso that matching communication events must always occur simultaneously.

4. The Query

The INCA Query Language uses a Lisp-like syntax, which is not surprising, as INCA is written in Common Lisp. The general form of a query is

```
<query> ::= '(' 'omega-star-less' { <sequence> } ')'
```

For the origins of the term "omega-star-less", see [2].

In INCA, one generally defines a query and binds it to a name (say to the name `"my_query"`) using the syntax:

```
(defquery "my_query" "nofair" <query>).
```

The string `"nofair"` indicates that no fairness assumptions are to be added to the execution model described in Section 3 when interpreting this query. There are in fact other options which can be used in place of `"nofair"` but these are rarely used and are not discussed in this document.

The query represents a set of execution-prefixes of a system of communicating FSAs. We say that an execution-prefix *matches* the query if the execution is in the set represented by the query. In INCA, the query is used to define the set of executions which violate the property one wishes to verify.

We now focus on the question: when does an execution-prefix match a query?

The execution-prefix matches the query if, and only if, it matches at least one of the sequences.

When does an execution-prefix match a sequence?

A sequence has the form

```
<sequence> ::= '(' 'sequence' { <interval> } ')'
```

where

```
<interval> ::= '(' 'interval' [':initial' <boolean>]
                              [':progress' <boolean>]
                              [':open' <boolean>]
                              [':perpetual' <boolean>]
                              [':starts-after' <symbol list>]
                              [':ends-with' <symbol list>]
                              [':require' <symbol list>]
                              [':force' <symbol list>]
                              [':forbid' <symbol list>]
                              [':restrict' <restriction list>]
                              [':constraints' <constraint list>]
                              [':costs' <cost spec>] ')'

<symbol list>      ::= ''(' { <symbol spec> } ')'
<restriction list> ::= ''(' { <restriction> } ')'
<constraint list>  ::= ''(' { <constraint> } ')'
```

A *symbol spec* specifies a set of symbols and these will be discussed in Section 7. The syntax and semantics for `<cost spec>`, `<restriction>`, and `<constraint>` will be given in the relevant parts of Section 5.

Suppose we are given a sequence with k intervals $I_1, \ldots, I_k$. Assume also that there are no occurences of "`:perpetual t`" in any of the intervals. (This is used to specify a set of infinite executions, and these will be discussed in Section 6.) Then an execution-prefix $(S_1, \ldots, S_m)$ matches the sequence if, and only if, there are positive integers $1 \leq j_1 \leq \ldots \leq j_k \leq m+1$ such that the k *execution segments*

$$(S_{j_1}, \ldots, S_{j_2-1}), (S_{j_2}, \ldots, S_{j_3-1}), \ldots, (S_{j_k}, \ldots, S_m)$$

match the intervals $I_1, \ldots, I_k$. (Note: one or more of the segments may be empty.) What it means for the segments to match the intervals is the subject of Section 5.

5. The Interval Requirements

Each interval I in the sequence imposes certain requirements on the execution segments. The requirements are determined by the values of certain parameters associated with the interval. The values of the parameters are set using the keyword-value notation in Common Lisp. The various keywords, their default values, and the requirements they impose are explained below. The segments will match the intervals if, and only if, all of the interval requirements for all of the intervals hold.

Note that in order for the execution segments to match the intervals, *all* of the requirements specified by all of the intervals must hold. There is no (direct) way to say "or" in the query language, i.e., "this requirement must hold or that one must hold," within a single sequence.

For the following descriptions, let us fix an interval I, with corresponding segment $(S_a, ..., S_b)$. We now give for each interval parameter, the type of values that parameter takes, the default value, and a description of the requirement(s) which are implied by that parameter.

`initial` *type:* boolean *default:* `nil` (false)

Requirement: If initial is true (`t`), then we require $a = 1$.

`progress` *type:* boolean *default:* `nil` (false)

Requirement: If true, we require that at the end of the execution segment: (i) there is at least one i for which F_i is not in an accepting state, and (ii) there does not exist a transition set S such that the sequence $(S_1, \ldots, S_b, S)$ is an execution-prefix. In other words, at least one FSA is permanently blocked at the end of this segment.

`open` *type:* boolean *default:* `nil` (false)

Requirement: By itself, this does not add any requirements. It is used in conjuction with `ends-with` to alter the requirements associated with that parameter. See `ends-with`.

`starts-after` *type:* symbol list *default:* `nil` (empty list)

Requirement: For each i for which A_i has at least one symbol in at least one of the sets specified in the symbol list: consider the execution-prefix of T_i obtained by projecting the entire execution-prefix. Then at the beginning of the segment, T_i must be in a state which has at least one incoming transition labeled by a symbol in at least one of the sets specified in the symbol list.

`ends-with` *type:* symbol list *default:* `nil` (empty list)

Requirement: For each i for which A_i has at least one symbol in at least one of the sets specified in the symbol list, if we take the projection of the segment onto T_i, then (i) this projected segment is nonempty, and (ii) the label of the last transition in the projected segment is an element of `ends-with`. In addition, if `open` is false (which it is by default), then we also require (iii) that the label of every transition in the projected segment other than the last is *not* in `ends-with`.

`require` *type:* symbol list *default:* `nil` (empty list)

Requirement: For each symbol spec in the symbol list, each event in the set corresponding to that symbol spec must occur at least once as a label of a transition in the segment. *Note*: the meaning of `require` is different for infinite traces; see Section 6 below.

`force` *type:* symbol list *default:* `nil` (empty list)

Requirement: Exactly the same as `require`. But unlike `require`, the meaning does not change for infinite traces; see Section 6.

`forbid` *type:* symbol list *default:* `nil` (empty list)

Requirement: None of the events in the sets of the symbol list can occur as the label of a transition in the segment.

`restrict` *type:* restriction list *default:* `nil` (empty list)

Requirement: Each restriction defines an equation or inequality which we require to hold. The grammar is as follows:

```
<restriction> ::= '(' <relation> '(' 'total' { <symbol spec> } ')'
                                  <number> ')'
<relation>    ::= '<=' | '=' | '>='
```

This is interpreted as follows: each symbol represents the number of times that event occurs as the label of a transition in the segment; we total up these values over all symbols in the symbols specs, and we require that the resulting value be either less than or equal to, equal to, or greater than or equal to `<number>`, depending on whether `<relation>` is '`<=`', '`=`', or '`>=`'. For example,

```
(<= (total "a" "b" "a" "a" "c") 15)
```

means $3a + b + c \leq 15$, where here a represents the number of occurences of a transition labeled by `a` in the segment, etc.

`constraints` *type:* constraint list *default:* `nil` (empty list)

Requirement: This is similar to restrict, but more general. The grammar is as follows:

```
<constraint> ::= '(' <relation> <expr> <expr> ')'
<expr>       ::= <number>
                 | <symbol>
                 | '(' 'aux' <name> ')'
                 | '(' 'traversals' '(' <node> <symbol> <node> ')' ')'
                 | '(' <op> <expr> { <expr> } ')'
<node>       ::= '(' <number> <name> <number> ')'
<op>         ::= '+' | '-' | '*'
```

In a `<node>`, the first `number` is a positive integer representing an interval, the `name` is the name of an FSA, and the second `number` is the identification number of a state in that FSA. In the list following "`traversals`", the two nodes should have the same interval numbers and be in the same FSA; this list then represents a particular transition in that FSA in the specified interval, and the expression

```
(traversals (<node> <symbol> <node>))
```

represents the total number of times that transition is taken in the specified interval. (*Note:* if there are two (or more) transitions between the same two nodes with the same label, then only one of them will be selected.) Note that the specified interval does not have to be I (the interval in which this clause occurs). As usual, a `<symbol>` by itself just represents the total number of occurences of transitions in I labeled by that symbol.

The "`aux`" expression is used to define an auxiliary variable which can be used in any constraint (in this interval or another). There is no need to declare the auxiliary variable. For example:

```
(defquery "t" "nofair"
  (omega-star-less
   (sequence
    (interval
     :initial t
     :constraints
     '(
       (= (aux "a") "call(t1;t2.a)")
       (= (aux "b") "call(t1;t2.b)")
       (>= (aux "b") 1)
       (>= (aux "a") (* 3 (aux "b")))))
    (interval
     :constraints
     '(
       (= "call(t1;t2.a)" (* 2 (aux "a")))
       (= "call(t1;t2.b)" (* 3 (aux "b"))))))))
```

This describes executions of two segments in which:

(i) In the first segment, the number of calls to entry `b` is at least 1, and the number of calls to entry `a` is at least 3 times the number to entry `b`.
(ii) In the second segment, the number of calls to entry `a` is exactly 2 times the number of such calls in the first segment, and the number of calls to entry `b` is exactly 3 times the number of such calls in the first segment.

Notice that `(aux "a")` refers to the same value no matter which interval it occurs in, whereas the value of `"call(t1;t2.a)"` depends upon the interval: it represents the number of calls to entry `a` in the segment corresponding to the interval in which it appears.

Note: In INCA, one may use any name one likes for an auxiliary variable, as long as the name does not conflict with one already in use by INCA. To see the auxiliary variable names INCA uses for a particular system, use the INCA "booklet" command.

`costs` *type:* cost-spec *default:* `"ILP-variable-unit"`

Requirement: This does not add any requirements; instead, it is used to define or modify the objective function in the ILP system generated by INCA, which can impact (a) how fast the system can be solved by the ILP tool, and (b) the nature of the counterexample produced if the system has a solution. The grammar is as follows:

```
<cost spec> ::= <standard objective>
              | '(' 'costs' '('
                        ['(' ':base' <number> ')']
                        { '(' <symbol spec> <number> ')' } ')' ')'
```

This specifies the cost function by starting with a base value for each ILP variable and then adding a given value if it represents the occurrence of a symbol in one of the symbol specs.

```
<standard objective> ::= '"ILP-variable-unit"'
                       | '"event-symbol-unit"'
                       | '"connect-arc-unit"'
                       | '"connect-arc-unit-self-loops-bad"'
                       | '"connect-arc-unit-self-loops-real-bad"'
                       | '"random"'
```

These are interpreted respectively as

- "All ILP vars have weight 1."
- "Event occurrence has weight 1."
- "Connect arcs have weight -1."
- "Connect arcs and self loops have weight -1."
- "Connect arcs have weight -10, self loops have weight -3, and all other arcs have weight 1."
- "Each ILP variable has random weight between 0 and `*max-capacity*`."

The exact meanings of these objective functions are beyond the scope of this documentation, as they have no bearing on the set of executions matching the query.

6. Infinite Traces

By an *infinte execution* we mean an infinite sequence of sets of transitions:

$$S_1, S_2, \dots$$

for which (i), (ii), and (iii) of Section 3 hold and also

(iv') For all $i = 1, \dots, n$: the projection of the sequence onto T_i is either (i) finite and is a complete execution of F_i (i.e., F_i is in an accepting state at the end), or (ii) is infinite and passes through an accepting state of F_i infinitely often.

A sequence in which the final interval is declared *perpetual* represents a set of infinite executions. In order for the sequence to be valid, only the final interval may be declared perpetual. Also, the parameter `progress` must be `nil` (false) in a perpetual interval.

`perpetual` *type:* boolean *default:* `nil` (false)

Requirement: If true, this interval is declared perpetual.

When does an infinite execution match the sequence in this case? If, and only if, there are positive integers $1 \leq j_1 \leq \dots \leq j_k$ such that the execution segments

$$(S_{j_1}, ..., S_{j_2-1}), (S_{j_2}, ..., S_{j_3-1}), \dots, (S_{j_k}, \dots)$$

satisfy all the interval requirements. The requirements are essentially the same as in the finite case, with the following modifications *for the final interval and segment only*:

`require` *type:* symbol list *default:* `nil` (empty list)

Requirement: For each event e in the union of the sets specified in the symbol list, let A_i be the alphabet containing e, and consider the projection of the infinite segment onto T_i. Then we require that either (i) the projection is empty (i.e., F_i has terminated), or (ii) e is the label of at least one transition in the projection.

If we wish to require that an event e occurs in the perpetual segment in any case, use `:force`.

`ends-with` *type:* symbol list *default:* `nil` (empty list)

Requirement: For each i for which A_i intersects notrivially the union of the sets specified in the symbol list, if we take the projection of the segment onto T_i, then (i) this projected segment is nonempty *and finite*, and (ii) the label of the last transition in the projected segment is an element of at least one of the sets specified in the symbol list. In addition, if `open` is false (which it is by default), then we also require (iii) that the label of every transition in the projected segment other than the last is *not* in any set specified in the symbol list.

7. Symbols

A *symbol spec* specifies a set of symbols. It provides a convenient way for specifying a set without having to explicity list each element. INCA expands the symbol spec in a "pre-processing" stage into an explicit list of symbols. The grammar is as follows:

```
<symbol spec> ::= <symbol>
                | '(' 'rendezvous' <string> ')'
                | '(' 'rend' <string> ')'
                | '(' 'contains' <string> ')'
                | '(' 'prefix' <string> ')'
                | '(' 'or' <symbol spec> { <symbol spec> } ')'
                | '(' 'and' <symbol spec> { <symbol spec> } ')'
```

The semantics for each case are as follows:

`"X"`: A symbol by itself just represents the set of one element containing that symbol.

`(rendezvous "X")`: This represents the set of two elements `"accept(X)"` and `"call(X)"`.

`(rend "X")`: This represents the set of all elements in the alphabet which begin with either `"accept(X"` or `"call(X"`. It is equivalent to

```
(or (prefix "accept(X")) (prefix "call(X")))
```

`(contains "X")`: This represents the set of all elements in the alphabet which contain the substring `"X"`.

`(prefix "X")`: This represents the set of all elements in the alphabet which begin with the substring `"X"`.

`(or a b)`: Here `a` and `b` are symbol specs. This represents the union of the set specified by `a` and the set specified by `b`.

`(and a b)`: Here `a` and `b` are symbol specs. This represents the intersection of the set specified by `a` and the set specified by `b`.

Example 7.1 (Rendezvous with parameters)**.** In SEDL, as in Ada, entries may have parameters which allow data to be passed from one task to another during a rendezvous. The INCA front end deals with this by creating a separate rendezvous for each possible set of values of the parameters. (Since in SEDL all types are finite, there are only a finite number of these.) For example, suppose there is a system with two tasks, `t1` and `t2`, and `t2` has an entry `e` with a paramter that takes values in an enumerated type `{a1,a2,...,a20}`. INCA will essentially replace this rendezvous with 20 "ordinary" (parameter-less) rendezvous. Assuming all 20 of these calls actually occur at some place in the SEDL program, this means the alphabet for `t1` will contain the 20 events

`"call(t1;t2.e;a1)"`, ..., `"call(t1;t2.e;a20)"`

and the alphabet for `t2` will contain the corresponding matching events

`"accept(t1;t2.e;a1)"`, ..., `"accept(t1;t2.e;a20)"`.

In this situation, `(rend "t1;t2.e")` represents all 40 of these events. On the other hand, `(rendezvous "t1;t2.e")` specifies the empty set, while

`(rendezvous "t1;t2.e;a1")`

represents two events (one call and one accept). Now `(rend "t1;t2.e;a1")` represents 22: not only `"accept(t1;t2.e;a1)"` and `"call(t1;t2.e;a1)"`, but also

"accept(t1;t2.e;a10)", ..., "accept(t1;t2.e;a19)",
"call(t1;t2.e;a10)", ..., "call(t1;t2.e;a19)"

The moral is, one must be very careful when using these symbol macros!

8. Examples

Example 8.1 (Global Absence)**.** Suppose we have a system of communicating FSAs in which the alphabet of one FSA, say F_i, contains the symbol "P". Consider the following query:

```
(defquery "global_absence_of_p" "nofair"
  (omega-star-less
    (sequence
      (interval :initial t :ends-with '("P")))))
```

Suppose $S_1, \ldots, S_m$ is an execution-prefix for the system, and consider the projection $t_1, \ldots, t_k$ of the execution-prefix onto F_i. Then $S_1, \ldots, S_m$ will match the query if, and only if, t_k is labeled by "P", and t_j is not labeled by "P" for $j < k$.

If there is no execution-prefix matching this query then there can be no execution-prefix of the system in which event "P" occurs. For if event "P" did occur in an execution-prefix, there must be a first occurence of "P" in the prefix, and cutting off the prefix after that first occurence would yield a prefix which matches the query.

Recall that in INCA, a query is used to describe the set of executions which violate the property one wishes to verify. So if we show that the query has no matches, we have verified that "P" will not occur in any execution-prefix of the system, hence the name of the property, "Global Absence of P."

Example 8.2 (Absence of P Before R)**.** Suppose we have a system of communicating FSAs in which two distinct symbols "P" and "R" occur. Say "P" belongs to the alphabet of F_i and "R" to that of F_j. (It is possible that $i = j$.) Consider the query:

```
(defquery "absence_of_p_before_r" "nofair"
  (omega-star-less
    (sequence
      (interval :initial t :ends-with '("R") :require '("P")))))
```

Suppose $S_1, \ldots, S_m$ is an execution-prefix for the system, and consider the projections onto F_i and F_j respectively:

$$t_1, \ldots, t_k \qquad\qquad u_1, \ldots, u_l.$$

(These will be the same if $i = j$.) Then $S_1, \ldots, S_m$ will match the query if, and only if, all of the following hold:

(i) For at least one a ($1 \leq a \leq k$), t_a is labeled by "P".
(ii) u_l is labeled by "R".
(iii) u_a is not labeled by "R" for $a < l$.

If we can show that there is no execution-prefix matching this query, then we have verified the following property of our system:

> *Absence of P Before R*: In any execution-prefix in which "R" occurs, there can be no "P" before the first "R".

For if there is a prefix $S_1, \ldots, S_m$ which violates the property, then we can cut off the prefix after the first occurence of `"R"` and the result will be a prefix which satisfies (i) through (iii).

Actually, we might have verified something even stronger than the property, for it is possible that for some systems there are execution-prefixes which match the query but are not violations of the property. Specifically, as long as i does not equal j, it is conceivable there is an execution-prefix of the system in which the `"P"` occurs *after* the `"R"`, but still satisfies (i), (ii), and (iii). The simplest possible example is where `"P"` and `"R"` are local events in different tasks and the execution-prefix has the form $\{u\}, \{t\}$, where $\text{label}(u) =$ `"R"` and $\text{label}(t) =$ `"P"`. This prefix matches the query, but it clearly does not violate the property as it is stated above.

So in some cases our query over-estimates the set of all possible violations of the property. That is fine, because if we show there is no execution-prefix matching the query, there can certainly be none which violates the property. However, if an analysis tool such as INCA produces a "counterexample," i.e., an execution-prefix which matches the query, one would have to check it to determine whether or not it represents an actual violation of the property. In other words, this query is a conservative, but not precise, representation of the space of violating executions.

(However, in the case where `"P"` and `"R"` are in the same task, this query is actually precise.)

Example 8.3 (Existence of `P` Between `Q` and `R`)**.** Suppose we have a system of communicating FSAs which involves events `"P"`, `"Q"`, and `"R"`, where `P` does not equal `Q`, and `P` does not equal `R`. (But we do allow `Q` = `R`.) We wish to verify:

> *Existence of `P` between `Q` and `R`*: If $S_1, \ldots, S_m$ is any execution-prefix, $1 \leq i < k \leq m$, `"Q"` labels a transition in S_i, and `"R"` labels a transition in S_k, then there exists j such that $i \leq j \leq k$ and `"P"` labels a transition in S_j.

Consider the query:

```
(defquery "existence_of_p_between_q_and_r" "nofair"
  (omega-star-less
    (sequence
      (interval :initial t :open t :ends-with '("Q"))
      (interval :ends-with '("R") :forbid '("P")))))
```

An execution-prefix $S_1, \ldots, S_m$ will match the query if, and only if, there is some l, $1 \leq l \leq m+1$, such that the following three conditions hold:

(i) The projection of $S_1, \ldots, S_{l-1}$ onto the task containing `"Q"` has its last transition labeled by `"Q"`.
(ii) The projection of $S_l, \ldots, S_m$ onto the task containing `"R"` has its last transition labeled by `"R"`, and no other transition in the projection is labeled by `"R"`.
(iii) $S_l, \ldots, S_m$ contains no transition labeled `"P"`.

We claim that if there is a violation of the property then there will be a prefix which matches the query.

To see this, suppose $S_1, \ldots, S_m$ is a violation. That means there are integers i and k with $1 \leq i < k \leq m$, such that `"Q"` labels a transition in S_i, `"R"` labels a transition in S_k , and no transition in any S_j $(j = i, \ldots, k)$ is labeled `"P"`.

Let k' be the least integer such that $i < k' \leq k$ and `"R"` labels a transition in $S_{k'}$. (We know there is such an integer, because k itself is one.)

Consider the execution-prefix $S_1, \ldots, S_{k'}$. Let $l = i + 1$. Then it is clear that (i) through (iii) hold for this prefix with this l, and our claim is proved.

Hence, if we show the query has no matches, we have verified that the system satisfies "Existence of `P` Between `Q` and `R`."

References

[1] J. C. Corbett. The S-expression design language (SEDL). ICS-TR-93-02, Information and Computer Science Department, University of Hawaii at Manoa, 1993.

[2] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, January 1995.

Laboratory for Advanced Software Engineering Research, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610

E-mail address: `siegel@cs.umass.edu`