Automatic Generation of Inequality Systems for Constrained Expression Analysis*

George S. Avrunin Ugo Buy James Corbett

May 25, 1990

1 Introduction

This report describes a prototype tool for automating the generation of systems of inequalities, and for aiding in the interpretation of the solutions of such systems, in the analysis of *constrained expressions*. The constrained expression approach to the analysis of concurrent and distributed software systems is described in [4, 5]; more detailed descriptions of the use of inequalities in such analysis are given in [2, 5, 6]. This prototype implements a piece of a toolset supporting automated analysis of distributed system designs written in the CEDL design language [11], which is based on Ada. The structure of the toolset and its application are described briefly in the next section and in more detail in [4]. Detailed discussions of the other components of the toolset are given in [1, 7, 9, 12]. This document describes version 2.0 of the prototype, whose implementation has been completed recently.

Version 2.0 of the prototype extends significantly an earlier version, described in [3], and incorporates algorithms and some portions of code from that version. A number of major areas have been changed. One such change is the added capability to generate inequalities from deterministic finite-state automata (DFAs) and a hybrid form we call regular expression deterministic finite-state automata (REDFAs) in addition to regular expressions. In some cases, these automata are significantly more compact representations of event sequences than their equivalent regular expressions and thus produce much smaller inequality systems. Also, if a task expression has been run through the constraint eliminator, and thus converted to a DFA or an REDFA, no time must then be spent converting it back to a regular expression.

A second area of change is the actual generation of inequalities for regular expressions. By shifting some of the burden to the interpretation of the system and its possible solutions, the current version avoids a great deal of redundancy in the system of inequalities and therefore produces much smaller (though less sparse) systems. Our preliminary experience indicates that the systems generated by the new version typically have only about

*Research supported by National Science Foundation grant CCR-8806970 and Office of Naval Research Grant N00014-89-J-1064

one-sixth the variables and inequalities of those produced by the earlier version. Since the interpretation is done only once, after solving the system, and standard integer programming methods usually require repeated solution of non-integer programming problems derived from the original system of inequalities, this represents a significant improvement in the efficiency of the analysis.

The third area that has been changed is the user interface, which was extremely rudimentary in the earlier version. The current prototype produces an extensive report on the system of inequalities it generates and on any solutions found by the integer programming package, making it much easier for the user to interpret the results of the analysis. It also provides an interactive facility allowing the user to add inequalities to the system, something that is frequently helpful in analyzing special aspects of the behavior of a distributed software system.

A fourth area of change is that now the inequality generator can produce files that are suitable for input to the IMINOS prototype for solving integer programming systems [7]. Previously only input files for the Land and Powell package could be generated [13]. Consequently, now both the Land and Powell package and IMINOS can be used to solve the integer systems generated.

Fifth, two types of objective functions can be generated by the current prototype. One type of objective function assigns a unit cost to each ILP variable generated. Consequently, an ILP system of this kind will be aimed at minimizing the sum of the values of the variables appearing in an ILP system. The other type of objective function assigns a unit cost to each event symbol appearing in the task expressions that are input to the inequality generator. Consequently, an ILP system of this kind will be aimed at minimizing the total number of event symbols appearing in a solution. In this case, the cost associated with an ILP variable is an integer that reflects the "weight" of that variable with respect to the event symbols appearing in the derived constrained prefixes. This concept is explained in more detail later in this document. At present the prototype does not support the generation of non-unit cost objective functions. Consequently, if non-unit costs are required for a given analysis problem, the actual costs associated with each variable must be entered manually in the input file for the Land and Powell package or IMINOS.

The prototype described here has been implemented in the language VAXLISP running on the DEC VAXstation 2000 under the VMS 5.1 operating system. Moreover, the present version of this prototype has been ported to the Franz and the Lucid Common LISP environments running on the DECstation 3100, and to the Lucid Common LISP environment running on the Sun 3/60. No changes to the LISP sources have been required for portability.

The remainder of this document is organized as follows. The next section consists of an overview of the inequality generator, describing its major components and features. Section 3 discusses the main program modules in more detail, and section 4 gives documentation for LISP definitions appearing in the program.

2 Overview and Background Information

The inequality generator described here is a component of a prototype toolset supporting the constrained expression analysis of concurrent system designs written in the CEDL language. Since in normal use of the toolset, the input to the inequality generator is provided by other components of the toolset and the output of the inequality generator is intended as input to another component, we give a brief description of the toolset and its application here. For a more detailed description of the application of the toolset, we refer the reader to [4]. The individual components are described in [1], [7], [9], and [12]. A detailed discussion of the CEDL design language is given in [11].

The toolset is used to analyze concurrent system designs written in CEDL, an Ada-based design language that focuses on the expression of communication and synchronization among the tasks of a concurrent system. Designs written in CEDL are translated into constrained expressions by the *deriver* tool. Output of this tool consists of a regular expression, called a *task expression*, corresponding to each task in the design and a set of constraints, which are also regular expressions. The alphabets of these regular expressions consist of various *event symbols* corresponding to events occurring in behaviors of the concurrent system.

As discussed below, our techniques for automated inequality generation produce systems of inequalities that do not reflect the ordering of events. In the constrained expression representation of a CEDL design, the constraints that enforce the consistent use of data (i.e., make sure that the value read from a variable is the value last written to that variable) restrict the order of occurrences of certain event symbols, but not the numbers of those occurrences. Therefore, inequalities generated directly from the constrained expression representations produced by the deriver will have solutions that correspond to event sequences in which data are used inconsistently. To avoid this problem, we use a *constraint eliminator* tool that takes a task expression and constraints involving symbols appearing in that expression and produces a DFA recognizing the language consisting of the strings in the language of the task expression that satisfy those constraints. The task expression is then replaced by the DFA (which can be converted back to a regular expression or an REDFA, if desired), and the constraints are eliminated from the constrained expression. The constraint eliminator is normally applied to all task expressions whose control flow depends on the values of variables. After applying this tool, the constrained expression representation of the CEDL system consists of a set of task expressions and task DFAs, together with the remaining constraints.

The inequality generator tool described in this report accepts a set of task expressions and task DFAs produced in this fashion, and generates a system of linear inequalities reflecting the constrained expression and assumptions or queries about the CEDL system that are added by the user of the toolset. For example, the user might add an inequality corresponding to the assumption that a certain deadlock occurs. We then use an integer linear programming tool to attempt to solve this system. When a solution is found, we use a *behavior generator* tool to try to produce a corresponding string of events representing

a real behavior of the CEDL system. When no solution exists, we conclude that there is no behavior satisfying the assumptions made by the user.

The prototype inequality generator contains two major components. The first component, whose top-level entry point is the LISP function analyze, generates a system of integer inequalities from a set of regular expressions and DFAs derived from a CEDL design. In particular, each regular expression represents one of the task expressions appearing in a CEDL design. Each DFA represents the automaton that accepts the strings that can be generated from a task expression appearing in a CEDL design. Consequently, each task expression in a CEDL design can be input as a regular expression, a DFA, or an REDFA to the inequality generator.

The second component, whose top-level entry point is the function generate-report, produces a report describing the system of generated inequalities. The inequalities express relations on the number of times event symbols contained in the task expressions and DFAs may appear in the behaviors of the distributed system described by the CEDL design.

Since the system of generated inequalities is linear and the variables appearing in the inequalities must take non-negative integer values, this approach is amenable to the use of well-known ILP (Integer Linear Programming) techniques for solving a system of inequalities [15]. Since the inequalities produced by this tool give only a partial representation of the constrained expression, not every solution of the system necessarily corresponds to a constrained prefix. Those aspects of the constrained expression that are not reflected in the system of inequalities will be pointed out in section 3.4.

2.1 Input to the Inequality Generator

Input to the inequality generator component consists primarily of a list of regular expressions and DFAs representing the tasks appearing in a CEDL design. The regular expressions are expressed in a prefix notation in which the regular expression operators are denoted by the LISP strings "SEQUENCE", "OR", and "STAR" and event symbols are denoted by arbitrary LISP strings [8]. Thus, for example, the task expression

$$beg\_loop(\mathrm{T})\left(\begin{array}{l} beg\_rend(\mathrm{S},\mathrm{T.entry})end\_rend(\mathrm{S},\mathrm{T.entry}) \\ starve_a(\mathrm{T.entry})stop(\mathrm{T}) \end{array}\right)^*$$

would be written as

```
("SEQUENCE" "beg_loop(T)"
    ("STAR" ("SEQUENCE" "beg_rend(S,T.entry)"
                        "end_rend(S,T.entry)"))
    "starve_a(T.entry)" "stop(T)")
```

The abstract data type DFA from the constraint eliminator is used to process the DFAs which are input in their encoded forms as lists and then decoded into a more efficient

internal representation using hash tables. The list encoding is shown below. For more information on the internal form of the DFA, see [9].

```
DFA = (<number of states>
       <start state>
       (<accept state> <accept state> . . .)
       (<transitions for state> <transitions for state> . . .))
```

where

```
<transitions for state> = (<state name> <transition list for state>)
<transition list for state> = (<transition> <transition> . . .)
<transition> = (<letter> <next state>)
```

Additional input to the inequality generator specifies whether, after the inequalities for the input task expressions and DFAs have been generated, additional inequalities need to be specified manually by a user of the prototype. In addition, input to the inequality generator can be used to indicate whether the system of generated inequalities is to be written in a file to be input to an ILP system, such as the Land and Powell package or the IMINOS prototype [7, 13]. Last, input to the inequality generator specifies the type of cost vector to be generated, that is, whether ILP variables are unit cost or event symbols are unit cost.

Note, however, that all the input parameters to function analyze are optional in order to make this function easier to invoke. When an argument to analyze is not supplied, a default value stored in one of a set of global variables is used. In particular, the list of input task expressions defaults to the value of the global variable *task-expressions*; the list of input task DFAs defaults to the value of the global variable *task-dfas*; the parameter controlling the generation of an input file for an integer programming package defaults to the value of the variable *default-ILP-system*; and the type of cost vector is determined by the variable *default-cost-vector*. All these LISP objects are documented in section 4.

2.2 The Inequality Generator

The inequality generator performs the following sequence of activities for a given set of input task expressions and DFAs. First, each task expression is recursively analyzed to generate inequalities reflecting the semantics of a regular expression with respect to its derived event sequences. Second, each task DFA is analyzed state by state to generate inequalities reflecting the semantics of the automaton. Third, inequalities are generated that reflect the semantics of some of the constraints appearing in a constrained expression for a CEDL design. The generation of these inequalities is based on interpreting the event symbols appearing in the input task expressions and DFAs in a way that is consistent with a constrained expression derived from a CEDL design. Fourth, depending on the input to the generator, an interactive query can be performed to allow a user of the prototype

to enter additional inequalities by hand. Fifth, depending on additional input to the inequality generator, a file can be output in the input format required by the Land and Powell package or the IMINOS prototype. Each of these five activities is described in detail in the next section.

2.3 The Report Generator

The report generator component is intended to aid a human user of this prototype in the analysis of the behavior represented by a given solution to the inequality system corresponding to the input task expressions and DFAs. If the Land and Powell package or IMINOS is used to solve the system of inequalities and an integer solution is found, output from the ILP system can be used to include information about the solution in the report.

The report generator takes as input data structures set up by the inequality generator to output a report describing an inequality system. Consequently the report generator can be invoked in a given LISP session only after the inequality generator has been executed. Additional input to the report generator is the name of an output device, such as a file or a terminal, to which a report on the inequality system is written.

Moreover, the name of an err file produced by the Land and Powell package, or the name of a sol file produced by IMINOS can be provided as an input to the report generator. In this case, the report generator first parses the file in an effort to find the solution to the inequality system computed by the ILP system. If such a solution is parsed successfully, the values of the solution are incorporated in the report. The two activities of report generation and parsing are described separately in the next section.

3 Major Modules in the Prototype

The prototype consists of a number of program modules that perform various activities related to inequality generation and to report generation. Each of these program modules is described in detail in a separate subsection of this section. For each module the major LISP functions contained in the module, the data structures used and modified by the module and the input/output behavior of the module are described. In addition, this section gives an assessment of the expressiveness of the inequality systems generated with respect to the corresponding constrained expression-based representation of a concurrent program. In particular, the ensuing subsections discuss the following issues:

1. Generation of inequalities reflecting regular expression semantics
2. Generation of inequalities reflecting finite automata semantics
3. Generation of inequalities reflecting CEDL constraints
4. Assessment of expressiveness of the inequality systems generated

5. Manual insertion of inequalities
6. Generation of input files for the Land and Powell package and IMINOS
7. Generation of report describing inequalities
8. Translation of REDFA solution information to DFA solution information
9. Low-level utilities

The modules describing the inequality generation component of this prototype are described in subsections 1, 2, 3, 5, and 6, while the subsection 7 describes the module constituting the report generation component. A discussion of the expressiveness of the inequality systems generated is given in the fourth subsection. The eighth subsection describes a module that translates solution information for an REDFA into solution information for the corresponding DFA. This solution information is used by the behavior generator tool. The last subsection describes a module that implements abstract data types used by all the other modules.

3.1 Generation of Inequalities Reflecting Regular Expression Semantics

During this activity inequalities are generated that reflect the semantics of regular expressions with respect to the event sequences that can be derived from each input task expression. To this end, non-negative integer variables are associated with given subexpressions of an input task expression to denote the number of occurrences of the corresponding subexpressions in the derived event sequences. For instance, if variable x_i denotes the number of occurrences of the i-th operand in an alternation operation and variable x denotes the number of occurrences of the alternation subexpression as a whole, the following equality can be generated:

$$x - x_1 - ... - x_i - ... - x_n = 0 \tag{1}$$

This equality reflects the fact that the sum of the occurrences of the alternation operands must be equal to the number of occurrences of the whole alternation subexpression in every derived event sequence.

More generally, a non-negative integer variable is associated with each operand in an alternation or Kleene star operation appearing in an input task expression. A linear equality similar to (1) above is generated for each alternation expression appearing in an input task expression. In addition, for every input task expression T_i, a variable x_{T_i} is associated with the whole expression. The following equality is then generated to capture the fact that each input task expression models a CEDL task to be executed once:

$$x_{T_i} = 1 \tag{2}$$

For a given variable x, the "**range** of variable x" is defined to be the subexpression E associated with x, excluding the ranges of variables nested within E. Since no equalities

and variables are generated for concatenation operands, the range of a variable x may contain multiple event symbols. For instance, given the following alternation subexpression:

("OR" ("SEQUENCE" "a" "b" "c") "d") (3)

variable x_1, could be associated with the whole alternation subexpression, whereas variables x_2 and x_3 could be associated with the first and the second operand in the alternation subexpression, respectively. In this case the range of variable x_2 consists of the concatenation subexpression that includes the event symbols a, b, and c and the range of variable x_3 is the event symbol d.

The top level function contained in this module is analyze-expr, which is called by analyze and invokes analyze-seq, analyze-or and analyze-star to process each different type of subexpression. The output of this module consists primarily of a set of inequalities that are stored in three LISP lists, namely lhs-list, rel-list and rhs-list.

Each element in lhs-list is a LISP string that contains a linear expression on ILP variables representing the left-hand side of an inequality. Each variable appearing in the LISP string is written as N-x, where x denotes the subscript of the variable. Elements in rel-list can be one of the LISP strings "<=", "=" or ">=" to indicate the relation type of an inequality. Elements in rhs-list are LISP strings containing a non-negative integer to denote the righ-hand side of an inequality.

The elements in the three lists are arranged consistently by keeping information about a given inequality in the same position in each list. For instance, the left-hand side, relation type, and right-hand side for inequality

$$x_1 - x_2 - x_3 \leq 1$$

would be represented by LISP strings "N-1 - N-2 - N-3", "<=", and "1" in lhs-list, rel-list, and rhs-list, respectively. In addition, the position of these LISP strings would be the same in each of the corresponding lists.

An additional output of this module is a set of structures describing the ILP variables generated. In particular, for each variable the event symbols contained in the range of the variable are stored in the LISP list ILP-variable-bags. Entries in this list are LISP lists whose first element is a LISP symbol denoting an ILP variable and whose remaning elements are LISP strings denoting the event symbols appearing in the range of the given ILP variable. For instance, in the case of variable x_2 associated with the first operand of the alternation subexpression (3) above, the following element would appear in ILP-variable-bags: (N-2 "a" "b" "c"). The term *bag of an ILP variable* will be used in the remainder of this document to denote a list of the event symbols appearing in the range of the variable.

The LISP list ILP-variable-expressions stores, for each ILP variable x, the *environment* of variable x and the subexpression associated with variable x. The environment of variable x is defined as the variable associated with the range directly enclosing the range of variable x. If no such variable exists (i.e. variable x is associated with a whole task

expression), the environment of variable x is defined to be the LISP symbol TOP-LEVEL. For instance, the environment of variables x_2 and x_3 in example (3) above is variable x_1. Each element of ILP-variable-expressions is a LISP list L containing the following items:

1. LISP symbol denoting an ILP variable
2. LISP symbol denoting the environment of the variable (i.e. another ILP variable or TOP-LEVEL)
3. a number of LISP objects resulting from splicing the subexpression associated with the ILP variable under point (1) into list L.

In the case of example (3) above, variable x_2, would have the following entry in ILP-variable-expressions:

(N-2 N-1 "SEQUENCE" "a" "b" "c")

The hash table ILP-symbol-var-table stores the variable associated with each subexpression consisting of a single event symbol. In the above example, when the subexpression "a" is parsed by analyze-expr, not only would "a" be placed in the variable bag for N-2, but an entry would be added to ILP-symbol-var-table with key "a" and value N-2. For this to work, different occurrences of an event symbol must have different entries in ILP-symbol-var-table. This is accomplished by making ILP-symbol-var-table an eql hash table and insuring that different occurrences of event symbol strings in the expressions are actually different strings (i.e., not eql). Reading the expressions from a file guarantees this, and the constraint eliminator is careful to produce expressions with this property as well.

Finally, the non-negative integer LISP variable n-count, written by this module, is used to record the total number of ILP variables generated.

3.2 Generation of Inequalities Reflecting Deterministic Finite Automaton Semantics

We have found [10] that using DFAs and REDFAs rather than regular expressions to represent the behavior of tasks in the concurrent system leads to more compact representations and smaller systems of inequalities. In addition, generating inequalities from the DFAs and REDFAs produced by the constraint eliminator means that it is not necessary to convert DFAs back to regular expressions, an expensive process. We now describe the generation of inequalities from DFAs and REDFAs.

Regular expression deterministic finite automata (REDFAs) are DFAs whose arcs are labeled with regular expressions such that the following conditions are satisfied:

1. Given a regular expression re, define $START(re)$ to be the set of symbols which can begin strings generated by re. For each state in the REDFA with transitions on the expressions $re_1, re_2, \ldots, re_n$, we must have

$$START(re_i) \cap START(re_j) = \emptyset,\ i \neq j$$

2. Each expression in the REDFA is deterministic in the following way. For each disjunction of subexpressions $re_1, re_2, \ldots, re_n$ in the expression, i.e.

$$(\mathsf{OR}\ re_1\ re_2\ \ldots\ re_n)$$

we must have

$$START(re_i) \cap START(re_j) = \emptyset,\ i \neq j$$

Clearly an ordinary DFA is just a special case of an REDFA where the arc labels are all trivial one-symbol expressions. We will describe how to generate inequalities that reflect the semantics of REDFAs.

Create a variable for each arc in the REDFA. These are called *transition* variables and represent the number of times that arc was crossed in the accepting path taken through the REDFA. At each state that is neither starting nor accepting, the following relation must hold: the number of times the state was entered equals the number of times the state was exited. For instance, if $x_1, x_2, \ldots, x_n$ are the transition variables into a given state and $y_1, y_2, \ldots, y_m$ are the transition variables out of that same state, then we must have:

$$y_1 + y_2 + \ldots + y_m - x_1 - x_2 - \ldots - x_n = 0$$

For the start state, the number of transitions out of the state will exceed the number of transitions into the state by one, so the above inequality will have a one on the right hand side. For the accepting state that the computation ends on (call it the *terminal* accept state), the number of transitions into the state will exceed the number out by one. To express this, we create one *accept* variable for each accepting state, which is one if that accept state is the terminal accept state, else it is zero. We then create an equality stating that the sum of all the accept variables is one, which forces exactly one of the non-negative accept variables to be one. At each accepting state, we add the value of the accepting variable for that state to the left hand side of the inequality for that state. This will force the number of transitions into the terminal accepting state to exceed the number of transitions out of that state by one.

The algorithms described above are implemented using the DFA abstract data type from the constraint eliminator and two global data structures. The list accept-variable-list stores a list of the accept variables for this REDFA. The hash table ILP-transition-var-table stores the transition variable names for all the tasks. The variable for a transition out of a state s on an expression e in task t is hashed into the table by the three-element list $(t\ \ s\ \ e)$. The function transition-variable takes a task, a state, and an expression and returns the transition variable for the transition in that task, from that state, on that expression. If the task-state-expression triple is already in the hash table, the variable name stored there is simply returned. If the triple is not found, a new transition variable is created and inequalities are created for the expression using the algorithm described in section 3.1. This is accomplished by calling the function analyze-expr on the expression labeling the arc, where the transition variable is the environment for the expression. Thus

deriving inequalities for REDFAs amounts to deriving inequalities for each of the regular expressions that label the arcs and then generating additional inequalities that relate the values of their top level variables.

Inequality generation for the task DFAs and REDFAs is accomplished as follows. For each DFA or REDFA on the list *task-dfas*, the function analyze calls the function analyze-state on each state. Function analyze-state calls the function transition-variable for each transition in or out of the state to retrieve the variable for that transition, or to create a variable for the transition and analyze the regular expression that labels the arc. It then assembles these variables into an equality stating that the number of times the state is entered must equal the number of times the state is exited, as described above. If the state being analyzed is the start state, then the right hand side of the equality is set to one, so the exits from the state exceed the entrances to the state by one. If the state being analyzed is an accept state, an accept variable is created for the state and added to both accept-variable-list and the left hand side of the equality for that state, so the entrances to the state may exceed the exits by one if this state is selected as the terminal accept state. When all states have been analyzed, an equality is created that says that the sum of the accept variables in accept-variable-list is one, to force exactly one accept state to be selected as terminal.

The data structures that store information about the variables are also updated. ILP-var-types, which stores a type for each variable, contains either of the LISP symbols TRANSITION for transition variables or ACCEPT for accept variables. ILP-variable-expressions, which stores the environment and subexpression associated with a variable for a regular expression, stores similar information for the new types of variables. The environment of a transition variable is a two-element list containing the task name and the name of the state from which the transition occurs. The subexpression associated with a transition variable is the regular expression that labels the transition arc. For accept variables, the environment is a two-element list containing the task name and the name of the accept state associated with the variable; the subexpression is NIL.

Details about the DFA abstract data type and its legal operations can be found in [9]. The operations used in this analysis are get the list of states, get the transition list for a state, get the start state, and get the list of accept states.

3.3 Generation of Inequalities for CEDL Constraints

The top-level function in this module, cedl-constraints, generates inequalities that capture the semantics of some of the CEDL constraints. Input to cedl-constraints is primarily given by the structure symbol-table, a hash table associating each event symbol appearing in the input task expressions and DFAs with a list of the ILP variables whose bags contain the event symbol. Information in symbol-table is used to compute the total number of occurrences of event symbols in event sequences that can be derived from a given regular expression or DFA. Indeed, the total number of occurrences of an event symbol in an event string can be defined as the sum of the ILP variables whose bags contain the symbol, taking

into account multiple occurrences of an event symbol in an ILP variable bag. The functions that create, access and modify symbol-table are described in the subsection on low-level utilities. The output produced by this module is given by suitable additions to the LISP lists lhs-list, rel-list and rhs-list, as described previously.

The function cedl-constraints goes through the hash table symbol-table entry by entry. If the key belongs to any one of four types of event symbols, an appropriate function is called with the key as argument to generate an inequality corresponding to a constraint. When the key is a *call*(_,_) symbol, it calls the function cedl-call-const; when the key is an *end_rend*(_,_) symbol, it calls the function cedl-erend-const; when the key is a *kill_rend*(_) symbol, it calls the function cedl-krend-const; when the key is a $starve_a$(_) symbol, it calls the function cedl-starve-const.

Given a *call*(T,E) symbol as argument, the function cedl-call-const searches symbol-table for the variable giving the total number of corresponding *beg_rend*(T,E) symbols. It then adds an equation to the inequality lists stating that the number of these *call*(T,E) symbols is equal to the number of *beg_rend*(T,E) symbols.

Given an *end_rend*(T,E) symbol as argument, the function cedl-erend-const searches symbol-table for the variable giving the total number of corresponding *resume*(T,E) symbols. It then adds an equation to the inequality lists stating that the number of these *end_rend*(T,E) symbols is equal to the number of *resume*(T,E) symbols.

Given a *kill_rend*(E) symbol as argument, the function cedl-krend-const searches symbol-table for the variables giving the total numbers of *dead_rend*(T,E) symbols, where T ranges over the tasks calling entry E. It then adds an equation to the inequality lists stating that the number of *kill_rend*(E) symbols is equal to the sum of the numbers of these *dead_rend*(T,E) symbols.

Given a $starve_a$(E) symbol as argument, the function cedl-starve-const searches symbol-table for the variables giving the total numbers of $starve_c$(T,E) symbols, where T ranges over the tasks calling the entry E. For each of the $starve_c$(T,E) symbols found, cedl-starve-const adds an inequality to the lists. Each inequality states that the sum of the number of occurrences of the $starve_a$(E) symbol and of the number of occurrences of a $starve_c$(T,E) symbol is at most 1.

3.4 Assessing the Expressiveness of the Inequality Systems

The inequalities generated by this prototype are intended to capture various aspects of a constrained expression representation of a concurrent program. In particular, the first two modules, described in 3.1 and 3.2 respectively, generate inequalities aimed at capturing the semantics of the task expressions and DFAs that can be derived from a CEDL design. The third module, described in 3.3, generates inequalities intended to capture the semantics of some of the constraints that can be derived from a CEDL design. Nonetheless, it should be noted that an inequality system generated in this way does not reflect the complete semantics of the input task expressions, DFAs, and their derived constraints due to the reasons listed below. As a result, the inequality system may have *spurious* solutions, that

is, solutions that do not correspond to an actual behavior of the original CEDL design. Spurious solutions are detected using the behavior generator tool and can arise for the following reasons.

1. Information about the relative ordering of event symbols in a derived event sequence is not represented in the inequality system. As a result a solution to the inequality system may reflect an ordering of event symbols that is inconsistent with the input task expressions. In this case the solution does not correspond to an actual system behavior for the CEDL designs under consideration.

2. Dataflow information in a CEDL design is not taken into account by the inequality generator. This is because the constraints that enforce consistent use of data involve the relative order of certain event symbols. Thus, it is possible for a solution to assume that at some point a CEDL variable has different value from the one that was last assigned to such a variable. This kind of spurious solution can be avoided by incorporating dataflow constraints into the task expressions using the constraint eliminator [9].

3. The inequality system is built in such a way that an ILP variable x associated with the operand in a Kleene star subexpression can always take any value in a solution. This captures the fact that a Kleene star operand can be repeated an arbitrary number of times in a derived event sequence. Nonetheless, this approach allows variable x to take a non-zero value even when variable x_e, the environment of x, is zero. If x_e is zero, the subexpression associated with variable x_e does not appear in a constrained prefix and therefore the Kleene star operand should not appear in the constrained prefix either. This is to say that, whenever x_e is zero, x should also be zero. This constraint could be enforced by the following quadratic inequality

$$x \cdot x_e - x \geq 0$$

   Since we are only dealing with linear inequality systems, this constraint is not enforced in the inequality systems generated at present. We have developed, but not used, a method for dealing with this problem using linear inequalities, when all the variables have upper bounds.

   This problem also arises when generating inequalities from DFAs. A cycle in a DFA can occur regardless of whether a path to it was traversed in the solution. A self loop is the simplest cycle; note that the inequality for a state will not constrain the number of times the loop is traversed at all, since the variable for the arc is both added to and subtracted from the left hand side. Any cycle will be similarly unconstrained. Given a solution, the set of arcs with nonzero transition variables represents the path through the DFA found by ILP. If a solution path contains a cycle that cannot be reached from the start state, it is clearly spurious. Fixing the problem in this case is more difficult than in the case of regular expressions. For

each possible cycle, we must generate an inequality that states that if every entrance to that cycle is zero, then the cycle must be zero (an entrance to a cycle is any arc incident to a state in the cycle). Since the number of cycles can be exponential in the size of the graph, this can be very expensive, though no more expensive than if we converted the task DFAs back to regular expressions and used the methods described above, since each cycle in the DFA will be unrolled to its own Kleene star in the expression.

In summary, the first two points reflect the fact that this prototype only generates inequalities applying to a complete behavior of the CEDL system and does not capture sequencing information related to prefixes of subsequences in a behavior. The third point results from our decision to restrict ourselves to linear inequalities to simplify the integer programming problem.

3.5 Generation of User Defined Inequalities

This module consists of functions that perform input and output through the LISP stream *terminal-io*, in order to facilitate the insertion of inequalities by hand. In particular, the top-level function add-constraints performs a major loop that prints a list of options and waits for a user to enter a selection. Available choices include adding an inequality, writing an IMINOS input file, writing a Land and Powell input file and exiting the interactive session. If a user wants to add an inequality, the function add-constraint is invoked to handle the input of the inequality. If a user wants to write the IMINOS input file, function IMINOS-input is invoked and the loop exited. If a user wants to write the Land and Powell input file, the function LP-input is invoked and the loop exited. If a user wants to exit the loop, add-constraints simply returns without writing any files. Functions IMINOS-input and LP-input are described in the next subsection.

Function add-constraint consists of a main loop that prints a set of available options and waits for user input. The available choices include entering an ILP variable as a positive or a negative term in the left-hand side of an inequality, entering the ILP variables denoting the total occurences of an event symbol in the left-hand side of the inequality, selecting the relation type of the inequality (i.e. one of $\leq$, $\geq$ and $=$), entering the constant term in the right-hand side of the inequality, and exiting the loop.

The output of this module consists mostly of modifications to the lists lhs-list, rel-list and rhs-list to reflect the inequalities entered manually by an interactive user. In addition, this module uses the global structures n-count and symbol-table while performing an interactive query.

3.6 Input to the Integer Linear Programming Package

The program contains several functions that write a generated system of inequalities to a file suitable for processing by an ILP system, such as the Land and Powell package or the IMINOS prototype [7, 13]. The chosen ILP system can then be used to solve the

generated system of inequalities. The file formats for these two ILP systems are different and therefore two different sets of LISP functions are used by our prototype to generate the two kinds of input files.

The input file for the Land and Powell package consists of four sections. The first section contains header information giving the numbers of inequalities and variables and control information. The second section gives the coefficients of the objective function. The third section gives the relations and constant terms of the inequalities, while the fourth section gives the coefficients of the variables appearing in each inequality. For a detailed description of the format of the input file, see [13].

Input files for the Land and Powell package are generated by the function LP-input. This function first generates a cost vector for the problem at hand by invoking function generate-cost-vector. The cost vector is stored in the variable cost-vector. The coefficients appearing in the cost vector are usually non-positive, since the Land and Powell package always maximizes the objective function whereas we are interested in finding minimal solutions. Next, LP-input opens the file input.txt and calls the functions header-lines, objective, constant-vector, and coefficient-matrix that write the corresponding sections of the input file.

The function header-lines writes the first two lines to the file. The only parts of these lines that change are the numbers of inequalities and variables.

The function objective writes the coefficients of the objective function stored in the variable cost-vector. The required format is eight variable index-coefficient pairs per line, and this section of the file is terminated by a line consisting of ten 9's.

The function constant-vector keeps count of the inequalities and calls the function constant-vector-1 to write the number of the inequality, its relation (coded as 0 for $=$, 1 for $\leq$, and 2 for $\geq$), and the constant term for each inequality. These are formatted with eight inequalities per line. This section of the file is also terminated by a line consisting of ten 9's.

The function coefficient-matrix writes the lines giving the coefficients of the variables in each inequality. It calls the function parse-string to read an element of the list lhs-list and extract the numbers of the variables appearing with their coefficients. It then writes the number of the inequality, the variable numbers, and the coefficients to the input file, with at most seven variable-coefficient pairs per line. This section of the program is also terminated by a line consisting of ten 9's.

The input file format for the IMINOS prototype is identical to the format used by the MINOS system for solving non-integer inequality systems [14]. This is so because IMINOS implements a branch-and-bound strategy that uses the MINOS system to solve LP-relaxed linear problems. See [7] for a detailed description of the IMINOS prototype.

The format of our input files to IMINOS consists of four sections. The first section contains general information about an ILP problem, such as the numbers of rows and columns, the maximum number of branch-and-bound steps to be performed, and an upper bound on the variable values. In addition, in this section parameters are defined that control the type and the amount of output to be produced by IMINOS during the branch-

and-bound activity. At present these parameters are set so as to minimize the amount of output produced at each branch-and-bound iteration. The second section defines a name for each of the rows appearing in an ILP system. In addition, this section defines the type of relation appearing in each row (i.e. whether $\leq$, $=$ or $\geq$). The third section defines, for each column in the ILP system, a name for the column and the non-zero coefficients appearing in the column. Thus, this section defines the coeffient matrix and cost vector of the ILP system in column-major order. Finally, the fourth section defines the non-zero coeffiecients in the cost vector. See [14] for a more detailed description of the MINOS input file format. The input files for the IMINOS prototype are generated by invoking the function IMINOS-input. This function first generates a cost vector for the current problem by invoking function generate-cost-vector. The resulting vector is stored in the variable cost-vector. In this case the coefficients appearing in the cost vector are non-negative, since IMINOS minimizes the objective functions and we are in fact interested in finding minimal solutions. Next, IMINOS-input opens the file iminos.txt in the current directory and invokes the following functions to generate each of the four portions of an IMINOS input file: iminos-header, iminos-rows, iminos-columns and iminos-rhs. Each of these functions prints a portion of the IMINOS input in the file iminos.txt.

Function iminos-header first prints a line containing the keyword BEGIN and the problem name stored in the variable *problem-name*. Next, this function prints a sequence of lines. Each line in this sequence consists of a keyword denoting a parameter and a value for the parameter. Last, iminos-header prints a line containing the keyword END and the problem name.

Function iminos-rows first prints a line containing the keyword NAME and the problem name defined in the BEGIN line. Next, this function prints a line containing the keyword ROWS. A sequence of lines is printed next, where each line has information regarding a row in the ILP system generated. In particular, each line contains a keyword denoting the type of relation appearing in the row (i.e. L for $\leq$, G for $\geq$ and E for $=$), and a name for the row. Row names are in the form R-x, where x is an increasing sequence of integer numbers, starting from one. Last, iminos-rows prints a line containing the words N and COST, to define the cost vector.

Function iminos-columns first invokes the function column-indices that uses the information in lhs-list to store the coeffiecient matrix of the ILP system in column-major format. Next, iminos-columns prints a line containing the keyword COLUMNS. Subsequently this function prints a sequence of lines, where each line defines information relative to an ILP variable. The first entry in each line is a variable name. Similar to row names, variable (i.e. column) names are generated in the form X-x, where x is an integer number. The variable name is followed by one or two pairs, each pair consisting of a row name and a real coefficient. In each pair the row name, along with the variable name appearing in the same line, identifies uniquely the position of the corresponding coefficient in the coefficient matrix or the cost vector of the ILP problem. Since the MINOS input file format allows up to two pairs for each line in this section, whenever possible we put two pairs in each line to make the resulting input file more readable. Only the non-zero coefficients need to

be specified.

Function iminos-rhs first prints a line containing the keyword RHS. Next, this function prints a sequence of lines where each line defines one or two coefficients appearing in the right-hand sides of the generated inequalities. Each of these lines contains the keyword RHS, followed by one or two pairs each consisting of a row name and a coefficient. In each pair the row name identifies a row in the ILP system and the coefficient denotes the right hand side value for the row.

Last, the string END DATA is printed by the function iminos-end-data.

3.7 Report Generation

The report generated by the prototype consists of three parts. The first part merely lists the inequalities generated. In the second part information concerning each variable appearing in the inequality system is displayed. Given an ILP variable, the environment of the variable and the subexpression associated with the variable are given. In addition, since the subexpression associated with the variable must be an alternation, a Kleene star operand, a transition, or an accept, this information is also displayed. Last, if a solution to the inequality system was found in an output file written by IMINOS or the Land and Powell package, the value of the variable in the solution is listed as well. The third part of the report gives, for each event symbol appearing in the task expressions input to the inequality generator, a list of the variables whose bag contains the event symbol.

The report generator can be executed by invoking the function generate-report. This function takes as input the output device to which the report is to be written (e.g. a text file or an interactive user terminal). Additional input specifies both whether an output file containing a solution to the inequality system is available and what the name of such a file is. Function generate-report uses functions print-inequalities, print-ILP-variable-expressions and print-symbol-table, to generate each part of the report.

To generate the report, the functions contained in this module use the information stored by the inequality generator in the LISP lists lhs-list, rhs-list, rel-list, symbol-table, ILP-variable-bags, ILP-var-types and ILP-variable-expressions. If the report generator is executed immediately after the inequality generator during a LISP session, these structures are already set appropriately; otherwise, these structures can be set up by first invoking the function analyze with arguments specifying that no input file for an ILP system be written (e.g. with *default-ILP-system* set to the empty string). The global variable ILP-values-defined? described below is used to determine whether a solution to a system of inequalities has been read into LISP since the last execution of the inequality generator.

In addition, generate-report can be used to parse a file output by an ILP system to include data about a solution in the report. In this case, depending on the value of variable *default-ILP-system*, generate-report invokes either parse-iminos-output or find-solution to parse an IMINOS output .sol file, or a Land and Powell output .err file, respectively. If no ILP system is defined (i.e. *default-ILP-system* is the empty string), an error is signaled. If a solution is parsed successfully from an .err or .sol file, generate-report sets the logical

variable ILP-values-defined? to be used during the subsequent report generation activity. This variable is cleared by function analyze, whenever a new inequality system is generated.

Function find-solution reads iteratively each line in the .err file. When a line is found that contains the header of a solution to the inequality system, the ILP variable values contained in the subsequent lines are read into the array ILP-values. The value of the LISP variable n-count is used to determine the number of values to be read each time a solution is found in the .err file.

Function parse-iminos-output performs a loop reading LISP tokens from an IMINOS .sol file. Each line in the file stores an ILP variable name and a value for the variable. Additional lines in the file are disregarded. Similar to function find-solution, the values of the ILP variables are stored in the vector ILP-values.

In order for find-solution and parse-iminos-output to work properly, the LISP system must contain information relative to the same inequality system as the one used by the ILP system of choice to produce the .err or .sol file under consideration. It is the user's responsibility to ensure that the information contained in this file is consistent with the data structures in the LISP system. For instance, during a typical interactive session a user may first use function analyze to generate the inequality system and write the IMINOS or Land and Powell input file. Next the ILP system of choice can be run outside the LISP environment, resulting in the creation of an .err or .sol file. Last, function generate-report can be invoked, assuming that the LISP environment that generated the inequality system is still available. If such an environment is not available, the function analyze must be invoked before generate-report to recapture the same context as the one used to generate the input file to the chosen ILP system.

3.8 Translating REDFA solution information into DFA solution information

The behavior generator tool uses tokenized DFAs to search the space of possible program behaviors, but inequality generation is usually done from untokenized REDFAs, which produce the smallest inequality systems [10]. (In a *tokenized* DFA, all event symbols have been converted from strings to LISP symbols.) The detailed solution information (i.e., how many times each arc was traversed) for the DFA is not directly available if the inequalities were generated from the REDFA; however, this information can be constructed from the DFA, the REDFA, and ILP-symbol-var-table. The behavior generator needs a table like ILP-transition-var-table that contains task-state-letter triples for the tokenized DFA versions of the tasks. Call this table the arc variable table; it will be owned by the behavior generator tool but built in this module.

Figure 1 illustrates the translation. The five state DFA was converted to the REDFA shown above it and inequalities were generated from the REDFA. The transition variable for the arc is x_1. Analyzing the expression produces two more variables, x_2 and x_3, for the operands of the OR. The arc variable table must associate each arc of the DFA with one of these variables, as shown in the figure. The contents of the relevant tables for

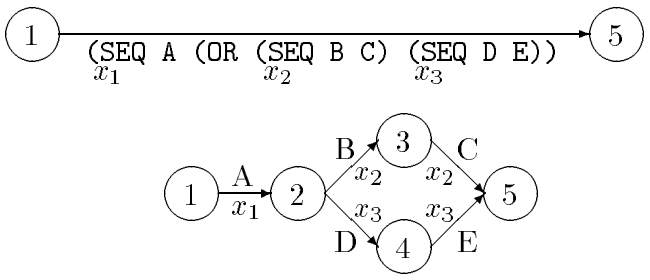

Figure 1: DFA and REDFA for task T

```
ILP-transition-var-table
(T 1 (SEQ A (OR (SEQ B C) (SEQ D E)))) --> X1

ILP-symbol-var-table          arc variable table
A --> X1                      (T 1 A) --> X1
B --> X2                      (T 2 B) --> X2
C --> X2                      (T 2 C) --> X2
D --> X3                      (T 3 D) --> X3
E --> X3                      (T 4 E) --> X3
```

Figure 2: Hash tables for translation example of task T (`KEY --> VALUE`)

this example are given in Figure 2 (the event symbols in the REDFA, ILP-transition-var-table and ILP-symbol-var-table are untokenized; the event symbols in the DFA and the arc variable table are tokenized).

The translation is accomplished by traversing the expression labeling the REDFA arc and tracing out the corresponding path taken in the DFA (note that the uncollapsed states in the REDFA have the same name as they had in the DFA). The recursive traversal function is called traverse. When an event symbol is traversed, the current state of the DFA is changed and an arc variable table entry is created for the DFA arc crossed. The entry associates the DFA arc with the variable in ILP-symbol-var-table stored for this occurrence of the event symbol. To traverse a sequence, each subexpression is traversed in order, starting each traversal at the DFA state where the previous traversal left off. To traverse a disjunction, each subexpression is traversed starting from the current DFA state.

The function transfer-arc-info takes a task name, an untokenized REDFA, a tokenized DFA, and an arc variable table, and creates entries in the arc variable table for each arc in the DFA using the information in ILP-symbol-var-table.

3.9 Low-Level Utilities

This module defines LISP objects that are used by the other modules to access and modify global data structures, such as symbol-table and ILP-variable-bags. In addition, this module defines the function re-normalize that eliminates redundant subexpressions from a regular expression.

The symbol-table structure is a hash table that associates the event symbols appearing in the input task expessions to the ILP variables whose bags contain the corresponding event symbol. Thus, a LISP string denoting an event symbol is used as the key in symbol-table to retrieve a list of LISP symbols denoting ILP variables. The symbol-table structure provides information that is used to compute the total number of occurrences of an event symbol in a derived event sequence. In fact, given an event sequence and the corresponding assignment of values to the ILP variables, the total number of occurrences of an event symbol in the event sequence is the sum of the values of the ILP variables associated with the event symbol in symbol-table (see section 3.3 above).

The function build-symbol-table uses the structure ILP-variable-bags to set up symbol-table. This function is invoked by analyze immediately after the inequalities reflecting the regular expression semantics of the input task expressions have been generated and before the inequalities for the CEDL constraints are to be generated. To maintain case insensitivity concerning event symbols, LISP strings stored as keys in symbol-table are lower case.

The function symbol-table takes as input an event symbol and returns the list of ILP variables associated with the symbol in the symbol-table structure. The function table-update takes as input an event symbol and an ILP variable and modifies the symbol-table structure so that the entry for the input event symbol will contain the ILP variable. A new entry for the event symbol is created, if none existed before the update. Finally, the function clear-symbol-table clears the contents of the symbol-table structure.

The ILP-variable-bags structure, described in section 3.1 above, is accessed and modified by the functions ILP-variable-bag and bag-update, respectively. The function ILP-variable-bag takes as input a symbol denoting an ILP variable and returns a lists of strings denoting the event symbols contained in the bag of the ILP variable. Function bag-update takes as input a LISP symbol denoting an ILP variable and a LISP string denoting an event symbol and updates the ILP-variable-bags structure, so that it will contain the desired event symbol in the entry for the input ILP variable. ILP-variable-bag is initialized to NIL by analyze before inequalities for an input set of task expressions are generated.

Function re-normalize takes as input a regular expression and returns an equivalent regular expression from which certain redundancies have been removed. The input and the return expressions are equivalent in that the same language (i.e. set of event sequences) is derived from these two expressions. In particular, this operation removes concatenation and alternation operations with no or one operand. In addition a subexpression whose operand is the same as the innermost enclosing subexpression can be removed (e.g. one of two nested Kleene stars). Function re-normalize is invoked by function analyze to eliminate

possible redundancies in the input task expressions before the inequality generator is executed. Function re-normalize first performs some type checking of the input regular expression, then invokes function re-normalize-1 to carry out a recursive analysis of the expression. The input task expressions are copied and not modified by re-normalize and re-normalize-1.

4 Documentation of LISP objects

ADD-CONSTRAINTS [Function]

This function implements an interactive capability that allows a user of the prototype to enter inequalities by hand. It consists of a main loop with the options of (1) adding an ILP constraint, (2) terminating the interactive session, (3) generating an IMINOS input file, and (4) generating a Land and Powell input file. If the first option is chosen, a new option menu is presented that allows an interactive user to specify the left hand side, right hand side, and relation type of a new constraint.

ANALYZE
&OPTIONAL *re-list dfa-list LP-input manual-add-const objective-type* [Function]

This is the top-level function of the inequality generation module. Optional argument *re-list*, which defaults to the value of the global variable *task-expressions*, is a list of the input task expressions to be analyzed. Optional argument *dfa-list*, which defaults to the value of the global variable *task-dfas*, is a list of the task DFAs to be analyzed. The objects in both of these lists are labeled: the elements of each list are of the form `(<label> <object>)`. The labels are the names of the tasks. Optional argument *LP-input* must be a LISP string used to denote whether an input file for an ILP package must be generated or not. If an empty string is supplied no input file is generated. If the string "IMINOS" is supplied, then the input file *iminos.txt* is generated for the IMINOS prototype. If the string "Land" is supplied, then the input file *input.txt* is generated for the Land and Powell package. This argument is case insensitive and defaults to the value of the global variable *default-ILP-system*. A non-NIL value to the fourth argument, *manual-add-const*, indicates that constraints must be added interactively after the inequality system for the input task expressions has been generated. This argument defaults to NIL. The fifth optional argument is a LISP string denoting the type of the objective function to be generated. This argument is case insensitive and defaults to the value of the global variable *default-cost-vector*. Possible values are "ILP-variable-unit", to indicate that each ILP variable must have a unit coefficient, and "event-symbol-unit", to indicate that each event symbol appearing in the input task expressions must have unit weight.

CONSTRAINTS [Variable]

This global variable is used to store constraints input to the inequality generator. This

variable is initialized to NIL, set by macro defconstraint and reset to by function init-CE.

DEFAULT-COST-VECTOR [Variable]

This global variable is a LISP string denoting the type of cost vector to be used as the default for inequality systems to be generated. Possible values for this variable are "ILP-variable-unit", which is also the initial value of this variable, and "event-symbol-unit". The function toggle-cost-vector should be used to modify this variable. This variable is used in the body of function generate-cost-vector to determine the type of cost vector to be generated. When this variable is "ILP-variable-unit", the absolute value of each of the coefficients in the cost vector is one. When this variable is "event-symbol-unit", the absolute value of the coefficient for a variable x_i is the number of events symbols contained in the bag of variable x_i. In either case, the sign of the coefficients will depend on whether the Land and Powell package or IMINOS is used to solve the resulting ILP system (i.e. negative for Land and Powell and positive for IMINOS).

DEFAULT-ILP-SYSTEM [Variable]

This global variable is a LISP string denoting the default ILP package to process the ILP systems generated by this prototype. Thus, this variable can be used to control the default behavior of the prototype with respect to the generation of an input file for the ILP package of choice. Possible values of this variable are the empty string "", denoting no ILP package (and therefore no input file), the string "IMINOS", denoting the IMINOS package, and the string "Land", denoting the Land and Powell package. The initial value of this variable is the empty string and the function toggle-ILP-system should be used to modify this variable.

DEFCONSTRAINT *name* &BODY *constraint-expression* [Macro]

This macro is used to input a constraint into the inequality generator. The first parameter, *name*, is a name for the constraint. The name is either a LISP symbol, the constraint name, or a list whose first element is the constraint name and whose remaining elements are the names of the tasks constrained by this constraint. Using the latter form of constraint name speeds up subsequent processing in the constraint eliminator and behavior generator. The second parameter is a regular expression in standard prefix format [8] that defines the input constraint.

DEFTASK *name* &BODY *task-expression* [Macro]

This macro is used to input a task expression into the inequality generator. The first parameter, *name*, is a name for the task. The second parameter is a regular expression in

standard prefix format [8] that defines the input task expression.

DEFTASKDFA *name* &BODY *encoded-dfa* [Macro]

This macro is used to input a task DFA into the inequality generator. The first parameter, *name*, is a name for the task. The second parameter is a task DFA enocoded as a list [9].

GENERATE-REPORT &OPTIONAL *file-name LP-err-file* [Function]

This function generates a booklet describing a system of inequalities. The optional argument *file-name*, which defaults to the stream bound to *TERMINAL-IO*, indicates the device to which the booklet is output. If *file-name* is a string, it is meant to denote a file name. In this case, an output stream is opened to the named file, the booklet output, and the stream closed. If *file-name* is a stream, the booklet is output to the stream and the stream is not closed. The optional argument *LP-err-file* can be used to specify an input stream or a string denoting the name of a file output by an ILP system. This could either be an .err file produced by the Land and Powell package or a .sol file produced by IMINOS. The default value of this parameter is NIL, in which case no action is taken. However, if a valid input stream or file name is specified, the file is searched in an effort to find the optimal solution to an inequality system. It is the user's responsibility to ensure that the LISP context at the time generate-report is invoked is the same as the context used to produce the input file for the integer programming system of choice. In particular, the structures lhs-list, rel-list, rhs-list, symbol-table, ILP-variable-expressions and n-count must have the same values.

The booklet consists of three parts:

1. The inequalities forming the ILP system
2. The CEDL expression associated with each ILP variable and the environment of such a variable
3. For each event symbol appearing in the alphabet of the CEDL system, the ILP variables denoting occurrences of the event symbol

T is returned.

GENERATE-SMALL-REPORT &OPTIONAL *file-name LP-err-file* [Function]

This function is similar to function generate-report, the main difference being that only the third part of the booklet is put in the argument *file-name*. The *file-name* argument, which defaults to the stream bound to *TERMINAL-IO*, indicates the device to which the report is output. If *file-name* is a string, it is meant to denote a file name. An output stream is opened to the named file, the report output, and the stream closed. If *file-name* is a stream, the report is output to the stream and the stream is not closed. The report describes, for each event symbol appearing in the alphabet of the CEDL system, the ILP

variables denoting occurrences of the event symbol. T is returned.

INIT-CE [Function]

This function initializes global data structures, such as the variable *task-expressions*, before a set of task expressions and constraints are loaded into the LISP environment. In addition this function clears the data structures that are used to store the output of the inequality generator, such as the lists lhs-list, rhs-list, rel-list, ILP-variable-bags and ILP-variable-expressions.

IMINOS-INPUT [Function]

This function opens the file iminos.txt in the current directory and writes an ILP system in the IMINOS format to this file. Consequently this function can only be invoked after the inequality generator has generated an inequality system. T is returned.

PROBLEM-NAME [Variable]

This global variable is bound to a LISP string indicating the name of the ILP system to be generated. This string is used when creating an input file for the IMINOS prototype. The default value of this variable is "ILP-SYSTEM".

RE-NORMALIZE *regular-expression* [Function]

This function takes as input a regular expression in prefix syntax and returns a semantically equivalent regular expression, in which:

1. Subexpressions with an empty operand list are removed
2. SEQUENCE and OR subexpressions with one operand are collapsed
3. Nested SEQUENCE, nested OR and nested STAR subexpressions are collapsed

A fatal error is signaled if the input argument is not a legal regular expression (i.e. a LISP string or list). This operation is non-destructive, that is, the input argument is not modified during the execution of this function.

TASK-EXPRESSIONS [Variable]

This global variable is used to store task expressions input to the inequality generator. This variable is initialized to NIL, set by macro deftask and reset to by function init-CE.

TASK-DFAS [Variable]

This global variable is used to store task DFAs input to the inequality generator. This

variable is initialized to NIL, set by macro deftaskdfa and reset to by function init-CE.

TOGGLE-COST-VECTOR [Function]

This function toggles the value of the global variable *DEFAULT-COST-VECTOR*. If the current value is "Event-symbol-unit", this variable is set to "ILP-variable-unit" and vice versa. The new value of the variable is returned. An error is returned if the variable does not have one of these two values when this function is invoked.

TOGGLE-ILP-SYSTEM &OPTIONAL *nullify* [Function]

This function toggles the value of the global variable *DEFAULT-ILP-SYSTEM*. If the current value is "IMINOS", this variable is set to "Land" and vice versa. The new value of the variable is returned. The optional argument, which defaults to NIL, specifies whether the variable is to be set to the empty string. In fact, if a non-NIL value is supplied, the variable is set to "", regardless of its previous value. If this function is executed when the current value of the variable is "" and the optional argument is NIL, the variable is set to "IMINOS". An error is returned if the variable does not have one of these three values; otherwise the new variable value is returned.

TRANSFER-ARC-INFO *task-name redfa dfa arc-variable-table* [Function]

Translates the solution information about an untokenized REDFA from which inequalities were generated into solution information about a tokenized DFA by creating entries in the arc variable table. Each entry associates an arc in the DFA with a variable. This function uses information stored during inequality generation in the table ILP-symbol-var-table. See section 3.8 for details.

References

[1] S. Avery. A tool for producing constrained expression representations for CEDL designs. Technical Report SDLM 89-2, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, 1989.

[2] G. S. Avrunin. Experiments in constrained expression analysis. Technical Report 87-125, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, November 1987.

[3] G. S. Avrunin. A prototype inequality generator. Technical Report SDLM 88-1, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, 1988.

[4] G. S. Avrunin, L. K. Dillon and J. C. Wileden. Experiments with automated constrained expression analysis of concurrent software systems. In *Proceedings ACM SIGSOFT 3rd Symp. on Software Testing, Analysis and Verification*, pages 124–130, December 1989.

[5] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2):278–292, 1986.

[6] G. S. Avrunin and J. C. Wileden. Describing and analyzing distributed software system designs. *ACM Transactions on Programming Languages and Systems*, 7(3):380–403, July 1985.

[7] J. Burnett and U. Buy. Solving integer programming systems using the IMINOS prototype. Technical Report, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, in preparation.

[8] U. Buy. Documentation on an analysis tool for constrained expressions. Technical Report SDLM 87-1, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, May 1987.

[9] J. Corbett. Automatic elimination of constraints for constrained expressions analysis. Technical Report, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, in preparation.

[10] J. Corbett. On selecting a form for inequality generation in the constrained expression toolset. Technical Report SDLM 90-1, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, May 1990.

[11] L. K. Dillon. Overview of the constrained expression design language. Technical Report TRCS86-21, Department of Computer Science, University of California, Santa Barbara, California, October 1986.

[12] M. Greenberg and S. Avery. A behavior generator. Technical Report SDLM 89-1, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, 1989.

[13] A. H. Land and S. Powell. *Fortran Codes for Mathematical Programming: Linear, Quadratic and Discrete*. John Wiley & Sons, Ltd., London, 1973.

[14] B. A. Murtagh and M. A. Saunders. MINOS 5.1 User's Guide. Technical Report SOL 83-20R, Department of Operations Research, Stanford University, Stanford CA, December 1983.

[15] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., New York, New York, 1988.

[16] J. C. Wileden and G. S. Avrunin. Toward automating analysis support for developers of distributed software. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 350–357. IEEE Computer Society Press, June 1988.