REDLIB
A Library for Integrated BDD-like Diagrams *

Farn Wang
Dept. of Electrical Engineering
Graduate Institute of Electronic Engineering
National Taiwan University
1, Sec. 4, Roosevelt Rd., Taipei, Taiwan 106, ROC;
+886-2-33663602; FAX:+886-2-23671909;
farn@cc.ee.ntu.edu.tw; http://cc.ee.ntu.edu.tw/~farn

RED 7.0 is available at http://cc.ee.ntu.edu.tw/~val

October 14, 2008

Table of contents

1 Introduction ............................................................... 4
2 Technology of REDLIB .................................................. 13
3 Structure of a REDLIB program ....................................... 14
   3.1 Model declaration through API ................................. 14
   3.2 Model declaration through a file ............................... 21
   3.3 Model modification through API ................................. 23
   3.4 Model modification through a file ............................... 24
4 Variable declarations .................................................... 25
5 Expression strings ....................................................... 26
6 Constraint strings ....................................................... 26
7 Model structure declarations .......................................... 32

*The work is partially supported by NSC, Taiwan, ROC under grants NSC 93-2213-E-002-130.
References ................................................................. 102

Appendix B Syntax of RED input file format. ......................... 107
1 Introduction

REDLIB is constructed out of the TCTL model-checker RED. The basic motivation is for BDD-like diagrams that allow the integrated representation and manipulation of state-space characterizations of both discrete and dense variables. Conceptually, a BDD-like diagram is a directed acyclic graph (DAG) whose internal nodes are labeled with variables and whose external nodes are with true or false. The BDD-like diagrams used in REDLIB were originally called RED (Region-Encoding Diagrams) which are zero-suppressed. At this moment, REDLIB supports the following four types of variables.

- **Boolean variables:** Each variable in this category may have value true or false.
- **Discrete variables:** Each variable in this category is declared with a value lower-bound and an upper-bound. The values of such a variable are within the lower-bound and the upper-bound.
- **Clock-restriction variables:** Each variable in this category is of the form $x - y$ where $x$ are $y$ are declared dense-value clock variables. The values of such a variable are upper-bounds like either $< c$ or $\leq c$, where $c$ is no greater than the biggest timing constants used in a model description and specification. The increment rates of all clocks are uniform. Such variables are specifically used for the verification of timed automata (TA) [2]. Note that the users of REDLIB do not declare the clock-restriction variables. They are automatically constructed from the model description, specification, and state-space manipulation.
- **Hybrid-restriction variables:** Each variable in this category is of the form $a_1x_1 + a_2x_2 + \ldots + a_nx_n$, where $a_1, a_2, \ldots, a_n \in \mathbb{Z}$ and $x_1, x_2, \ldots, x_n$ are dense variables. The dense variables can increment or decrement their values at different rates. Such variables are specifically used for the verification of linear hybrid automata (LHA) [3]. Note that the users of REDLIB do not declare the clock-restriction variables. They are automatically constructed from the model description, specification, and state-space manipulation.

One way that REDLIB gain its performance is through an integration of BDD-like diagrams for BDD, MDD, CRD, and HRD. Such integration allows structure-sharing among constraints for discrete and dense spaces. REDLIB does not allow a BDD-like diagram that uses both clock-restriction variables and hybrid-restriction variables. One without hybrid-restriction variables is called a CRD+MDD while one without clock-restriction variables is called a HRD+MDD.

**Example 1:** The CRD+MDD for $(0 - x_1 < -3 \land (b \land x_1 - x_3 < -1) \lor (\neg b \land (4 \leq d \leq 6 \lor 1 \leq d \leq 2))) \lor (0 - x_1 \leq -5 \land (x_1 - x_3 < -1 \lor (x_1 \leq 5 \land (4 \leq d \leq 6 \lor 1 \leq d \leq 2))))$ is in figure 1(a). Here $x_1$ and $x_3$ are declared clock variables, $b$ is a Boolean variable, and $d$ is a discrete variable. Note that we allow upper-bound $< \infty$ which is always satisfied. Variables like
\[ x - y \text{ in the diagrams are either specified by the users or constructed by programs.} \]

In figure 1(b), we have an HRD+MDD for

\[ (-x_1 + 3x_2 < \frac{3}{2} \land ((b \land -x_2 < -\frac{3}{4}) \lor ((-b) \land (4 \leq d \leq 6 \lor 1 \leq d \leq 2)))) \lor (-x_1 + 3x_2 \leq -\frac{5}{8} \land (-x_2 < -\frac{1}{4} \lor (2x_1 + 3x_2 - x_3 \leq 5 \land (4 \leq d \leq 6 \lor 1 \leq d \leq 2)))) \]

Here \( x_1, x_2, x_3 \) are declared dense variables. Variables like \(-x_1 + 3x_2, -x_2, \) and \(2x_1 + 3x_2 - x_3\) in the diagrams are either specified by the users or constructed by programs.

**REDLIB** not only allows us to manipulate BDD-like diagrams. It also allows us to declare behavior structures represented as parameterized communicating automata (PCA). A communicating automaton consists of a set of process automata that communicate through CSP-style communication channels. A communicating automaton is parameterized since we let many process automata share the same automaton template and identify each process automaton with a process index. **REDLIB** also allows us to declare local variables of each process automaton, and reference the local variables through their process indices.

There are two types of dense variables that we may use with **REDLIB**. The first type is for clock variables, in timed automata [2], whose increment rates are always 1. The second type is for dense variables, in linear-hybrid automata [3], whose rates of changes can be any real numbers specified in an interval with rational bounds. **REDLIB** has the following restrictions on the model construction and verification tasks.

- If we want to do full TCTL model-checking, then all dense variables must be clock variables.
  In this case, the models are also called CTA (Communicating Timed Automata).
- If we want to reason the constraints on unknown dense parameters, then we must use linear-hybrid automata as our models. In this case, we may declare dense variables whose rates of changes may not be uniform, we may specify any constraints on any linear expressions.

---

Figure 1: A CRD+MDD diagram
of dense variables, and we may perform parametric analysis of the dense parameters. In this case, we call the model \textit{CLHA (Communicating Linear-Hybrid Automata)}.

**Example 2**: We may declare the CSMA/CD bus arbitration protocol in figure 2 as a CTA of three process automata. The six ovals represent the control locations, wait, transm, retry, idle, active, and collision. The graph in figure 2(a) is a template of the process automata for all the sender processes. In the template, we use a local clock \( x \). For process \( p \), we may reference its local clock by name \( x[p] \) in a specification formula. The graph in figure 2(b) is a template of the process automata for the bus process. The CTA consists of two sender processes and one bus process.

Inside each oval, we may write a formula for the invariance condition of that location. We use arcs to represent the transitions between locations. By each arc, we label the corresponding triggering condition, synchronization events in CSP-style, and the actions.

To use the behavior structure and local variables, the users must first declare the range of process indices. The process indices are from 1 to a positive integer constant given by a user. The initial formula and the specification formulas are to be described separately. The initial and specification formulas can be input as strings to \textsc{REDLIB}. For details, please read section 3.

**Example 3**: We may write down the initial condition of the system in figure 2 as follows.

\[
\text{idle[1]} \land x[1] = 0 \land \forall i : i > 1, (\text{wait[i]} \land x[i] = 0)
\]
The formula says that process one starts its execution from location \texttt{idle} with local clock $x[1]$ set to zero. Moreover, all other processes start their execution from location \texttt{wait} also with their local clock set to zero.

After the declaration of the process count, the variables, and the model structure, we can use \texttt{REDLIB} to manipulate the state-spaces and carry out our verification tasks. In the following, we briefly discuss the capabilities of \texttt{REDLIB}.

### Basic diagram manipulations

Given two BDD-like diagrams $D_1$ and $D_2$, we can use \texttt{REDLIB} to do the following basic operations.

- A logic atom. Please check page 80.
- A disjunction of $D_1$ and $D_2$. Please check page 49.
- A conjunction of $D_1$ and $D_2$. Please check page 49.
- The complement of $D_1$. Please check page 49.
- The restriction of $D_1$ with an atom. Please check pages ??, ??, and ??.
- Fourier-Motzkin elimination of a variable $v$ from $D_1$. Conceptually, the result is $\exists v(D_1)$. Please check page ??.

### Precondition & postcondition calculation

\texttt{REDLIB} supports precondition & postcondition construction at the following granularity.

- A single discrete action. Please check pages ?? and ??.
- A single discrete transition rule. Please check pages 54, 55, 62, and ??.
- A synchronous global transition composed a set of process transitions from synchronizing processes. This could be efficient when each synchronization does not engender the enumeration of many processes’ triggerible transitions. Please check pages 55, 56, ??, ??, 59, and 60.
- A set of synchronous global transitions represented as a BDD-like diagrams. This could be efficient when the synchronization among processes engenders the enumeration of many processes’ triggerible transitions. Please check pages 57, 57, 58, and 58.
- Time-progress. Please check page 54.

### Reachable state-spaces

\texttt{REDLIB} supports the calculation of various reachabilities.

- The backward and forward reachabilities. Please check pages ?? and ??.
• The backward and forward reachabilities with untimed abstraction. Please check pages ?? and ??.
  Note that the untimed abstraction is carried out in the calculation in the intermediate steps instead of at the final reachability images.

• The backward and forward reachabilities with magnitude abstraction. Please check pages ?? and ??.
  Note that the magnitude abstraction is carried out in the calculation in the intermediate steps instead of at the final reachability images.

• The backward and forward reachabilities with diagonal abstraction. Please check pages ?? and ??.
  Note that the diagonal abstraction is carried out in the calculation in the intermediate steps instead of at the final reachability images.

Normalization

Diagrams with only Boolean variables and discrete variables are automatically canonical and minimal. But in general, CRD+MDDs and HRD+MDDs are not canonical. REDLIB also supports the following procedures for the normalization.

• **Tight form**: A diagram is in tight form (or all-pair-shortest-form) if every constraints that can be transitively deduced for a convex polyhedron are also included. For CRD+MDDs, REDLIB supports the calculation of tight forms. For HRD+MDDs, strict tight form may not be computable since the set of inequalities for each convex polyhedron may be infinite. REDLIB does supports the procedure that adds in inequalities, that can be constructed from two inequalities in one iteration, to a convex polyhedron. REDLIB also supports procedures that get rid of some subsumed inequalities from a convex polyhedron. Please check page 50.

• **Magnitude reduced form**: A magnitude inequality is either like \( x \leq c \) or \( -x \leq c \). This is very much similar to the tight form except that all inequalities that are subsumed by two magnitude inequalities are eliminated. Please check page 50.

State-space abstraction

Given a model structure and an initial condition, REDLIB has several procedures to automatically construct abstractions of the forward reachable state-space. Following are the possibilities.

• **The untimed reachable state-space**: This is the forward reachability obtained by ignoring all timing variables of the automata. Please check page ??.

• **Magnitude reachable state-space**: This is the forward reachability obtained by ignoring all timing inequalities containing more than one dense or clock variables. Please check page ??.
• **Discrete-time reachable state-space:** This is the forward reachability obtained by only recording the integer values of all clocks. Please check page ??.

• **Diagonal reachable state-space:** This is a forward reachability obtained by keeping only some timing constraints of the form $x - y \sim c$. Note that constraints with only one clock or dense variable are also omitted. Please check page ??.

Such abstractions can be used to check whether the verification tasks can be carried out with the abstract state-spaces characterizations.

**Reduction techniques**

REDLIB supports several reduction techniques. If we do state-space manipulation in fine granularity piece by piece, then they can be invoked as procedures. If we do it in coarse granularity, like one iterations of a fixpoint, then we can use options to choose whether to invoke the reductions or not.

• **Inactive variable elimination:** A variable is inactive if it will not be read unless it is written to again. Inactive variables’ values do not affect the computation and thus are omitted from state-space representation. This reduction is invoked automatically in coarse granularity. Please check page 53.

• **Symmetry reduction:** REDLIB supports an approximation of process-oriented symmetry reduction. It can be either invoked with procedures or with flags. Please check page 53. For more technical details, please check [8, 17].

• **Early decision of greatest fixpoint evaluation (EDGF):** While doing greatest fixpoint evaluation, usually we are in the context of evaluating a formula like $p \rightarrow \forall \diamond q$ whose negation is $p \land \exists \Box \neg q$. The EDGF strategy is to return $false$ when we find the conjunction of $p$ and the image of the greatest fixpoint of $\exists \Box \neg q$ is $false$ in a fixpoint iteration. This strategy holds since the greatest fixpoint images of $\exists \Box \neg q$ shrinks iteration by iteration. EDGF is very effective and does not cost much. Thus it is always invoked automatically in the coarse granularity. For technical details, please check [16].

• **Pruning strategy based on parameter space construction (PSPSC):** This is a strategy used to speed up the parametric analysis of linear hybrid systems. It is always invoked automatically in the coarse granularity. Please check [14] for its technical explanation.

**Verification tasks**

At this moment, REDLIB supports the following verification tasks.
Safety analysis

Users can write a risk condition or a safety condition. The safety condition is translated to its negation for a risk analysis.

Example 4: We may write down the risk condition of the system in figure 2 as follows.

\[
\text{risk transm}[2] \land \text{transm}[3] \land (x[2] \geq 52 \lor x[3] \geq 52);
\]

The formula says that both processes 2 and 3 are in location transm while one of their local clock \( x \) reads no less than 52 time units.

REDLIB uses an on-the-fly approach to construct a state-space representation. If the risk condition is satisfiable, REDLIB can also construct a counter-example. Please check page ??.

If the model is a parameterized communicating linear-hybrid automata, REDLIB constructs a constraint for the reachability of the risk condition. Please check page ??.

TCTL model-checking of CTA

REDLIB supports more than full TCTL model-checking with. It also supports strong and weak fairness assumptions of CTA. There are options supported by REDLIB for model-checking with or without Zeno-ness assumption. Please check page ??.

Example 5: For the system in figure 2, we may want to require that if process 2 is in location transm with its local clock reads no less than 52 time units, process 2 will inevitably enter location wait again. In CMU's notations, this can be written as follows.

\[
AG((\text{transm}[2] \land x[2] \geq 52) \rightarrow AF\text{wait}[2])
\]

In Pnueli's notations, this can be as follows.

\[
\forall \Box((\text{transm}[2] \land x[2] \geq 52) \rightarrow \forall \Diamond\text{wait}[2])
\]

In REDLIB, we just input the following string through our API.

\[
tctl \forall x \forall y \forall t ((\text{transm}[2] \land x[2] \geq 52) \implies \forall t \exists y \text{wait}[2]);
\]

REDLIB supports several options to help enhancing the performance of inevitability checking.

REDLIB also allows the assumptions of strong and weak fairness. Conceptually, a run satisfies a strong fairness assumption of \( \phi \), if for every \( t_1 \in \mathbb{R}^{\geq 0} \), there is a \( t_2 > t_1 \) such that along the run \( \phi \) is true at time \( t_2 \). A run satisfies a weak fairness assumption of \( \phi \), if there exists a \( t_1 \in \mathbb{R}^{\geq 0} \) such that for every \( t_2 \geq t_1 \), \( \phi \) is true at time \( t_2 \). In REDLIB, you may specify that
Figure 3: A process to finish its execution with fairness assumption

for every run that satisfies some strong and weak fairness assumptions, a property is true. The motivation for the fairness assumption is that sometimes some liveness properties can only be proven when you assume that some ‘good’ things have a ‘fair’ share of execution time.

**Example 6**: Suppose that we have a process automaton that needs 10 time units to finish its execution. The operating system may only occasionally let the process execute. Figure 3, we have drawn such an automaton. There is a global discrete variable `step` and a local clock `x`. There is only one process. Initially, the system is in location `wait`. Everytime, the system gets to execute, it increment the value of `step` by one. We may want to prove that if the system has infinitely opportunity to execute, then eventually the value `step` is no less than 10. In REDLIB, this can be written as the following formula.

\[ \text{tctl} \ \forall \text{strong} \{ \text{execute}[1]; \} \ \text{eventually} \ step \geq 10; \]

The predicate in the parentheses after ‘strong’ is a strong fairness assumption. Users may also write down many strong fairness assumptions in the same parentheses, each terminated with a semicolon.

**Simulation-checking of CTAs**

Based on a new formulation of the simulation between two timed automata, REDLIB is now able to support the simulation-checking of CTAs. A *simulation* between a CTA $A_1$ and another CTA $A_2$ is a binary relation between the states of $A_1$ and $A_2$. Conceptually, a binary relation $B$ is a simulation from $A_1$ to $A_2$ if for every $(\nu_1, \nu_2) \in B$, in case $A_1$ can do a transition with some synchronization events at time $t$ from $\nu_1$, then $A_2$ can also match the transition with the same synchronization events at time $t$ from $\nu_2$. $A_1$ is *simulated by* (or *implements*) $A_2$ if there exists a simulation $B$ from $A_1$ to $A_2$ such that for every initial state $\nu_1$ of $A_1$, there is an initial state $\nu_2$ of $A_2$ and $(\nu_1, \nu_2) \in B$. In a rigorous way, this is also called a *stuttering* (or *branching*) *simulation*. 

11
Since in REDLIB, systems are described as parameterized communicating automata, we create the concept of model processes, specification processes, and environment processes. That is, given a system of \( m \) processes, the users invoke the simulation-checking capability by telling REDLIB which processes are for the model and which are for the specification. The remaining processes are for the environment. For example, for the system in figure 2, we may have 1 bus and 3 senders. Then we may want to check if the CTA of processes 1 (the bus), 2, and 4 is simulated by the CTA of processes 1, 3, and 4. Intuitively, this could be interpreted as to check whether process 2 is simulated by process 3 with respect to the environment of processes 1 and 4. In REDLIB, this can be written as the following specification.

\[
\text{implementation 2; 3;}
\]

Such a design could significantly reduce the complexity in representing the bisimulation. Please check page ??.

**Miscellaneous**

There are the following procedures that can also be used to support users’ verification tasks with REDLIB.

**Print-out services**

We can use REDLIB to print out a diagram in several formats. We can print it out as a formula with parentheses and Boolean operators for users’ convenience. This could be easy to read when the formula is not too complicate. We can also print out a diagram as a directed graph. In fact, we print it out as a tree with shared structures printed out only at the first time. We can also print out a diagram as a sequence of zones. Each zone is printed out as a sequence of discrete constraints or dense inequalities. Please check subsection 12.9 in page 90.

**Sizes and memory**

We can also use REDLIB to return the size (number of nodes and arcs) or the memory of a diagram. Please check subsection 12.7 in page 89.

We can also use REDLIB to tell us all the memory used by all the diagrams and the other supporting data-structures. Please check subsection 12.8 in page 90.

**Queries to the declarations and model structures**

We can also use REDLIB to check the process count, the declared variables, the declared model structures, the initial condition, and the specification formula. Please check subsection ?? in
Also we may query for the invariance condition derived from the model declaration. This invariance is actually the conjunction of the invariance conditions of all the processes. The invariance condition of a process is the disjunction of the invariance conditions of all locations that can be reached by the process in its automaton graph. Please check page 43. In a sense, the is the starting place for the calculation of all reachabilities.

Garbage collection

At this moment, REDLIB does not do automatic garbage collection. The users can invoke a garbage-collector in REDLIB to reclaim all the diagram structures that are not referenced. It is up to the users’ discretion to determine when to do this. The users can check the total memory consumptions of REDLIB and determine whether to collect the garbage or not. REDLIB also supplies a stack. All diagrams saved in this stack will not be garbage-collected. Please check page 81.

Option setting

There are also many options that we can choose to tune the performance of REDLIB. We can also check the values of those flags of REDLIB. Please check page ??.

2 Technology of REDLIB

REDLIB supports BDD-like diagrams with many sorts of variable types, including Booleans, discrete with finite ranges, clocks, and dense variables. Precisely, REDLIB is a package for shared BDD-like diagrams with zero suppressed. That is, all diagrams calculated out of the REDLIB packages share common structures. This could add some overhead in diagram manipulations. But it could also save memory consumption in representations and save computation time in identity checking.

Unlike most BDD packages that use hash tables to check the structure sharing in diagrams, REDLIB uses 2-3 trees. The 2-3 tree management is through non-recursive procedures and provides a stable performance.

While analyzing a communicating automaton, REDLIB does not support the construction of the product automaton first. REDLIB constructs fixpoint images in an on-the-fly style. However, REDLIB supports the construction of several abstraction of the reachability images, forward and backward. Please check page ??, ??, ??, and ?? . Such abstract reachability images can be used to effectively constrain the exploration space in an on-the-fly reachability analysis.
Table 1: A template for using REDLIB.

For example, in the standard backward safety analysis procedures, REDLIB first calculate the an abstraction of the forward reachability. Then while calculating the on-the-fly backward reachability, REDLIB uses the abstraction to constrain the backward exploration. Depending on the characeristics of the verification tasks, different abstractions may enable us to finish the verification tasks in the abstract state-spaces.

3 Structure of a REDLIB program

In the following, we first show two templates for using REDLIB. Then we show examples.

There are two ways that we can incorporate REDLIB in users’ applications. The first is to make declarations through REDLIB API (Application Program Interface). The second is to read declarations from a file. We explain how to declare a model in these two ways respectively in subsections 3.1 and 3.2.

Moreover, sometimes, we may want to change the transition rules without changing the number of processes and delcaration of variables. We can also We discuss how to do this through API and through files respectively in subsections 3.3 and 3.4.

3.1 Model declaration through API

Now we briefly explain the first approach. REDLIB procedures can be used intermixing with C/C++ statements. REDLIB needs to be used according to the template in table 1. Statement
(A) starts a REDLIB session while statement (G) ends a REDLIB session. At this moment, we do not allow overlapping sessions. Parameter *system_type* is for the system type. Now it can be **RED_SYSTEM_TIMED** for timed automaton verification. It can also be **RED_SYSTEM_LINEAR_HYBRID** for linear-hybrid automaton verification. Parameter *name* is for the name of the session. The name can be used for the creation of many working variables in the session. Parameter *n* is for the number of processes in the system model. The parameters *name* in both statements (A) and (G) must be the same.

Statements (B) starts the declaration section while statement (E) finishes it. Statement (E) also constructs all the tables used for the verification, including the variable table, transition table, process tables, etc. Note that the BDD-like diagrams of **REDLIB** also consists of local variables, clock inequalities, and linear-hybrid inequalities. The ordering among the local variables and inequality variables are cannot be told from the declaration of the discrete, clock, and dense variables. Statement (E) is necessary since clock inequality variables, like \(-x - y\), and linear-hybrid inequality variables, like \(-x + 3y + 2z\), are not declared and must be generated automatically from the clock variables and the dense variables. The indices in the variable actually specify the variable ordering in the BDD-like diagrams constructed with **REDLIB**.

Those code lines at segment (C) are for the declaration of macro constants and variables. Segment (D) is optional and can be used to describe a communicating automaton. Segment (F) is also optional and is used to manipulate BDD-like diagrams.

What statements (A), (C), and (D) do is the following. It creates a file with the session name and writes all the declarations to a file. The file adheres to the format of the input file to TCTL model-checker **RED**. Then statement (D) calls the parser module in **RED** to construct the parsing tree and the tables. The created file can be viewed by the users for the debugging and educational purposes.

In the following, we have an example piece of code for the system in figure 2.

```c
/* (1)*/ #include <stdlib.h>
/* (2)*/ #include <ctype.h>
/* (3)*/ #include <stdio.h>
/* (4)*/ #include <string.h>
/* (5)*/ #include <math.h>
/* (6)*/ #include <float.h>
/* (7)*/ #include "redlib.h"
/* (8)*/ #include "redlib.e"
/* (9)*/ main(int argc, char **argv) {
/* (10)*/ redgram *red_ini;
/* (11)*/ int ini, inv, rch;
```
/*(12)*/ struct reachable_return_type    *rr;

/*(13)*/ red_begin_session(REDSYSTEM_TIMED, "CSMA-CD", 3);

/*(14)*/ if (argc < 2)
/*(15)*/   exit(0);
/*(16)*/ red_switch_output(fopen(argv[1], "w"));

/*(17)*/ // start all the declaration.
/*(18)*/ red_beginDeclaration();

/*(19)*/ // define constants used in RED descriptions.
/*(20)*/ red_comment("Three constants used in the specification.");
/*(21)*/ red_define_const("A", 26);
/*(22)*/ red_define_const("B", 52);
/*(23)*/ red_define_const("LAMBDA", 808);

/*(24)*/ // declare variables
/*(25)*/ red_comment("One local clock.");
/*(26)*/ red_declare_local_variable(REDTYPE_CLOCK, 0, 0, "x");

/*(27)*/ // declare synchronizers, which are also global variables
/*(28)*/ red_comment("4 synchronizers.");
/*(29)*/ red_declare_variable(REDTYPE_SYNCHRONIZER, 0, 0, "begin");
/*(30)*/ red_declare_variable(REDTYPE_SYNCHRONIZER, 0, 0, "end");
/*(31)*/ red_declare_variable(REDTYPE_SYNCHRONIZER, 0, 0, "busy");
/*(32)*/ red_declare_variable(REDTYPE_SYNCHRONIZER, 0, 0, "cd");

/*(33)*/ // start declaring the optional model structure.
/*(34)*/ // modes for the bus.
/*(35)*/ red_begin_mode("idle", "true");
/*(36)*/ red_transition("?begin (true)", "x= 0; goto active;");
/*(37)*/ red_end_mode();
/*(38)*/ red_begin_mode("active", "true");
/*(39)*/ red_transition("?end (true)", "x= 0; goto idle;");
/*(40)*/ red_transition("!busy (x >= A)", ", "");
/*(41)*/ red_transition("?begin (x < A)", "x= 0; goto collision;");
/*(42)*/ red_end_mode();
/*(43)*/ red_begin_mode("collision", "x < A");
/*(44)*/ red_transition("!cd !cd (x < A)", "x= 0; goto idle;");
/*(45)*/ red_end_mode();

/*(46)*/ // modes for the senders.
/*(47)*/ red_comment("3 modes for the senders.");
/*(48)*/ red_begin_mode("wait", "true");
/*(49)*/ red_transition("!begin (true)", "x= 0; goto transm;");
/*(50)*/ red_transition("?cd (true)", "x= 0;");
/*(51)*/ red_transition("?cd (true)", "x= 0; goto retry;");
/*(52)*/ red_transition("?busy (true)", "x= 0; goto retry;");
/*(53)*/ red_end_mode();
/*(54)*/ red_begin_mode("transm", "x <= LAMBDA");
/*(55)*/ red_transition("!end (x==LAMBDA)", "x= 0; goto wait;");
red_transition("?cd (x<B)", "x= 0; goto retry;");
red_begin_mode("retry", "x < B");
red_transition("!begin (x < B)", "x= 0; goto transm;");
red_transition("?busy (true)", "x= 0;" );
red_transition("?cd (x < B)", "x= 0;" );
red_end_mode();

red_translation();

red_end_declaration();

red_print_variables();
red_print_xtions();
red_print_sync_xtions();
red_print_diagram(red_bulk_xtions());

red_ini = red_diagram(
  "idle[1] && x[1]==0 && forall i:i>1, (wait[i] && x[i]==0)"
);
ini = red_push(red_ini);
red_print_line(red_stack(ini));

red_reach_fwd(
  red_stack(ini), red_stack(inv), red_false(),
  RED_TASK_GOAL,
  RED_NO_PARAMETRIC_ANALYSIS,
  RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
  RED_FULL_REACHABILITY,
  -1,
  RED_NO_COUNTER_EXAMPLE,
  RED_NO_TIME_PROGRESS,
  RED_NORM_ZONE_NONE,
  RED_ACTION_APPROX_UNTIMED,
  RED_REDUCTION_INACTIVE,
  RED_OAPPROX_MODL_GAME_UNTIMED |
  RED_OAPPROX_SPEC_GAME_UNTIMED |
  RED_OAPPROX_ENVR_GAME_UNTIMED |
  RED_OAPPROX_GLOBAL_GAME_UNTIMED,
  RED_NO_SYMMETRY,
  RED_NO_PRINT
);
red_set_stack(inv, rr->reachability);

red_reach_fwd(
  red_stack(ini), red_stack(inv), red_false(),
  RED_TASK_GOAL,
  RED_NO_PARAMETRIC_ANALYSIS,
This piece of code uses REDLIB to carry out backward reachability analysis. For convenience of discussion, we have labeled each statement line with a commented statement number to the left. At statements (7) and (8), we include the header files for REDLIB. At statements (10), (11), and (12), we declare a file variable, a BDD-like diagram variable, and three index...
variables to the stack supported by REDLIB.

Statements (13), (18), (64), and (90) respectively correspond to statements (A), (B), (E), and (G) in table 1. Statement (16) sets the output file pointer of the whole REDLIB session, i.e. RED\_OUT, to the file specified as the first command-line argument. Statements (21) to (32) are for variable and constant declarations and correspond to segment (C) in table 1. Statements (35) to (62) are for the model behavior structure declaration and correspond to the optional segment (D) in table 1. Statements (65) to (89) are for BDD-like diagram manipulation and correspond to the optional segment (F) in table 1.

Statements (21) to (23) declare three macro constants used in the model behavior structure. Statements (26) to (32) declare one local clock and four global synchronizers. At statements (20), (25), and (28), we also add comments to the input file to the RED parser.

We pick some statements in segment (D) to explain how to declare the model behavior structure with REDLIB. The declaration is a sequence of mode declarations. A typical mode declaration can be found from statements (54) to (57). Statement (54) declares a mode whose name is "transm" and whose invariance condition is "x <= LAMBDA." Statement (57) finishes the mode declaration.

Note that the invariance condition is specified without process index reference to variable ‘x.’ In general, in the mode declarations, we assume the transitions are to be executed by a process in the mode (or control location). All local variables without a process index reference are assumed to reference the local variable of the executing process. A constraint with local variables without process index references is called a local constraint. One without is called a global constraint. A global constraint is a special case of local constraint. Local constraints are only allowed in mode declarations.

Inside a mode declaration, we may have transition declarations. Statement (55) declares a transition rule with a synchronizer ‘end,’ a triggering condition “x==LAMBDA,” and two actions “x=0; goto wait;.” There is an implicit local discrete variable, mode, created by REDLIB. The variable records the operation mode of each process. Action “goto wait;” is executed by setting local variable mode to the index of mode ‘wait.’

From statements (65) to (89), we use REDLIB to process BDD-like diagrams for the safety analysis. Statements (66) to (68) print out the variable table, the transition table, and the synchronous transition table to file ‘out.’ The transition table records the rules declared by the users. In the execution, since we are using CSP-style communication channels, several transition rules may have to be combined to make a global simultaneous execution. For example, the rule declared at statement (55) can be executed only when the one at statement (39) is also executed simultaneously. For another example, the rule declared at statement (44) can be executed by
the bus only when the two sender processes both execute a transition with synchronizer ‘?cd’ at the same time. The synchronization of some transition rules are recorded in the synchronous transition table.

For performance consideration, not all transition synchronizations are recorded in the synchronous transition table. REDLIB also records some transition synchronization information in a BDD-like diagrams called red_bulk_xtions(). Statement (70) prints out this BDD-like diagram in a tree-like format.

Statement (71) uses procedure ‘red_diagram()’ to create a BDD-like diagram. red_diagram() allows the users to write a formula as a string in C/C++ format. It automatically translates the formula in the string to a BDD-like diagram. The diagram is stored in variable red_ini of type red_type. Statement (74) pushes the diagram to a stack created by REDLIB. Any BDD-like diagrams in the stack will not be reclaimed in a latter garbage-collection process. Procedure-call red_push() at statement (74) returns the stack index of the frame used to store the just-pushed diagram. Later, for example in statement (75), we can use procedure-call red_stack(ini) to refer to the diagram stored in this frame. We can also use procedure-call red_set_stack() to change the content of a stack frame. For example, in statements (78) and (79).

Statements (76) to (79) calculates an abstract image of the forward reachability and uses this image for the refined invariance condition. Statement (77) gets the diagram for an invariance condition of the model structure with procedure-call red_declared_invariance() and pushes it to the stack. This invariance condition is constructed out of the invariance conditions in the mode declarations. Statement (78) then uses this invariance condition as the system global invariance to calculate an untimed abstraction of the forward reachable state-space from the initial states.

Statement (79) uses the untimed abstraction of the reachable state-space as a new global invariance condition to calculate a finer abstraction of the forward reachable-space from the initial states.

Statements (80) to (86) then do the risk analysis with backward reachability analysis. Statement (81) constructs a BDD-like diagram for contraint


as the risk condition. Statement (82) constructs the backward reachability analysis to the risk condition. Statements (83) to (86) check whether the backward reachability contains any initial states. Statement (83) calls procedure red_normal() to normalize the representation of the BDD-like diagram for the intersection between the reachability and the initial condition. If the normalized representation is \texttt{false}, there is no initial state that can reach the risk states and the

20
system is safe. Otherwise, the system is unsafe.

Note that procedure-calls `red_reach_unimede_fwd()` at statement (78), `red_reach_magnitude_fwd()` at statement (79), `red_reach_bck()` at statement (82), and `red_normal()` at statement (83) all may invoke garbage-collections. Thus it is wise to keep important diagrams in the stack while calling such procedures.

### 3.2 Model declaration through a file

Now we briefly explain the second approach. We can directly input the model from a file in the format of to the parser of RED. REDLIB needs to be used according to the template in table 2. The only difference is that statement (B) in table 2 now replaces statement (B), segment (C), segment (D), and statement (E) in table 1. The input file for the declaration in table 1 is as follows.

```c
/* Three constants used in the specification. */
#define A 26
#define B 52
#define LAMBDA 808

process count = 3; /* 1 is for bus, the others for senders. */
/* One local clock. */
local clock x;
/* 4 synchronizers. */
global synchronizer begin, end, cd, busy;
/* 3 modes for the bus. */
```
mode idle (true) {
    when ?begin (true) may x= 0; goto active; /* 1 */
}

mode active (true) {
    when ?end (true) may x= 0; goto idle; /* 2 */
    when !busy (x >= A) may ; /* 3 */
    when ?begin (x < A) may x= 0; goto collision; /* 4 */
}

mode collision (x < A) {
    when !cd !cd (x < A) may x= 0; goto idle; /* 5 */
}

/* red_comment("3 modes for the senders. */
mode wait (true) {
    when ?begin (true) may x= 0; goto transm; /* 6 */
    when ?cd (true) may x= 0; /* 7 */
    when ?cd (true) may x= 0; goto retry; /* 8 */
    when ?busy (true) may x= 0; goto retry; /* 9 */
}

mode transm (x <= LAMBDA) {
    when !end (x==LAMBDA) may x= 0; goto wait; /* 10 */
    when ?cd (x<B) may x= 0; goto retry; /* 11 */
}

mode retry (x < B) {
    when !begin (x < B) may x= 0; goto transm; /* 12 */
    when ?busy (true) may x= 0; /* 13 */
    when ?cd (x < B) may x= 0; /* 14 */
}

initially
    idle[1] && x[1]==0 && forall i:i>1, (wait[i] and x[i]==0);
risk

Segment (C) and statement (D) in table 2 are then respectively the same as segment (F) and
statement (G) in table 1. Note that all comments are written in C-style. The mode declarations start with reserved words `mode`. The transition rule declarations start with reserved words `when`. The action sequence in a rule starts with reserved word `may`.

In the beginning of the file, we may also declare the number of processes. But it is overridden by the third parameter of procedure-call `red_begin_session()`. In the end, we may also declare the initial condition and the risk condition. They can be referenced in REDLIB with procedure-calls `red_query_diagram_initial()` and `red_risk()`.

### 3.3 Model modification through API

Sometimes we may want to change the behavior structure of a model without changing the variable declarations and process declarations. In this case, we can use the template in table 3 to do the job. Note the piece of code is almost the same as the one from statements (B) to (F) in table 1 except that we change procedure `red_begin_declaration()` to `red_change_declaration(flag_vars, flag_rules)`. This procedure, `red_change_declaration(flag_vars, flag_rules)`, allows us to make some augmentations to the original declarations. Between statements (B') and (E) in table 3, we can make any variable, mode, transition declarations. If variable declarations are also made between the two statements, we only check if they have already been declared or in conflict with some previous declarations.

`flag_vars` may have the following two values.

- **RED_RENEW_VARIABLES**: This option says that in the new round of declaration, all previously declared variables will be discarded. Also a new variable table, a new mode table, a new transition table, and the other derived new tables will be constructed. All previous BDD-like diagrams will be reclaimed.

---

<table>
<thead>
<tr>
<th>red_change_declaration(flag_vars, flag_rules);</th>
<th>(B')</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLE_DECLARATIONS:</td>
<td>(C)</td>
</tr>
<tr>
<td>[OPTIONAL_MODEL_STRUCTURE_DECLARATION:];</td>
<td>(D)</td>
</tr>
<tr>
<td>red_end_declaration();</td>
<td>(E)</td>
</tr>
<tr>
<td>[OPTIONAL_PROCESSING_OF_THE_DIAGRAMS:];</td>
<td>(F)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 3: A template for using REDLIB.
• **RED_ADD_VARIABLES**: This option says that in the new round of declaration, newly declared variables will be added to those previously declared variables. The newly and previously declared will be used together in verification session henceforth. However, any inconsistency in the new declarations and the previous ones will be signaled and terminate the program. For example, if we have a local clock declaration \( x \) in both the previous and the new declarations, then it is considered consistent and OK. On the other hand, if \( x \) is declared as a discrete in the new declaration and a clock in the previous one, then this is considered an inconsistency and program termination happens.

A new variable table, a new mode table, a new transition table, and the other derived new tables will be constructed. All previous BDD-like diagrams will be reclaimed.

• **RED_CHECK_VARIABLES**: This option says that in the new round of declaration, all newly declared variables will be discarded. Only the previously declared variables will be used as for the new verification session henceforth. However, any inconsistency in the new declarations and the previous ones will be signaled and terminate the program. For example, if we have a local clock declaration \( x \) in both the previous and the new declarations, then it is considered consistent and OK. On the other hand, if \( x \) is declared as a discrete in the new declaration and a clock in the previous one, then this is considered an inconsistency and program termination happens.

The previous variable table will be used. All previous BDD-like diagrams will be maintained.

**flag_rules** may have the following two values.

• **RED_RENEW_RULES**: This option says that in the new round of declaration, all previously declared rules will be discarded. Also a new mode table, a new transition table, and the other derived new tables will be constructed.

• **RED_ADD_RULES**: This option says that in the new round of declaration, newly declared rules will be added to those previously declared rules. The newly and previously declared will be used together in verification session henceforth. A new mode table, a new transition table, and the other derived new tables will be constructed.

In fact, procedure-call `red_begin_declaration()` is implemented exactly as `red_change_declaration(RED_RENEW_RULES)`.

However, we cannot change the concurrency sizes at this moment with procedure `red_change_declaration(flag)`.

To change the concurrency sizes, we need to initiate a new REDLIB session.

3.4 Model modification through a file

REDLIB also supports the run-time modification to the model declarations. The procedure is
The two flags are exactly the same as the ones explained in subsection 3.3. The procedure behaves the same as `red_input_model(file)` except that we may discard or add variables and rules in the new declaration according to the values of the two flags.

4 Variable declarations

This is for variable declarations. A global variable can be declared with the following statement.

```c
red_declare_variable(type, lb, ub, name, ...);
```

Note that we can certainly use the corresponding values specified in `redlib.h` instead of the macro names for the constants. But then the application programs could suffer from incompatibility with future versions of REDLIB with redefinitions of the macro constants.

Parameter ‘type’ specifies the type of the variable. Values of the parameter can be as follows.

- RED_TYPE_DISCRETE
- RED_TYPE_POINTER
- RED_TYPE_BOOLEAN
- RED_TYPE_CLOCK
- RED_TYPE_DENSE
- RED_TYPE_SYNCHRONIZER

Type value ‘RED_TYPE_DISCRETE’ creates a discrete variable. Type value ‘RED_TYPE_POINTER’ creates a discrete variable whose range is from zero to the number of processes. Type value ‘RED_TYPE_BOOLEAN’ creates a Boolean variable. Type value ‘RED_TYPE_CLOCK’ creates a clock variable. Type value ‘RED_TYPE_DENSE’ creates a dense variable in linear-hybrid system.

Parameters ‘lb’ and ‘ub’ are only used for discrete variables. These two non-negative integers specify the lower-bound and upper-bound of the range of a discrete variable.

Parameter ‘name’ is a format string (like the one in printf()) for the name of the variable. The format string can be followed by a variable-length list of arguments. This allows for the development of concise code for the variable declaration. For example, we may want to declare 10 binary variables named `s0, s1, ..., s9`. We may use the following code to do the job.

```c
for (i = 0; i < 10; i++)
    red_declare_variable(RED_TYPE_DISCRETE, 0, 1, "s%1d", i);
```
For the construction of concise models, REDLIB also supports the declaration of local variables. A local variable declaration has an instance for each process. The instance of a local variable with name $x$ for process $i$ can be accessed as $x[i]$. A local variable can be declared with the following procedure.

\[
\text{red\_declare\_local\_variable}(\text{type}, \text{lb}, \text{ub}, \text{name}, \ldots);
\]

It also allows for a variable-length list of arguments. \text{name} is again a format string like the format string in printf().

## 5 Expression strings

An arithmetic expression (or expression) is an arithmetic combination of variable references. In general, an expression $E$ of REDLIB can be constructed with the following inductive grammar rules. For convenience, we use letters between apostrophes for terminals of reserved words.

Slanted letters for non-terminals.

\[
\begin{align*}
E &::= M \mid M \, ‘+’ \, E_1 \mid M \, ‘-’ \, E_1 \\
M &::= V \mid V \, ‘*’ \, M_1 \mid N \, ‘/’ \, M_1 \mid N \, ‘%’ \, \text{number} \mid N \, ‘%’ \, \text{macro\_constant} \\
N &::= V \mid \text{number} \mid \text{macro\_constant} \mid ‘#PS’ \mid ‘P’ \\
V &::= \text{var\_name} \mid \text{var\_name}\[‘E’]\mid \text{var\_name} ‘->’ R \mid \text{var\_name}\[‘E’] ‘->’ R \\
R &::= \text{var\_name} \mid \text{var\_name} ‘->’ R
\end{align*}
\]

‘%’ is the modulo (remainder) operator. Now we only allow for modulo operations with constant divisors. \text{number} is an integer. Only when it is the first coefficient, it is allowed be negative. \text{macro\_constant} is a macro symbol for a constant declared with procedure \text{red\_define\_const}().

‘#PS’ is a macro constant for the number of processes in the model. ‘P’ can only be used in local constraints and represents the process index of the executing process.

A local variable can be referenced without an explicit process index only when it is used in a local constraint. The implicit interpretation of the process index is that of the executing process. The process index expressions must contain no clock and dense variables.

REDLIB also supports pointer references. Given a variable reference like ‘$y_1\rightarrow\ldots\rightarrow y_n\rightarrow x’$, $y_1$ through $y_n$ must be either discrete variables or pointers. If the value of $y_i$ in the current state is less than 0 or greater than #PS, the evaluation fails without execution.

## 6 Constraint strings

As can be seen from the example program in section 3, users can write strings for a constraint with REDLIB. There are three types of constraint strings that we can write with REDLIB.
They are the following.

- **Local constraints** are constraints with the following restrictions.
  - No variables in the constraints are synchronizers.
  - No modal operators (until, always, eventually, often, almost).

- **Global constraints** are constraints with the following restrictions.
  - No variables in the constraints are synchronizers.
  - No references to local variables with implicit process indices.

In general, the constraint strings \( F \) of REDLIB can be constructed with the following inductive grammar rules. For convenience, we use letters between apostrophes for terminals of reserved words. Slanted letters for non-terminals.

\[
\begin{align*}
F & ::= D_1 \mid D_1 \text{`implies'} D_2 \\
D & ::= C \mid C \text{`||'} D_1 \\
C & ::= L \mid L \text{`&&'} C_1 \\
L & ::= '(F' ')' \mid 'not' L \mid T
\end{align*}
\]

Here \( F \) is a formula, \( D \) is a disjunction with operator ‘or’ denoted as ‘||’. \( C \) is a conjunction with operator ‘and’ denoted as ‘&&’. \( L \) is a formula with parentheses, or a negation, or a temporal atom. Thus \( F \) is a Boolean combination of temporal atoms.

A temporal atom \( T \) can be of the following types.

- **false**
- **true**
- **Current mode specification.** In a local constraint, the current mode of the executing process can be written as

  \[
  \text{mode}\_\text{name}
  \]

where \( \text{mode}\_\text{name} \) is a declared mode name.

In both local and global constraints, we can also specify that the current mode of a specific process. This can be done as

\[
\text{mode}\_\text{name}[E]
\]

where \( E \) is an arithmetic expression. The square brackets around \( E \) denotes that \( E \) is to be interpreted as a process index. The grammar for arithmetic expressions will be discussed in page 26.

- **Inequalities.** An inequality is of the form \( E_1 \sim E_2 \) where \( E_1 \) and \( E_2 \) are arithmetic expressions and \( \sim \) is one of ‘<’ (less than), ‘<=’ (less than or equal to), ‘==’ (equal to), ‘!=’ (not equal to), ‘>=’ (greater than or equal to), and ‘>’ (greater than).

In REDLIB, there are two classes of variables. Class I consists of the clock and dense variables. Class II consists of the remainings. REDLIB does not allow an inequality
Table 4: correspondence between modal operators in REDLIB and those in the literature

<table>
<thead>
<tr>
<th>REDLIB modal operators</th>
<th>Pnueli’s</th>
<th>Clarke’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>forall</td>
<td>∀</td>
<td>A</td>
</tr>
<tr>
<td>exists</td>
<td>∃</td>
<td>E</td>
</tr>
<tr>
<td>until</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>always</td>
<td>□</td>
<td>G</td>
</tr>
<tr>
<td>eventually</td>
<td>♦</td>
<td>F</td>
</tr>
<tr>
<td>often</td>
<td>□♦ (or ♦∞)</td>
<td>GF</td>
</tr>
<tr>
<td>almost</td>
<td>□♦ (or □∞)</td>
<td>FG</td>
</tr>
</tbody>
</table>

with variables from both classes. Also, if the system is of type RED_SYSMTE_TIMED, an inequality with clock variables must be like one of the following.

- ‘x ∼ number’ ‘x[E] ∼ macro_constant’
- ‘number ∼ x’ ‘macro_constant ∼ x[E]’

- **Quantified constraints over processes.** We can make quantified constraints over the processes with REDLIB. Such a constraint can be in one of the following four forms.

  - ‘forall’ var_name ‘,’ ‘L₁’
  - ‘forall’ var_name ‘:’ ‘L₁ ‘,’ ‘L₂’
  - ‘exists’ var_name ‘,’ ‘L₁ ‘,’ ‘L₂’
  - ‘exists’ var_name ‘:’ ‘L₁ ‘,’ ‘L₂’

  var_name is a quantified variable name over the scope of L₁ and L₂. The value of var_name is over the set of process indices. var_name may or may not be declared in the variable declaration segment. In this scope, it is treated as a pointer variable.

- **Clock reset constraints.** We can evaluate a constraint with the assumption that a clock reads zero. Such a constraint can be written as follows.

  ‘reset’ clock_name L₁

  Note that such constraints can only be used in global constraints.

- **Temporal formulas.** We can also use TCTL formulas with fairness assumptions. Such a formula can only be used in global constraints and is in one of the following four forms.

  - ‘forall’ fairness sop ‘L₁’
  - ‘exists’ fairness sop ‘L₁’
  - ‘forall’ fairness ‘L₁ ‘until’ ‘L₂’
  - ‘exists’ fairness ‘L₁ ‘until’ ‘L₂’

  sop is a modal operators. The possibilities for sop are ‘eventually’, ‘always’, ‘often’, and ‘almost’. The correspondence between the modal operators to the traditional notations in the literature can be found in table 4.

  fairness is the fairness constraint. It can be an empty string, a weak fairness assumption, a strong fairness assumption, or both. A weak fairness assumption is specified as follows.

  ‘weak’ ‘{‘ W₁ ‘;’ ‘… ‘;’ Wₙ ‘;’ }’

  where W₁,…,Wₙ are either global constraints or event constraints. For a weak assumption
global constraint $W'$, it specifies that along the computation, eventually all states in the computation satisfies $W'$. For a weak assumption event predicate $W'$, it specifies that along the computation, eventually the synchronizers in all transitions must satisfy $W'$.

A strong fairness is similarly defined as

$$\text{'strong'} \{\{ S_1 ;; \ldots ;; S_m \} \}$$

For a weak assumption global constraint $S'$, it specifies that along the computation, there are infinitely many states in the computation satisfies $S'$. For a weak assumption event predicate $S'$, it specifies that along the computation, there are infinitely many transitions with synchronizers that satisfy $S'$. Note that the “infinitely many” is interpreted with the divergence of computations. Technically, this means that for any time value $t$, there is a $t' > t$ such that the strong fairness assumption is satisfied at time $t'$ along the computation.

Specifically, we have the following explanation of typical combinations of the components. A computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$ if and only if it has

- infinitely many states satisfying global constraints in $S_1, \ldots, S_m$,
- infinitely transitions satisfying event predicates in $S_1, \ldots, S_m$, and
- a tail computation (suffix) along which
  * all states satisfy global constraints in $W_1, \ldots, W_m$, and
  * all transitions satisfy event predicates in $W_1, \ldots, W_m$.

With this concept, we can explain the various combinations of the components in a modal formula as follows.

- A formula like

  $$\text{forall strong } \{S_1; \ldots; S_m\} \text{ weak } \{W_1; \ldots; W_m\} F_1 \text{ until } F_2$$

  specifies states from which if a computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$, then along the computation, $F_1$ is true until eventually $F_2$ is true.

- A formula like

  $$\text{exists strong } \{S_1; \ldots; S_m\} \text{ weak } \{W_1, \ldots, W_n\} F_1 \text{ until } F_2$$

  specifies states from which there is a computation satisfying strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$. Moreover, along the computation, $F_1$ is true until eventually $F_2$ is true.

- A formula like

  $$\text{forall strong } \{S_1; \ldots; S_m\} \text{ weak } \{W_1, \ldots, W_n\} \text{ eventually } F_2$$

  specifies states from which if a computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$, then along the computation,
eventually $F_2$ is true.

- A formula like

\[
\text{exists strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ eventually } F_2
\]

specifies states from which there is a computation satisfying strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$. Moreover, along the computation, eventually $F_2$ is true.

- A formula like

\[
\text{forall strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ always } F_2
\]

specifies states from which if a computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$, then along the computation, only transitions satisfying $E_1$ can happen.

- A formula like

\[
\text{exists strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ always } F_2
\]

specifies states from which there is a computation satisfying strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$. Moreover, along the computation, only transitions satisfying $E_1$ can happen.

- A formula like

\[
\text{forall strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ often } F_2
\]

specifies states from which if a computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$, then along the computation, infinitely many times a transition satisfying $E_1$ happens and ending at a state satisfying $F_2$.

- A formula like

\[
\text{exists strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ often } F_2
\]

specifies states from which there is a computation satisfying strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$. Moreover, along the computation, infinitely many times a transition satisfying $E_1$ happens and ending at a state satisfying $F_2$.

- A formula like

\[
\text{forall strong } \{ S_1; \ldots; S_m \} \text{ weak } \{ W_1, \ldots, W_n \} \text{ almost } F_2
\]

specifies states from which if a computation satisfies strong fairness assumption $S_1, \ldots, S_m$ and weak fairness assumption $W_1, \ldots, W_m$, then the computation has a tail computation along which, immediately after all transitions that satisfy $E_2$, $F_2$ is always true.

- A formula like
exists strong \{ S_1; \ldots; S_m \} weak \{ W_1; \ldots, W_n \} almost F_2

specifies states from which there is a computation satisfying strong fairness assumption \( S_1, \ldots, S_m \) and weak fairness assumption \( W_1, \ldots, W_m \). Moreover, along the computation, there is a tail computation along which, immediately after a transition that satisfies \( E_1, F_2 \) is always true.

Please be again reminded that phrases strong \{ \ldots \} and weak \{ \ldots \} are both optional.

REDLIB has the power to automatically construct the BDD-like diagrams for such constraints represented as strings. For an event constraint string \( F \), we can use procedure-call red_diagram(\( F \)) to construct the BDD-like diagram for the constraint. For a global constraint without modal operators \( F \), we can also use procedure red_diagram(\( F \)) to construct the BDD-like diagram. In fact, red_diagram() allows for variable-length arguments in the style of printf(...).

**Example 7:** Given a local clock variable \( x \), a local pointer variable \( p \), and a global discrete variable \( a \), procedure-call

```
“red_diagram("p[2]->x <= 3 && x[1] > 2 && a<= 5 && a->p > 1");”
```

returns a BDD-like diagram. On the other hand, procedure-call

```
“red_diagram("p->x <= 3 && x[1] > 2 && a<= 5 && a->p > 1");”
```

is illegal and returns NULL.

We may also write

```
red_diagram("%s==%d && %s!=%d", a, 1, b, 0)
```

When \( a \) is "friend" and \( b \) is "enemy", the procedure-call constructs a diagram for "friend==1 && enemy==0".

REDLIB also allows the users to construct diagrams from formula strings with local variable references. For such local constraints, we need to tell REDLIB how to interpret local variable names. Specifically, we need to tell REDLIB the index of the executing process to construct the corresponding BDD-like diagrams. This can be done through procedure-call like red_diagram_local(\( p, F, \ldots \)) where \( p \) is the index of an executing process.

**Example 8:** Given a local clock variable \( x \), a local pointer variable \( p \), and a global discrete variable \( a \), procedure-call

```
“red_diagram("p[2]->x<=%d && x[1]%id && a<= 5 && a->p > 1", 3, 2);”
```
returns the same BDD-like diagram as procedure-call

```
    "red_diagram_local(2, "p->x=%d && x[1]>%1d && a<= 5 && a->p > 1", 3, 2);
```

does. Note here we interpret variable name p as p[2].

7 Model structure declarations

This optional segment consists of a sequence of mode declarations. A mode is a control location. A control location can be labeled with an invariance condition. Inside a mode, we can then declare a sequence of transition rules. The template for a mode declaration with REDLIB is as follows.

```c
red_begin_mode(name, inv);
...
RED_RULES;
...
red_end_mode();
```

Here parameter `name` is the name of the mode. `inv` is a local constraint string that specifies the invariance condition of the executing process in the mode.

Specifically, REDLIB have the following two parameterized procedures to begin and end a mode declaration.

```c
int red_begin_mode(int flag_urgent, char *name, char *inv, ...)
```

This procedure starts the declaration of a mode with name `name` and invariance condition expressed as a format string `inv`. This procedure again is parameterized (as in the printf style) with variable length arguments. Users can use place-holders in the string `inv` and fill them in with the variable-length arguments represented with the three dots. There is also a flag argument `flag_urgent`. This flag tells REDLIB whether this mode is an urgent mode or not. An urgent mode does not allow time to progress before leaving it. The two possible values of `flag_urgent` are

```
    RED_MODE_FAIL if the operation fails. Otherwise it returns RED_MODE_SUCCESS.
```

```c
int red_end_mode()
```
This procedure ends the declaration of a mode. It does some primitive check that the mode declaration ends properly. The procedure returns macro constant RED_MODE_FAIL if the declaration does not end correctly. Otherwise it returns RED_MODE_SUCCESS.

7.1 Rules for rate specifications of dense variables

For convenience, a fractional expression is either an integer (or a macro constant) or an expression like c/d where c and d are integers (or macro constants). An interval expression is of the form

\[ \text{lbrac } K_1 , K_2 \text{ rbrac} \]

where \( K_1 \) and \( K_2 \) are fractional expressions, \( \text{lbrac} \) is either ‘[‘ (for left-closed intervals) or ‘(‘ (for left-open intervals), and \( \text{rbrac} \) is either ‘]’ (for right-closed intervals) or ‘)’ (for right-open intervals).

There can be zero or many rules in between a red_begin_mode() statement and a matching red_end_mode() statement. There are two types of rule. The first is for the declaration of the changing rate of a dense variable in a linear-hybrid system. A rule of this type can be declared with the following procedure-call.

```c
int red_dense_rate( var_name , rate_range)
char * var_name, /* a string for the variable name */
* rate_range; /* a string for the rate interval */
```

Here parameter `var_name` is a string that specifies a declared dense variable. `rate_range` is a string that specifies the range of rate. `rate_range` can be in one of the following three forms.

- A string for an integer constant (or macro constant).
- A string for an expression like c/d where c and d are two integer constants (or macro constants).
- A string for an interval expression.

Example 9 : Procedure-call “red_dense_rate("x", "3");” specifies that in a mode, the rate of variable x is 3.

Procedure-call “red_dense_rate("x", "(3/5,7]");” specifies that in a mode, the rate of variable x is in (3/5, 7].

7.2 Rules for discrete transitions

A CSP-style synchronization primitive is in one of the following forms.
Here `!` intuitively means the sending of a message while `?` means the receiving of a message. `sync_name` is a declared synchronizer variable name. `qvar_name` is an undeclared quantifier variable of type pointer. When a synchronizer name is followed by something like `"@ qvar_name"`, `qvar_name` can be used in the scope of a transition to reference the process index of the process that corresponds to this synchronizer. When a synchronizer name is followed by something like `"@ (E)"`, arithmetic expression `E` specifies the index of the process that must correspond to this synchronizer.

A discrete transition in a mode can be declared with the following procedure.

```c
int red_transition(char *rule, ...)
```

Here argument `rule` is a format string of the following syntax.

```plaintext
when trigger may actions
```

`trigger` is a string for the triggering condition of the transition while parameter `actions` is a string for a sequence of actions. Note that `actions` must consist of at least one action. However the action can be a null action represented by a single semicolon.

Since the procedure is with variable-length arguments, we can use the variable-length arguments to substitute in the place-holders in the format string `rule`.

A triggering condition is a sequence of CSP-style synchronization primitives followed by a local constraint in parentheses. A triggering condition conceptually means that when all the synchronization primitives are corresponded and the local constraint is satisfied, then the transition CAN happen. Note since the model is nondeterministic in nature, ‘CAN’ does not means ‘must.’ When a transition happens, it execute its sequence of actions instantaneously.

Since the explanation is long, we leave it to the following paragraphs.

An action can be of the following ten forms.

1. **Assignment action**: Such an action is like

   ```
   V = E ;
   ```

   Here `V` is a variable reference defined in page 26. Note that `V` can be with process index and pointer links. Also `E` is an expression defined in page 26.

   Here are some of the restrictions.

   - If the system type is `RED_SYSTEM_TIMED` and `V` is a clock variable reference, then `E` can only be either a number (or macro constant) or an expression like `y + c` where `y`
is another clock variable reference.

- If $V$ is a clock variable reference, then there is no discrete variable in $E$.

- If $V$ is a discrete variable reference, a pointer reference, or a Boolean variable reference, then there is neither clock variable nor dense variable in $E$.

Example 10: Suppose we are given a local clock variable $x$, a local pointer variable $p$, and a global discrete variable $a$. We may have the following simple actions.

\[
\begin{align*}
    x & = 3; \\
    x & = x + 3; \\
    x & = x[2]; + \\
    a & = 3a + 2p - 3;
\end{align*}
\]

We may also have the following actions with pointer references.

\[
\begin{align*}
    p->p->x & = 3; \\
    a->x & = p->x + 3; \\
    p[1]->x & = p[2]->x; + \\
    a->p & = 3*(a+2*p[2]->p) - 3;
\end{align*}
\]

However, the syntax combination of such actions can be very complicated. We have to ask for your understanding that REDLIB is a non-profit library developed in the academia. If you ever find some actions not supported as specified, please email to farn@cc.ee.ntu.edu.tw and we will try to correct it as soon as possible.

2. Interval assignment action: Such an action is like

\[
V \text{ ‘in’ } I \text{ ‘;’}
\]

Here $I$ is an interval expression. Semantically, this action assigns an arbitrary value in the interval expression to variable reference $V$.

Example 11: Suppose we are given a local clock variable $x$, a local pointer variable $p$, and a global discrete variable $a$. We may have the following simple actions.

\[
\begin{align*}
    x & \text{ in } [3,5); \\
    a & \text{ in } [3,7];
\end{align*}
\]

We may also have the following actions with pointer references.

\[
\begin{align*}
    p->p->x & \text{ in } [3,5); \\
    a->x & \text{ in } (0,7]; \\
    p[1]->x & \text{ in } (2,8); \\
    a->p & \text{ in } [1,2];
\end{align*}
\]

3. Increment action: Such an action is like

\[
V \text{ ‘++’ } number \text{ ‘;’}
\]

Here $number$ can also be a macro constant. Variable reference $V$ must be a discrete variable or a pointer. Semantically, this action increments the value of variable reference $V$ by the value of $number$. 

35
Example 12: Suppose we are given a local clock variable \( x \), a local pointer variable \( p \), and a global discrete variable \( a \). We may have the following simple actions.

\[
a  \mathbin{++} 5;
\]

We may also have the following actions with pointer references.

\[
a \mathbin{-}\mathbin{p}  \mathbin{++} (A+3);
\]

Here \( A \) is a macro constant. Note that no variables are allowed in the increment offset.

4. Decrement action: Such an action is like

\[
V  \mathbin{-}\mathbin{-} \text{ number} \mathbin{;};
\]

Here \text{ number} can also be a macro constant. Variable reference \( V \) must be a discrete variable or a pointer. Semantically, this action decrements the value of variable reference \( V \) by the value of \text{ number}.

Example 13: Suppose we are given a local clock variable \( x \), a local pointer variable \( p \), and a global discrete variable \( a \). We may have the following simple actions.

\[
a  \mathbin{--} 5;
p[2]  \mathbin{--} 5;
\]

We may also have the following actions with pointer references.

\[
a \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{p}  \mathbin{--} (3*A+1);
p[1] \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{p}  \mathbin{--} (3*(A+B*2));
\]

Here \( A \) and \( B \) are macro constants. Note that no variables are allowed in the increment offset.

5. Goto action: This action changes the control location (or mode) of the executing process. It is of the following format.

\[
\text{’goto’}  \text{ mode\_name}  \mathbin{;};
\]

As we have said, there is a system-generated local discrete variable \text{ mode\_name} which records the current mode of the executing process. This action merely assign the index of mode \text{ mode\_name} to the local variable of \text{ mode\_name}.

6. Empty action: This is simply a semicolon \( \mathbin{;} \) which conceptually means no-operation.

7. Block action: A block action is a sequence of action enclosed in a pair of parentheses.

Example 14: Suppose we are given a local clock variable \( x \), a local pointer variable \( p \), and a global discrete variable \( a \). Then we may have the following block action.

\[
\{  x[2] \mathbin{-}\mathbin{p} = 3;  p \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{p} \mathbin{-}\mathbin{x} = x + 3;  a = p[2] \mathbin{-}\mathbin{p}; \}
\]

8. Loop action: This is like a while-loop in the traditional program languages. It is of the following format.

\[
\text{’when’} \ (\mathbin{’\mathbin{’} F \mathbin{’} \mathbin{’}’) \text{ action}
\]

Here \( F \) is a formula and \text{ action} is an action.
\textbf{Example 15} : Suppose we are given a local clock variable $x$, a local pointer variable $p$, and a global discrete variable $a$. Then we may have the following loop action.

\begin{verbatim}
when (a > 3) a--1;
\end{verbatim}

We may have the following more complex loop action.

\begin{verbatim}
when (a > 3) { x[2]->p = 3; p->p->x = x + 3; a = p[2]->p; }
\end{verbatim}

9. \textit{If action}: This is like an if-statement in the traditional program languages. It is of the following format.

\begin{verbatim}
'if' '()' 'F' 'action
\end{verbatim}

Here $F$ is a formula and \textit{action} is an action.

\textbf{Example 16} : Suppose we are given a local clock variable $x$, a local pointer variable $p$, and a global discrete variable $a$. Then we may have the following loop action.

\begin{verbatim}
if (a > 3) a--1;
\end{verbatim}

We may have the following more complex loop action.

\begin{verbatim}
if (a > 3) { x[2]->p = 3; p->p->x = x + 3; a = p[2]->p; }
\end{verbatim}

10. \textit{If-else action}: This is like an if-statement in the traditional program languages. It is of the following format.

\begin{verbatim}
'if' '()' 'F' 'else' 'action1' 'else' 'action2
\end{verbatim}

Here $F$ is a formula and \textit{action1} and \textit{action2} are two actions.

\textbf{Example 17} : Suppose we are given a local clock variable $x$, a local pointer variable $p$, and a global discrete variable $a$. Then we may have the following loop action.

\begin{verbatim}
if (a > 3) a--1; else a++2;
\end{verbatim}

We may have the following more complex loop action.

\begin{verbatim}
if (a > 3) { p->p->x = x + 3; a = p[2]->p; } else { x[2]->p = 3; a++2; }
\end{verbatim}

\section{Accesses to the model structures}

\subsection{Declared model attributes}

We may use \texttt{REDLIB} to access the information stored in a model structure.

\textbf{Variable table}

The table includes all the system-generated variables, including \textit{false}, \textit{true}, all the declared variables, some system clocks, discrete variables for the marking the synchronization between
Macro constants | Attributes
---|---
RED_VAR_TYPE | the type of the variable
RED_VAR_SCOPE | the scope of the variable, local or global
RED_VAR_SYSGEN | a boolean value to tell if this variable is system-generated
RED_VAR_PRIMED | a boolean value to tell if this is a primed variable or not
RED_VAR_PROC | if this is a global variable, the value is zero; if this is a local one, the value is the index of the process to which the variable belongs.
RED_VAR_LB | if this is a discrete (or pointer) variable, this is the lower-bound of its values.
RED_VAR_UB | if this is a discrete (or pointer) variable, this is the upper-bound of its values.
RED_VAR_CLOCK1 | if this is a clock inequality variable, this is the clock index of the first clock.
RED_VAR_CLOCK2 | if this is a clock inequality variable, this is the clock index of the second clock.
RED_VAR_CLOCK_INDEX | if this is a clock variable, this is the clock index of the clock.

Table 5: Attribute type values of the variable table

processes, clock inequality variables, linear-hybrid inequality variables, and primed variables. The index of a variable in the table marks its evaluation ordering in the BDD-like diagrams.

Note that it is in general impossible to predict the number of linear-hybrid expressions that is to be constructed in the verification process of a linear-hybrid system. Thus REDLIB does not allocate a unique variable index for each linear-hybrid expression. Instead, a class of linear-hybrid expressions are mapped to a variable index.

For the variable table, we have the following procedures.

```c
int red_query_var_count()
```

This procedure returns the size of the variable table.

```c
int red_query_var_attribute( vi , attr )
```

```c
int vi , /* a variable index */
attr; /* an attribute index */
```

This procedure returns an integer attribute of the variable with index `vi`. The value of parameter `attr` specifies the attribute whose value is to return. Values of parameter `attr` are declared in `redlib.h` and can be found in table 5.

```c
char *red_query_var_name(vi)
```
int \textit{vi}; /* a variable index */

This procedure returns the name of the variable with index \textit{vi}.  

int red_query_var_index(vname, pi)
char \*vname; // a declared variable name int \textit{pi}; /* a process index */

This procedure returns the index of the declared variable with name \textit{vname} owned by process index \textit{pi}. If \textit{vname} is the name of a global variable, then the value of \textit{pi} does not matter.  

int red_query_clock_count()

This procedure returns the number of clocks, including global ones and the local copies of all local clocks, in the variable table.  

int red_query_clock_var(ci)
int \textit{ci}; /* a clock index */

This procedure returns the variable index of clock \textit{ci}.  

int red_query_zone_index(ci_1, ci_2)
int \textit{ci}_1, \textit{ci}_2; /* the left and right clock indices */

This procedure returns the variable index of expression for the difference between clocks \textit{ci}_1 and \textit{ci}_2.  

Transition tables

Note that a transition table stores all the information of those declared transitions in the input model structure. In addition, REDLIB adds in a null transition, with transition index 0, that does nothing. In general, transition information in a transition table is not enough for the execution of the model. Please be reminded that there may be synchronizations among the transitions. REDLIB constructs a synchronous transition table and a bulk synchronization diagram out of a transition table. The synchronization transition table and the bulk synchronization diagram together record the information necessary for the synchronous execution among the processes. We may use the following procedures to access information related to the transitions.

int red_query_xtion_count()

This procedure returns the number of transitions in a model structure.  

int red_query_xtion_attribute(xi, attr)
int \textit{xi}, /* a declared transition index */
Macro constants

<table>
<thead>
<tr>
<th>Macro constant</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED_XTION_SYNC_COUNT</td>
<td>the number of synchronization primitives</td>
</tr>
<tr>
<td>RED_XTION_SRC_MODE</td>
<td>the source mode index of the declared transition</td>
</tr>
<tr>
<td>RED_XTION_DST_MODE</td>
<td>the destination mode index of the declared transition</td>
</tr>
<tr>
<td>RED_XTION_PROCESS_COUNT</td>
<td>the number of processes that may execute it</td>
</tr>
<tr>
<td>RED_XTION_ACTION_COUNT</td>
<td>the number of top level actions</td>
</tr>
</tbody>
</table>

Table 6: Attribute type values of the declared transition table

`attr; /* an attribute index */`

This procedure returns an integer attribute of the declared transition with index `xi`. The value of parameter `attr` specifies the attribute whose value is to return. Values of parameter `attr` are declared in `redlib.h` and can be found in table 6.

```
redgram *red_query_xtion_trigger_diagram(xi, pi)
int xi, /* a declared transition index */
pi; /* a process index */
```

This procedure returns a BDD-like diagram that is the triggering condition of transition `xi` for process `pi`.

```
char *red_query_xtion_action_string(xi)
int xi; /* a declared transition index */
```

This procedure returns a string for the action of transition `xi`.

```
char *red_query_xtion_string(xi)
int xi; /* a declared transition index */
```

This procedure returns a string for transition `xi`.

```
int red_query_xtion_process(xi, i)
int xi, /* a declared transition index */
i; /* an index to the processes that execute transition xi */
The restriction is that `i < red_query_xtion_process_count(xi)`. When the restriction is satisfied, this procedure returns the index of the `i`th process that can execute transition `xi`.

Process attributes

We have the following procedures to let the users access the attributes of processes.

```
int red_query_process_count()
```

40
This process returns the number of processes in the system.

```c
int red_query_process_xtion_count(pi)
int pi; /* a process index */
```

This process returns the number of declared transitions that can be executed by process `pi`.

```c
int red_query_process_xtion(pi, xi)
int pi, /* a process index */
    xi; /* a declared transition index */
```

This process returns the index of the `xi`'th declared transition that can be executed by process `pi`.

### Mode attributes

We have the following procedures to let the users access the attributes of the declared modes.

```c
int red_query_mode_count()
```

The procedure returns the number of modes declared in the system.

```c
int red_query_mode_attribute(mi, attr)
int mi, /* a declared mode index */
    attr; /* an attribute index */
```

This procedure returns an integer attribute of the declared mode with index `mi`. The value of parameter `attr` specifies the attribute whose value is to return. Values of parameter `attr` are declared in `redlib.h` and can be found in table 7.

```c
char *red_query_mode_name(mi)
int mi; /* a mode index */
```

The procedure returns a string for the name of mode `mi`.

```c
int red_query_mode_xtion(mi, xi)
int mi, /* a mode index */
    xi; /* an index to a transition declared in mode mi */
```

The procedure returns an integer for the index of `xi`'th declared transition in mode `mi`. If `mi` is not a valid mode index, `xi` is smaller than zero, or `xi` is no less than `red_mode_attribute(mi, RED_MODE_XTION_COUNT)`, `-1` will be returned.

```c
int red_query_mode_process(mi, pi)
```
Macro constants | Attributes
--- | ---
**RED_MODE_XTION_COUNT** | the number of declared transitions that can be executed in this mode

**RED_MODE_URGENT** | a Boolean flag to tell if the mode is urgent. It returns a non-zero value if and only if it is. In an urgent mode, time is not allowed to progress.

**RED_MODE_PROCESS_COUNT** | the number of processes that can stay in this mode

**RED_MODE_RATE_LB_NUM** | the numerator of the lower-bound of the declared change rate of a dense variable in the mode in a linear-hybrid system. Note that if the returned value is -$1*\text{red\_hybrid\_oo()}$, it means no lower-bound. If the returned value is even, it actually means a closed rational lower-bound with numerator equal to half the returned value. If the returned value is odd, it means an open rational lower-bound with numerator equal to half the returned value minus one, i.e., \((\text{red\_mode\_attribute}(mi, \text{RED\_MODE\_RATE\_LB\_NUM})-1)/2\).

If the system is not linear-hybrid or \(vi\) does not index a dense or clock variable, \text{REDLIB} might not withdraw all your savings in the bank and burn the computer. But we don’t guarantee the proper running of the verification task.

**RED_MODE_RATE_LB_DEN** | the denominator of the lower-bound of the declared change rate of a dense variable in the mode in a linear-hybrid system. Note that if the returned value is -$1*\text{red\_hybrid\_oo()}$, it means no lower-bound.

**RED_MODE_RATE_UB_NUM** | the numerator of the upper-bound of the declared change rate of a dense variable in the mode in a linear hybrid system. Note that if the returned value is \(\text{red\_hybrid\_oo()}\), it means no upper-bound. If the returned value is even, it actually means a closed rational upper-bound with numerator equal to half the returned value. If the returned value is odd, it means an open rational upper-bound with numerator equal to half the returned value plus one, i.e., \((\text{red\_mode\_attribute}(mi, \text{RED\_MODE\_RATE\_LB\_NUM})1)/2\).

**RED_MODE_RATE_UB_DEN** | the denominator of the upper-bound of the declared change rate of a dense variable in the mode in a linear-hybrid system. Note that if the returned value is \(\text{red\_hybrid\_oo()}\), it means no upper-bound.

---

Table 7: Attribute type values of the declared mode table
Table 8: System type macro constants

<table>
<thead>
<tr>
<th>RED_SYSTEM_UNTIMED</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED_SYSTEM_TIMED</td>
<td>2</td>
</tr>
<tr>
<td>RED_SYSTEM_HYBRID</td>
<td>3</td>
</tr>
</tbody>
</table>

The procedure returns an integer for the index of the pi'th declared process that may stay in mode mi. If mi is not a valid mode index, pi is smaller than one, or pi is greater than red_process_count(), '-1' will be returned.

```c
redgram *red_query_mode_invariance_diagram(mi, pi)
```

The procedure returns a BDD-like diagram for the invariance condition of the mode for process pi.

8.2 Some derived system attributes

There are several information pieces important for the manipulation of dense-time spaces.

```c
/* Procedure for invariance conditions of all processes */
redgram *red_query_declared_invariance_diagram()
```

The procedure returns a BDD-like diagram for the invariance condition constructed out of the mode invariance conditions and their executing processes. Intuitively, it is the conjunction of the invariance conditions of all the processes. The invariance condition of a process is the disjunction of the invariance conditions of all locations that can be reached by the process in its automaton graph. Some simple control flow analysis is done to tell which process can execute in a mode.

```c
int red_system_type()
```

The procedure returns an integer for the system type of the model structure. The system types are indexed with the macro constants defined in redlib.h in table 8.
8.3 Synchronous transitions

As we have said, declared transitions may not be executable by themselves if they are with synchronizers. Several declared transitions can be combined to form a synchronous transition. The synchronous transitions are partitioned and stored in two ways. The partition is determined by a REDLIB parameter returned with procedure `red_query_sync_bulk_depth()`.

```c
int red_query_sync_bulk_depth()
```

This procedure returns a threshold integer value. Synchronous transitions with the number of processes no greater than this threshold will be given unique process indices and can be accessed and executed independently. Synchronous transitions without in a synchronous transition with the number of processes greater than this threshold will be saved in `red_query_diagram_xtion_sync_bulk()` and can only be used in execution as a whole.

When the model is read from an input file, the default value of `red_query_sync_bulk_depth()` is 2 for risk analysis and model-checking and 3 for simulation checking.

```c
void red_set_sync_bulk_depth(int d);
```

This procedure changes the value of `red_sync_bulk_depth()` to `d`. This procedure, if executed, must be executed before invoking either procedure `red_end_declaration()` or procedure `red_input_model` in the current session. The reason is that in those two procedures, REDLIB uses the value of `red_sync_bulk_depth()` to partition synchronous transitions into the synchronous transition table and the `red_xtion_sync()`. Violation of the restriction, um, has not effect.

If a synchronous transition involves no more than `redlib_sync_bulk_depth()` declared transitions, it is assigned a non-negative synchronous transition index. All information about this synchronous transition can then be accessed through the following procedures. We have designed a set of procedures to allow the users to access the synchronous transitions.

```c
int red_query_sync_xtion_count()
```

This procedure returns the number of synchronous transitions in a model structure constructed out of the declared transitions. Thus, integers 0 through `red_query_sync_xtion_count−1` are the indices to the valid synchronous transitions declared in a model. Moreover, there are two special synchronous transitions. The first is with index 0 and is the null synchronous transition that does nothing. The other is with
index red_query_sync_xtion_count−1 which represents the bulk synchronous transition.

```c
int red_sync_xtion_party_count(sxi)
int sxi; /* a synchronous transition index */
    This procedure returns the number of processes involved in the synchronization of synchronous transition sxi.
```

```c
int red_sync_xtion_party_process(sxi, pti)
int sxi; /* a synchronous transition index */
int pti; /* a party index */
    This procedure returns the process index for the party with party index pti involved in the synchronization of synchronous transition sxi. A valid party index must be in the range from 0 to red_sync_xtion_party_count(sxi)-1. The procedure aborts when either sxi is not a valid synchronous transition index or pti is not a valid party index for synchronous transition sxi.
```

```c
int red_sync_xtion_party_xtion(sxi, pti)
int sxi; /* a synchronous transition index */
int pti; /* a party index */
    This procedure returns the transition index for the party with party index pti involved in the synchronization of synchronous transition sxi. A valid party index must be in the range from 0 to red_sync_xtion_party_count(sxi)-1. The procedure aborts when either sxi is not a valid synchronous transition index or pti is not a valid party index for synchronous transition sxi.
```

```c
redgram *red_sync_xtion_trigger(sxi)
int sxi; /* a synchronous transition index */
    This procedure returns a BDD-like diagram that is the triggering condition of synchronous transition sxi.
```

```c
char *red_sync_xtion_action_string(sxi)
int sxi; /* a synchronous transition index */
    This procedure returns a string for the actions of synchronous transition sxi.
    Especially, we have the following procedure that outputs a synchronous transition in a special format that can be accepted for the calculation of preconditions or postconditions by REDLIB.
```

```c
char *red_sync_xtion_string(sxi)
int sxi; /* a synchronous transition index */
```

45
This procedure returns a string for synchronous transition \( sxi \).

Suppose that synchronous transition \( sxi \) involves the execution of declared transitions of \( n \) participating processes. This output string \( \text{red_sync_xtion_string}(sxi) \) is of the following form.

\[
\text{\textquote{sync} \textquote{xtion} XTION_1 \ldots XTION_n}
\]

Here \( F \) is a global constraint string defined in page 27. \( XTION_i, 1 \leq i \leq n \), are abbreviated strings for declared transitions. Each \( XTION \) is a string of the following syntax.

\[
i \textquote{';'} \textquote{MODE} \textquote{('} F \textquote{')} \textquote{ACTS}
\]

Here \( i \) is a constant for a process index. \( F \) is a local constraint string for the triggering condition. \( ACTS \) is a sequence of local actions defined in pages 34 to 37. \( MODE \) is a name of a mode at which the following transition is to be executed by process \( i \).

**Example 18**: Suppose we have the timed systems described in figure 2 and subsection 3.2. The fifth declared transition is declared in mode \textquote{idle} as follows.

\[
\text{when } \textquote{!cd} \textquote{!cd} (x < A) \text{ may } x=0; \text{ goto idle; } /* 5 */
\]

The seventh and eighth declared transition are then in mode \textquote{wait} respectively as follows.

\[
\begin{align*}
\text{when } \textquote{?cd} (\textquote{true}) & \text{ may } x=0; /* 7 */ \\
\text{when } \textquote{?cd} (\textquote{true}) & \text{ may } x=0; \text{ goto retry; } /* 8 */
\end{align*}
\]

Please be reminded there are three processes in the system. The execution of transition 5 sends out two signals \textquote{cd} while those of 7 and 8 respectively receive one signal \textquote{cd}. Thus the execution of transition 5 by process 1 and those of 7 and 8 respectively by processes 2 and 3 together make a legitimate synchronous transition with CSP-style synchronization primitives. Suppose that this synchronous transition is indexed 2. Then the invocation of

\[\text{red_sync_xtion_string}(2)\]

prints out the following string.

"\textquote{sync} xtion 1:idle(x<A) may x=0; goto idle; 2:wait(true) may x=0; 3:wait(true) may x=0; goto retry;"

This string, as will be noted, could be used as a parameter to some \textquote{REDLIB} procedures to calculate the precondition or post-condition of the synchronous transition. The string is simply the concatenation of the declared transitions without the strings for \textquote{when} and the synchronization primitives.
For all synchronous transitions that involve more than \texttt{red\_sync\_bulk\_depth()} declared transitions, we pack them in BDD-like diagrams accessible through the following procedures.

\begin{verbatim}
redgram *red_query_diagram_xtion_sync_bulk()
\end{verbatim}

This procedure returns a BDD-like diagram that characterizes the synchronization among the declared transitions. Those synchronization declared by the synchronous transitions are not included in this diagram.

\begin{verbatim}
redgram *red_query_diagram_xtion_sync_bulk_with_trigger()
\end{verbatim}

This procedure returns a BDD-like diagram that characterizes the synchronization among the declared transitions. Those synchronization declared by the synchronous transitions are not included in this diagram. Triggering conditions of the participating transitions are also incorporated in the diagram.

9 Basic diagram operations

\texttt{REDLIB} supports two ways of diagram manipulations. The first is the traditional one in which the users can make initial diagrams, conjunct them, disjunction them, and complement them. This way should be good for some basic symbolic manipulations to carry out ordinary state-space manipulations and verification tasks. The procedures to support this way of basic diagram manipulation are explained in subsection 9.1.

However the first way does not quite allow the users to explore the structure of diagrams and make specialized diagram operations. Sometimes, the users may want to experiment with specialized diagram reduction techniques or normalization techniques. Sometimes they may want to construct some special diagrams which cannot be done efficiently with those basic manipulation procedures. For example, we may want to find out the maximum value of a particular discrete variable in a diagram. We may also want to enforce the following constraints,

For every \(k1\) and \(k2\), if \(v1=k1\) and \(v2=k2\), then \(v3=k1+k2\).

Such constraints cannot be efficiently constructed out of the basic manipulation procedures since we may have to enumerate the ranges of variables \(v1\) and \(v2\). \texttt{REDLIB} supported a set of procedures for the customized manipulation of diagrams. These procedures allows the users to examine the structures inside a diagram in a recursive way. The way to do this is to call procedure \texttt{red\_diagram\_process}(\(D\), \texttt{proc})

on diagram \(D\). The users only have to provide a procedure \texttt{proc} as a parameter to tell \texttt{REDLIB}
how to process a typical node in the diagram. The users should implement \texttt{proc} with the procedures discussed in this section to manipulate the structures in $D$. Explanation of and examples for the procedures in the second way can be found in subsection ??.

9.1 Basic constraint construction

\textsc{REDLIB} supports the following procedures to manipulate diagrams.

\begin{verbatim}
/* Procedure that returns Boolean true. */
redgram *red_true()

This procedure returns a diagram for Boolean \texttt{true}. ■

/* Procedure that returns Boolean false. */
redgram *red_false()

This procedure returns a diagram for Boolean \texttt{false}. ■

redgram *red_diagram(F, ...)
    char *F; /* a global constraint string or an event constraint string. */
    ...; /* a sequence of arguments of either type string or type integer.*/

    The procedure returns a BDD-like diagram for constraint format string $F$ with variable
    numbers of arguments. The format string is like a constraint except that there could be
    place-holder strings like \texttt{"%s"} and \texttt{"%d"} for strings and integers respectively. The format
    string $F$ is like those in procedure \texttt{printf()}. The procedure substitutes the $i$'th arguments
    in the variable-length argument list for the $i$'th place-holder strings in $F$. ■

redgram *red_diagram_local(F, pi, ...)
    char *F; /* a global constraint string or an event constraint string. */
    int pi; /* a process index */
    ...; /* a sequence of arguments of either type string or type integer.*/

    The procedure returns a BDD-like diagram for constraint string $F$ interpreted with respect
    to process $pi$. The format string is like a constraint except that there could be place-holder
    strings like \texttt{"%s"} and \texttt{"%d"} for strings and integers respectively. The format string $F$ is
    like those in procedure \texttt{printf()}. The procedure substitutes the $i$'th arguments in the
    variable-length argument list for the $i$'th place-holder strings in $F$. ■

Note that unlike other package for BDD diagrams that supports procedure to construct

diagrams for atomic propositions, \texttt{red_diagram()} and \texttt{red_diagram_local()} are powerful in

48
constructing diagrams complicate constraints. The parameters to the two procedures can even contain linear-hybrid constraints.

9.2 Inductive constraint construction

**REDLIB** also supports construction of diagrams from other diagrams. The following three are the regular Boolean ones.

```plaintext
redgram *red_and(D1, D2)
    redgram *D1, *D2; /* Diagrams for the two conjuncts */
    The procedure returns a BDD-like diagram for the conjunction of D1 and D2. ■

redgram *red_or(D1, D2)
    redgram *D1, *D2; /* Diagrams for the two disjuncts */
    The procedure returns a BDD-like diagram for the disjunction of D1 and D2. ■

redgram *red_not(D)
    redgram *D; /* Diagrams to be complemented */
    The procedure returns a BDD-like diagram for the complement of D. ■

**REDLIB** also supports flexible quantification on a diagram. For example, given a diagram D with a local variable x[1] and a global variable y, if we want to calculate the constraint for the following expression.

\[ \exists x[1](x[1] \leq 5 \land \forall y(y < 3 \Rightarrow D)) \]

we may use the following procedure call.

```plaintext
red_quantify(D, "exists x[1], (x[1] \leq 5)\&\& \forall y, (y < 3)\Rightarrow\")
```

**REDLIB** also supports flexible parameterized invocation. For example, we may rewrite the same procedure-call as follows.

```plaintext
red_quantify(D, "exists %s, (%s[%1d] \leq 5)\&\& \forall %s, (%s<3)\Rightarrow\", "x[1]", "x", 1, "y", "y"
```

Procedure **red_quantify()** is described as follows.

```plaintext
redgram *red_quantify(D, K, ...)
    redgram *D; /* Diagrams to be restricted */
    char *K; /* a string of quantifications with optional restrictions. */
    ...; /* a sequence of arguments of either type string or type integer. */
```

49
The procedure returns a diagram by applying the quantification operations specified with argument $K$ and the variable-length arguments to diagram $D$. After the substitutions of the variable-length arguments to $K$, the syntax of $K$ should be as follows.

$$
K ::= S K | S \\
S ::= Q V ', (' F ')') P | Q V ', '~' | Q V ', '
$$

Here $K$ can be a non-empty sequence of structure $S$. There are three alternative structures for $S$. All the three alternatives start with a quantification operator, either ‘exists’ or ‘forall’. We can associate a quantification with a Boolean restriction. The first structure of $S$

$$
Q V ', (' F ')') P
$$

allows us to associate the quantification with a Boolean restriction of operator $P$ and formula $F$. $P$ can be a Boolean AND (i.e., ‘&’) or a Boolean implication (i.e., ‘=>’). $F$ is an for a general constraint. The syntax of $F$ can be found in page 6.

The second structure of $S$

$$
Q V ', '~'
$$

allows us to negate the formula to be quantified. ‘~’ is the complementation operator.

The third structure $S$

$$
Q V ', '
$$

does not use any Boolean restriction.

9.3 Normalization

CRD and HRD do not have a natural canonical form. REDLIB supports several algorithms to normalize the diagrams.

```c
/* Procedure for customized diagram manipulation with single diagram parameter */
redgram *red_norm(D, op)
    redgram *D//; the diagram to be normalized.
    int    op//; option for the normalization

This procedure returns a normalized diagram that is equivalent to $D$ according to option $op$. The choices of $op$ are the macro constants defined in redlib.h and listed in table 9.
```

9.4 Abstraction
<table>
<thead>
<tr>
<th>Macro constants</th>
<th>Options for normalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED_NORM_TIGHT</td>
<td>All-pair-shortest-form for timed automata.</td>
</tr>
<tr>
<td></td>
<td>One-pass transitive deduction for linear-hybrid automata.</td>
</tr>
<tr>
<td>RED_NORM_MAG_REDUCE</td>
<td>Using two magnitude constraints to subsume any other constraint.</td>
</tr>
</tbody>
</table>

Table 9: Option values for diagram normalization

```c
/* Procedure for untimed reduction */
redgram *red_abstract(
    redgram D,
    int flag_state_approx,
    char *role_spec // a string for role specification. +
    ... /* a sequence of arguments of either type string or type integer.*/
)
```

This procedure flexibly returns an abstract representation of diagram $D$ according to the description in `flag_state_approx` according to the role specification in string `role_spec`. 

**REDLIB** allows the users to make adaptive abstraction decision on each variable according to the class of the owner process of the variable. Variables are partitioned into four classes.

- the **global** class: The variable is declared global and belongs to no processes.
- the **model** class: The variable is declared local to a process in the model class.
- the **specification** class: The variable is declared local to a process in the specification class.
- the **environment** class: The variable is declared local to a process in the environment class.

This argument is composed of four flag values for the abstraction techniques to be used respectively for variables in these four classes. For the model class variables, we have the following flag values.

- **RED_NOAPPROX_MODL_GAME**: With this flag value, **REDLIB** makes no effort to remove any constraints with variables in the model class.
- **RED_OAPPROX_MODL_GAME_DIAG_MAG**: This flag value is only used with linear hybrid automatas. It means that for a dense variable in the model class, we keep all its magnitude\(^1\) constraints. We also keep every of its diagonal\(^2\) constraints if the other

---

\(^1\)A magnitude constraint is of the form $ax \sim c$ where $a$ is an integer constant, $x$ a dense variable, $\sim$ an inequality operator, and $c$ an integer constant.

\(^2\)A magnitude constraint is of the form $ax + by \sim c$ where $a, b$ are integer constants, $x, y$ dense variables, $\sim$ an inequality operator, and $c$ an integer constant.
variable in the diagonal constraint is not to be abstracted according to the flag values.

- **RED_OAPPROX_MODL_GAME_DIAGNAL**: This flag value tells the procedure to eliminate all magnitude constraint of clock variables in the model class.

- **RED_OAPPROX_MODL_GAME_MAGNITUDE**: This flag value tells the procedure to eliminate all diagonal constraint of clock variables in the model class.

- **RED_OAPPROX_MODL_GAME_UNTIMED**: This flag value tells the procedure to remove all clock constraints for clock variables in the model class.

- **RED_OAPPROX_MODL_GAME_MODE_ONLY**: This flag value tells the procedure to remove all local variables, except the mode variable, in the model class.

- **RED_OAPPROX_MODL_GAME_NONE**: This flag value tells the procedure to remove all local variables in the model class.

The flag values for variables in the specification class are **RED_NOAPPROX_SPEC_GAME**, **RED_OAPPROX_SPEC_GAME_DIAG_MAG**, **RED_OAPPROX_SPEC_GAME_DIAGNAL**, **RED_OAPPROX_SPEC_GAME_MAGNITUDE**, **RED_OAPPROX_SPEC_GAME_UNTIMED**, **RED_OAPPROX_SPEC_GAME_MODE_ONLY**, and **RED_OAPPROX_SPEC_GAME_NONE**. Their meanings are similar to the ones for the model-class variables.

The flag values for variables in the environment class are **RED_NOAPPROX_ENVR_GAME**, **RED_OAPPROX_ENVR_GAME_DIAG_MAG**, **RED_OAPPROX_ENVR_GAME_DIAGNAL**, **RED_OAPPROX_ENVR_GAME_MAGNITUDE**, **RED_OAPPROX_ENVR_GAME_UNTIMED**, **RED_OAPPROX_ENVR_GAME_MODE_ONLY**, and **RED_OAPPROX_ENVR_GAME_NONE**. Their meanings are similar to the ones for the model-class variables.

The flag values for variables in the global class are **RED_NOAPPROX_GLOBAL_GAME**, **RED_OAPPROX_GLOBAL_GAME_DIAG_MAG**, **RED_OAPPROX_GLOBAL_GAME_DIAGNAL**, **RED_OAPPROX_GLOBAL_GAME_MAGNITUDE**, **RED_OAPPROX_GLOBAL_GAME_UNTIMED**, **RED_OAPPROX_GLOBAL_GAME_MODE_ONLY**, and **RED_OAPPROX_GLOBAL_GAME_NONE**. Their meanings are similar to the ones for the model-class variables.

To specify an appropriate combination of the flags for the classes, we use bitwise disjunction of the flag values for the four variable classes.

A role specification string \( R \) is a string of the following form.

\[ "m_1, m_2, \ldots, m_k; s_1, s_2, \ldots, s_j;" \]

Here \( m_1, m_2, \ldots, m_k \), \( s_1, s_2, \ldots, s_j \) are process indices. We require that the sets of \( \{m_1, m_2, \ldots, m_k\} \) and \( \{s_1, s_2, \ldots, s_j\} \) are disjoint. Suppose that the process count is \( M \). a role specification string tells REDLIB that the model automaton is constructed of processes with indices in

\[ \{p \mid 1 \leq p \leq M, \land_{1 \leq i \leq j} p \neq s_i\} \]

while the specification automaton is constructed of processes with indices in

\[ \{p \mid 1 \leq p \leq M, \land_{1 \leq i \leq k} p \neq m_i\} \]

The format string is like a role specification string except that there could be place-holder
strings like “%s” and “%d” for strings and integers respectively. The format string $R$ is like those in procedure `printf()`. The procedure substitutes the $i$’th arguments in the variable-length argument list for the $i$’th place-holder strings in $F$.

**Example 19** The argument value of

```
RED_OAPPROX_MODL_GAME_DIAGONAL
| RED_OAPPROX_SPEC_GAME_MAGNITUDE
| RED_OAPPROX_ENVR_GAME_DIAGONAL
| RED_OAPPROX_GLOBAL_GAME_NONE
```

says that all magnitude constraints about either the model-class or the environment-class clock variables are to be removed, all diagonal constraints about the specification-class clock variables are to be removed, and all constraints about global variables are also to be removed. If a flag is not set for a class, its default value is no abstraction. For the explanation of the following procedures, we have the following terms. A constraint of dense variables is called magnitude if it is of the form $x \sim c$, where $c$ is a number constant and $\sim \in \{<,\leq,=,\neq,\geq,>,\}$.

A constraint is called diagonal if it is of the form $x - y \sim c$.

### 9.5 Reduction

At this moment, REDLIB supports several reduction techniques. Some techniques are automatically carried out. An example is the inactive variable elimination. The reduction procedures may lower the precision of the state-space representation. There are several reduction techniques discussed in subsection 1. However, only the symmetry reduction and the inactive variable elimination reduction can be invoked by the users. The others are always in effect.

```c
/* Procedure for process-oriented symmetry reduction */
redgram *red_symmetry(D)
    redgram *D; // the diagram to be reduced.
    This procedure returns a reduced diagram that is symmetry-equivalent to $D$ based on the process-oriented symmetry reduction techniques by Emerson, Sistla, et al.

/* Procedure for inactive variable elimination reduction */
redgram *red_reduc_inactive(D)
    redgram *D; // the diagram to be reduced.
    This procedure returns a reduced diagram out of $D$ by eliminating all recordings of inactive variables.
```
10 Precondition & postcondition constructions

We partition procedures discussed in this section into two classes. One is for the precondition and postcondition calculation of discrete transitions and time-progress. The other is for high-level procedures that calculate a large-step of reachability analysis or verification tasks. The procedures of the second class are built on those of the first class.

10.1 Preconditions & postconditions of time progress

We have the following procedures to support precondition and post-condition calculation.

- `redgram *red_time_bck(D_1, D_2)`
  `redgram *D_1, /* a diagram for the progress path */`
  `*D_1; /* a diagram for the post-condition */`
  This procedure returns a BDD-like diagram for the weakest precondition due to time progress in the dense time domain. $D_1$ specifies the space for the time-progress. $D_2$ specifies the space for the destination states of the time-progress.

- `redgram *red_time_fwd(D_1, D_2)`
  `redgram *D_1, /* a diagram for the progress path */`
  `*D_1; /* a diagram for the precondition */`
  This procedure returns a BDD-like diagram for the weakest post-condition due to time progress in the dense time domain. $D_1$ specifies the space for the time-progress. $D_2$ specifies the space for the starting states of the time-progress.

10.2 Preconditions & postconditions out of a declared model

REDLIB supports the precondition and post-condition calculation with the transitions declared in a model.

- `redgram *red_xtion_bck(D, pi, xi)`
  `redgram *D; /* a diagram for the post-condition */`
  `int pi, /* a process index */`
  `xi; /* a declared transition index */`
  The procedure returns a BDD-like diagram for the weakest precondition for states that go to a state in $D$ through executing declared transition $xi$ by process $pi$. Note that declared transition $xi$ may not be executable by itself since it may has some uncorresonded synchronization primitives. Also, process $pi$ may not be able to execute declared transition $xi$. The procedure constructs the diagram regardless of the two just-mentioned cases.
redgram *red_xtion_fwd(D, pi, xi)
  redgram *D; /* a diagram for the precondition */
  int pi, /* a process index */
  xi; /* a declared transition index */
  The procedure returns a BDD-like diagram for the weakest post-condition for states that come from a state in $D$ through executing declared transition $xi$ by process $pi$. Note that declared transition $xi$ may not be executable by itself since it may has some uncorresponded synchronization primitives. Also, process $pi$ may not be able to execute declared transition $xi$. The procedure constructs the diagram regardless of the two just-mentioned cases.

redgram *red_sync_xtion_bck(
  redgram D, /* a diagram for the post-condition */
  redgram P, /* a diagram for the path condition */
  int sxi, /* an index for a synchronous transition */
  int flag_game_roles, /* a flag telling who can execute. */
  int flag_time_progress, /* a flag telling to make time progress or not. */
  int flag_action_approx /* a flag for the approximation of evaluation */
)

The procedure returns a BDD-like diagram for the weakest precondition for states that go to a state in $D$ through executing synchronous transition $sxi$. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution.

- **flag_game_roles**: REDLIB allows the users to partition the processes into three classes: the *model*, the *specification*, and the *environment*. For safety analysis, risk analysis, and model-checking, the default is to let all processes be in the model class. Each process has a set of transition rules that it can execute. The argument allows the users to control the transition rules of which classes are to be executed in the reachability analysis. The argument actually consists of three flag values: RED_SIM_MODL (the same as RED_BISIM_MODL) for the model class, RED_SIM_SPEC (the same as RED_BISIM_SPEC) for the specification class, and RED_SIM_ENVR (the same as RED_BISIM_ENVR) for the environment class. To specify the transition rules of some classes are to be executed, we use bitwise disjunction to make the specifi-
cation. For example, if only the environment and the model classes are executing transition rules, then we let the argument value be

\[ \text{RED\_SIM\_MODL} \mid \text{RED\_SIM\_ENVR} \]

Such partitioning of the process classes is usually implicitly set with procedures `red_bisim_check()` and `red_sim_check()`.

- **flag_time_progress**: This argument tells the procedure whether to execute a time-progress step after each discrete-step evaluation. The two argument values are `RED\_NO\_TIME\_PROGRESS` and `RED\_TIME\_PROGRESS`.

- **flag_action_approx**: This argument tells the procedure how to abstract the predicate after each discrete-transition precondition or post-condition calculation. This is exactly the same as the argument with the same name to procedures `red_sync_xtion_string_bck()` and `red_sync_xtion_string_fwd()` in pages 60 and respectively. There are the following three values.
  - `RED\_NO\_ACTION\_APPROX`: This option says that no abstraction is used in the construction of the post-condition.
  - `RED\_ACTION\_APPROX\_NOXTIVE`: This option says that clock difference constraints, of the form \( x - y \sim c \) where \( c \) is an integer constant, will be removed from the constructed post-condition. That is, for clock constraints, only upper-bound and lower-bound constraints will be kept in the returned post-condition.
  - `RED\_ACTION\_APPROX\_UNTIMED`: This option says that all clock constraints will be removed from the returned post-condition. In other words, the returned diagram is an untimed abstraction of the true post-condition.

The number of synchronous transitions can be accessed with procedure `red_query_sync_xtion_count()`. The range of synchronous transition indices is in \([0, \text{red\_query\_sync\_xtion\_count}())\).

```c
redgram *red_sync_xtion_fwd(
    redgram D, /* a diagram for the precondition */
    redgram P, /* a diagram for the path condition */
    int sxi, /* an index for a synchronous transition */
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
```
The procedure returns a BDD-like diagram for the weakest post-condition for states that come from a state in $D$ through executing synchronous transition $sxi$. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: `flag_game_roles` and `flag_action_approx` are used exactly in the same way as the ones in procedure `red_sync_xtion_bck()` in page 55.

The number of synchronous transitions can be accessed with procedure `red_query_sync_xtion_count()`. The range of synchronous transition indices is in $[0, \text{red_query_sync_xtion_count()}]$.

```
redgram *red_sync_xtion_bulk_bck(
    redgram D, /* a diagram for the post-condition */
    redgram P, /* a diagram for the path condition */
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
```

This procedure returns a BDD-like diagram for the weakest precondition for states that go to a state in $D$ through executing some synchronous transitions recorded in `red_query_diagram_xtion_sync()`. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: `flag_game_roles` and `flag_action_approx` are used exactly in the same way as the ones in procedure `red_sync_xtion_bck()` in page 55.

```
redgram *red_sync_xtion_bulk_fwd(
    redgram D, /* a diagram for the precondition */
    redgram P, /* a diagram for the path condition */
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
```

This procedure returns a BDD-like diagram for the weakest post-condition for states that
come from a state in $D$ through executing some synchronous transitions recorded in $\text{red_query_diagram_xtion_sync_bulk()}$. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: $\text{flag_game_roles}$ and $\text{flag_action_approx}$ are used exactly in the same way as the ones in procedure $\text{red_sync_xtion_bck()}$ in page 55.

\begin{verbatim}
redgram *red_sync_xtion_bulk_given_bck(
    redgram D, /* a diagram for the post-condition */
    redgram P, /* a diagram for the path condition */
    redgram B /* a diagram for the bulk description of synchronous transitions */,
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
\end{verbatim}

This procedure returns a BDD-like diagram for the weakest precondition for states that go to a state in $D_1$ through executing some synchronous transitions recorded in $D_2$. This result of $\text{red_sync_xtion_bulk_bck}(D, P, f1, f2)$ is the same as $\text{red_sync_xtion_bulk_given_bck}(D, P \text{red_query_diagram_xtion_sync_bulk()}, f1, f2)$.

This procedure allows the users to experiment with different ways to evaluate the precondition through a bulk of synchronous transitions. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: $\text{flag_game_roles}$ and $\text{flag_action_approx}$ are used exactly in the same way as the ones in procedure $\text{red_sync_xtion_bck()}$ in page 55.

\begin{verbatim}
redgram *red_sync_xtion_bulk_given_fwd(
    redgram D, /* a diagram for the precondition */
    redgram P, /* a diagram for the path condition */
    redgram B /* a diagram for the bulk description of synchronous transitions */,
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
\end{verbatim}

58
This procedure returns a BDD-like diagram for the weakest post-condition for states that come from a state in $D_1$ through executing some synchronous transitions recorded in $D_2$. This result of \texttt{red_sync_xtion_bulk_fwd(D, P, f1, f2)} is the same as \texttt{red_sync_xtion_bulk_given_fwd(D, P red_query_diagram_xtion_sync_bulk(), f1, f2)}. This procedure allows the users to experiment with different ways to evaluate the post-condition through a bulk of synchronous transitions. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$. Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: \texttt{flag_game_roles} and \texttt{flag_action_approx} are used exactly in the same way as the ones in procedure \texttt{red_sync_xtion_bck()} in page 55.

\begin{verbatim}
redgram *red_sync_xtion_all_bck(
    redgram D, /* a diagram for the post-condition */
    redgram P, /* a diagram for the path condition */
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
\end{verbatim}

This procedure returns a BDD-like diagram for the weakest precondition for all states that go to some states in $D$ through executing some synchronous transitions constructed from the declared transitions in the behavior model. The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$. Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: \texttt{flag_game_roles} and \texttt{flag_action_approx} are used exactly in the same way as the ones in procedure \texttt{red_sync_xtion_bck()} in page 55. This procedure may perform autonomous garbage-collection.

\begin{verbatim}
redgram *red_sync_xtion_all_fwd(
    redgram D, /* a diagram for the precondition */
    redgram P, /* a diagram for the path condition */
    int flag_game_roles, /* a flag telling who can execute. */
    int flag_time_progress, /* a flag telling to make time progress or not. */
    int flag_action_approx /* a flag for the approximation of evaluation */
)
\end{verbatim}
This procedure returns a BDD-like diagram for the weakest post-condition for all states that come from some states in $D$ through executing some synchronous transitions constructed from the declared transitions in the behavior model.

The execution of the synchronous transition must happen in the state described with the path constraint diagram $P$.

Also, there are two flag values for the control of the synchronous transition execution. The two flag arguments: $flag\_game\_roles$ and $flag\_action\_approx$ are used exactly in the same way as the ones in procedure $red\_sync\_xtion\_bck()$ in page 55. This procedure may perform autonomous garbage-collection.

10.3 Flexible analysis-time precondition & post-condition calculation

```c
redgram red_sync_xtion_string_bck(D, flag\_action\_approx, sxt, ...)
redgram *D; /* a diagram for the post-condition */
int flag\_action\_approx; /* flag for action approximation */
char *sxt; /* a string for a declared transition */
...; /* a sequence of arguments of either type string or type integer */
```

The procedure returns a BDD-like diagram for the weakest precondition for states that go to a state in $D$ through executing a synchronous transition represented with string $sxt$ by abstraction option $flag\_action\_approx$. The string $sxt$ may not be with any transition declared in the model. This procedure allows the users to dynamically construct transitions and check their execution results.

The format string $sxt$ is like a sequence of pairs of process indices and transition rules. Each transition rule follows the syntax of a transition rule in $REDLIB$ file input. Formally speaking, string $sxt$ is of the following syntax.

$$sxt ::= p \; ':'; \; rule \; sxt |$$

Here $p$ is a process index. $rule$ is a string that declares a transition rule and starts with a reserved word $when$, then a triggering condition, then a reserved word $may$, and finally some actions. $sxt$ consists of zero or more pairs of process indices and rule declarations.

Suppose we have $sxt$ as a string $p_1 : r_1 p_2 : r_2 \ldots p_n : r_n$. The procedure then returns the precondition of $D$ through executing rule $r_1, \ldots, r_n$ respectively by process $p_1, \ldots, p_n$. The execution can be explained with the following pseudo-code.

For each $i = 1$ to $n$, do {
let $D$ be the precondition of $D$ through executing rule $r_i$ by process $p_i$.

Return $D \land \bigwedge_{1 \leq i \leq n} (\text{the triggering condition of } r_i \text{ for process } p_i)$.

Just like the format strings used in \texttt{printf()} and \texttt{red_diagram()}, we also allow placeholder strings like "\%s" and "\%d" for strings and integers respectively. The procedure substitutes the $i$'th arguments in the variable-length argument list for the $i$'th place-holder strings in $sxt$.

There are three options that we can use for argument \texttt{flag\_action\_approx}.

- \texttt{RED\_NO\_ACTION\_APPROX}: This option says that no abstraction is used in the construction of the precondition.
- \texttt{RED\_ACTION\_APPROX\_NOXTIVE}: This option says that clock difference constraints, of the form $x - y \sim c$ where $c$ is an integer constant, will be removed from the constructed precondition. That is, for clock constraints, only upper-bound and lower-bound constraints will be kept in the returned precondition.
- \texttt{RED\_ACTION\_APPROX\_UNTIMED}: This option says that all clock constraints will be removed from the returned precondition. In other words, the returned diagram is an untimed abstraction of the true precondition.

\begin{example}
Here we have an example of using the procedure for a model with three processes.

\begin{verbatim}
result = red_false();
for (i = 2; i <= red_process_count(); i++) {
    conj = red_sync_xtion_string_bck(
        d, RED\_NO\_ACTION\_APPROX,
        "%1d:when ?begin (active[%1d] && x<=3) may x=0; goto collision; \n        %1d:when !begin (wait[%1d]) may x=0; goto transm;",
        1, 1, i, i
    );
    result = red_or(result, conj);
}
\end{verbatim}

This piece of code calculates the precondition of two synchronous transitions of the space represented with diagram $d$. The precondition is saved in diagram $\texttt{result}$. The first synchronous transition consists of executing

\begin{verbatim}
when ?begin (active[1] && x<=3) may x=0; goto collision;
\end{verbatim}
and

\[
\text{when } !\text{begin (wait}[2]) \text{ may } x=0; \text{ goto transm};",
\]

respectively by processes 1 and 2. The second synchronous transition consists of executing

\[
\text{when } ?\text{begin (active}[1] \& \& x<=3) \text{ may } x=0; \text{ goto collision;}
\]

and

\[
\text{when } !\text{begin (wait}[3]) \text{ may } x=0; \text{ goto transm};",
\]

respectively by processes 1 and 3. The preconditions of the two synchronous transitions are unioned together and saved in variable \texttt{result}.

\begin{verbatim}
redgram red_sync_xtion_string_fwd(D, flag_action_approx, sxt, ...)
    redgram *D; /* a diagram for the post-condition */
    int flag_action_approx; /* flag for action approximation */
    char *sxt; /* a string for a declared transition */
    ...; /* a sequence of arguments of either type string or type integer.*/

    The procedure returns a BDD-like diagram for the post-condition for states that come from a state in \( D \) through executing a synchronous transition represented with string \( sxt \) by abstraction option \( flag\_action\_approx \). The string \( sxt \) may not be with any transition declared in the model. This procedure allows the users to dynamically construct transitions and check their execution results.

    The format string \( sxt \) is like a sequence of pairs of process indices and transition rules. Each transition rule follows the syntax of a transition rule in \texttt{REDLIB} file input. Formally speaking, string \( sxt \) is of the following syntax.

    \[
    sxt ::= p ':': rule sxt |
    \]

    Here \( p \) is a process index. \textit{rule} is a string that declares a transition rule and starts with a reserved word \texttt{when}, then a triggering condition, then a reserved word \texttt{may}, and finally some actions. \( sxt \) consists of zero or more pairs of process indices and rule declarations. Suppose we have \( sxt \) as a string \( p_1: r_1 p_2: r_2 \ldots p_n: r_n \). The procedure then returns the precondition of \( D \) through executing rule \( r_1, \ldots, r_n \) respectively by process \( p_1, \ldots, p_n \). The execution can be explained with the following pseudo-code.

\begin{verbatim}
Let \( D \) be \( D \land \bigwedge_{1 \leq i \leq n}(\text{the triggering condition of } r_i \text{ for process } p_i) \).
For each \( i = 1 \) to \( n \), do {
    let \( D \) be the post-condition of \( D \) through executing rule \( r_i \) by process \( p_i \).
\end{verbatim}
\end{verbatim}

62
Just like the format strings used in `printf()` and `red_diagram()`, we also allow placeholder strings like “%s” and “%d” for strings and integers respectively. The procedure substitutes the i’th arguments in the variable-length argument list for the i’th place-holder strings in `sxt`.

There are three options that we can use for argument `flag_action_approx`.

- **RED_NO_ACTION_APPROX**: This option says that no abstraction is used in the construction of the post-condition.
- **RED_ACTION_APPROX_NOXTIVE**: This option says that clock difference constraints, of the form $x - y \sim c$ where $c$ is an integer constant, will be removed from the constructed post-condition. That is, for clock constraints, only upper-bound and lower-bound constraints will be kept in the returned post-condition.
- **RED_ACTION_APPROX_UNTIMED**: This option says that all clock constraints will be removed from the returned post-condition. In other words, the returned diagram is an untimed abstraction of the true post-condition.

11 Packaged verification tasks

11.1 Reachability analysis

Reachability analysis means to construct a characterization of those states reachable in a space from or to a particular set of states with respect to a behavior model. There are two ways to do this. The first is called **forward reachability analysis** in which we construct a characterization of all states that can be reached from a set of initial states with respect to the declared behavior structure. The second is called **backward reachability analysis** in which we construct a characterization of all states that can reach some goal states with respect to the declared behavior structure. REDLIB supports on-the-fly construction of forward and backward reachability analysis with the following procedures.

```c
/* Backward reachability analysis */
struct reachable_return_type *red_reach_bck(
    redgram I, // a diagram for the initial states
    redgram P, // a diagram for the path states
    redgram G, // a diagram for the goal states
);```
int flag_task,
int flag_parametric_analysis,
int flag_game_roles,
int flag_full_reachability,
int flag_reachability_depth_bound,
int flag_counter_example,
int flag_time_progress,
int flag_normality,
int flag_action_approx,
int flag_reduction,
int flag_state_approx,
int flag_symmetry,
int flag_print
)

Here $I$, $P$, and $G$ serve as BDD-like diagrams that respectively specify the initial states, the path states, and goal states of the backward reachability analysis. This procedure returns a BDD-like diagram that characterizes the space of states satisfying $P$ and can reach some states in $G$ through a path of states satisfying $P$ with respect to the declared behavior structure.

Note that $P$ could be different from the global invariance diagram constructed with `red_query_decalred_invariance_diagram()`. The design of this procedure allows the users to experiment with strategies in abstract reachability analysis.

This procedure may perform autonomous garbage-collection.

There are sixteen flags we can set to control the construction of the backward reachabilities.

- **flag_task**: This argument tells the procedure in what verification task, this procedure is called. This argument could be used if we find a computation (or counter-example in safety/risk analysis) that makes a goal state reachable from an initial state. In case that such a computation is detected, the procedure may print out some messages in referring to the verification task.

There are the following possible values for the argument.

- **RED_TASK_SAFETY**: for safety analysis. This implies that $G$ is the negation of a safety predicate.
- **RED_TASK_RISK**: for risk analysis. This implies that $G$ is the risk predicate.
- **RED_TASK_GOAL**: for goal analysis. Then $G$ is the goal state predicate.
- **RED_TASK_ZENO**: for the reachability of Zeno states. Zeno states are those states
in a model from which no computation can lead to divergent time-progress. \( G \)
is supposedly a state predicate for Zeno states. The diagram for a subset of the Zeno states in a model can be accessed with procedure \texttt{red_query
zeno()}. Please check page 79.

- \texttt{RED\_TASK\_DEADLOCK}: for the reachability of deadlock states. Deadlock states in \texttt{REDLIB} are defined as those states in a model from which neither time can progress nor any non-trivial discrete transition can happen. The diagram for the deadlock states in a model can be accessed with procedure \texttt{red_query_deadlock()}. Please check page 79.

- \texttt{RED\_TASK\_MODEL\_CHECK}: for the model-checking of TCTL formulas. This is not expected. But it is allowed.

- \texttt{RED\_TASK\_BRANCH\_SIM\_CHECK}: for branching simulation checking. This is not expected but allowed.

- \texttt{RED\_TASK\_BRANCH\_BISIM\_CHECK}: for branching bisimulation checking. This is not expected but allowed.

- \texttt{flag\_parametric\_analysis}: This argument tells the procedure to also calculate the parameter predicate that makes the reachability happen. This is only meaningful for linear hybrid automatas. The two possible values of the argument are \texttt{RED\_PARAMETRIC\_ANALYSIS} and \texttt{RED\_NO\_PARAMETRIC\_ANALYSIS}.

- \texttt{flag\_game\_roles}: The use of this flag is exactly the same as the one for procedure \texttt{red\_sync\_xtion\_bck()} described in page 55.

- \texttt{flag\_full\_reachability}: This argument tells the procedure whether to stop when the reachability between an initial state and a goal state is detected or to pursue for the construction of the full reachable state spaces. The two argument values are \texttt{RED\_FULL\_REACHABILITY} and \texttt{RED\_NO\_FULL\_REACHABILITY}.

- \texttt{flag\_reachability\_depth\_bound}: This is an integer argument that tells the procedure to only construct the predicate only for those states that is backward reachable from a goal state in at most \texttt{flag\_reachability\_depth\_bound} timed transition steps. If \texttt{flag\_reachability\_depth\_bound} is -1, then the argument tells the procedure to do reachability without any bound on the number of timed transition steps.

- \texttt{flag\_counter\_example}: This argument tells the procedure whether to construct a counter example when a the reachability between an initial state and a goal state is detected. The two argument values are \texttt{RED\_COUNTER\_EXAMPLE} and \texttt{RED\_NO\_COUNTER\_EXAMPLE}.

- \texttt{flag\_time\_progress}: This argument tells the procedure whether to execute a time-progress step after each discrete-step evaluation. The two argument values are
\texttt{RED\_NO\_TIME\_PROGRESS} and \texttt{RED\_TIME\_PROGRESS}.

- **\texttt{flag\_normality}:** This argument tells the procedure how to normalize the zone representations. There are the following three values.
  - \texttt{RED\_NORM\_ZONE\_NONE}: No normalization at all.
  - \texttt{RED\_NORM\_ZONE\_MAGNITUDE\_REDUCED}: Normalization by trying to remove redundant clock difference constraints that can be derived from an upper-bound constraint (of either the form $x \leq c$ or the form $x < c$) and a lower-bound constraint (of either the form $x \geq c$ or the form $x > c$).
  - \texttt{RED\_NORM\_ZONE\_CLOSURE}: The tight form is used for the normal form. That is, all clock differences are tight. In other words, we cannot change lower the upper-bound on any clock constraints without changing the shapes of the zones.

- **\texttt{flag\_action\_approx}:** The use of this flag is exactly the same as the one for procedure \texttt{red\_sync\_xtion\_bck()} described in page 55.

- **\texttt{flag\_reduction}:** This argument tells the procedure whether to abstract out those inactive variables or not. A variable is inactive in a state if its value does not affect the the behavior of the model in the future. There are two possible values: \texttt{RED\_NO\_REDUCTION\_INACTIVE} (do not remove those inactive variables from a state predicate) and \texttt{RED\_REDUCTION\_INACTIVE} (remove all inactive variables from a state predicate).

- **\texttt{flag\_state\_approx}:** The explanation of this argument is the same as the one for procedure \texttt{red\_abstract()} in page 51.

- **\texttt{flag\_symmetry}:** This argument tells the procedure whether to perform any process-oriented symmetry reduction\textsuperscript{3} or not. There are four values for the argument.
  - \texttt{RED\_NO\_SYMMETRY}: Do not do process-oriented symmetry reduction.
  - \texttt{RED\_SYMMETRY\_ZONE}: Perform process-oriented symmetry reduction to normalize zone representations.
  - \texttt{RED\_SYMMETRY\_DISCRETE}: Perform process-oriented symmetry reduction to normalize discrete-space representations.
  - \texttt{RED\_SYMMETRY\_POINTER}: Perform process-oriented symmetry reduction to normalize the directed graphs constructed with the pointer variables. Note that the pointer variables in the \texttt{REDLIB} models all point to \texttt{NULL} or processes. In such a directed graph, the nodes are the processes and the arcs are the pointer-to relations.

\textsuperscript{3}Process-oriented symmetry reduction was based on the idea of Emerson and Sistla to permute the process indices to check if two state predicates are symmetric.
RED_SYMMETRY_STATE: Perform process-oriented symmetry reduction to normalize first the discrete-space then the representations of zones.

- *flag_print*: This argument tells the procedure whether to print out some messages or not. At this moment, there are only two values RED_NO_PRINT and RED_PRINT.

The return value of the procedure is a data structure of type reachable_return_type defined as follows.

```c
struct counter_example_party_type {
    int proc, xtion;
};

struct counter_example_node_type {
    int exit_sync_xtion_party_count;
    struct counter_example_party_type *exit_sync_xtion_party;
    char *exit_sync_xtion_string;
    redgram prestate;
    struct counter_example_node_type *next_counter_example_node;
};

struct reachable_return_type {
    int status,
    #define MASK_REACHABLE_RETURN (0xF)
    #define FLAG_RESULT_EARLY_REACHED 1
    #define FLAG_RESULT_FULL_FIXPOINT 2
    #define FLAG_RESULT_DEPTH_BOUND_FINISHED 4
    #define MASK_COUNTER_EXAMPLE_GENERATED (0x10)
    #define MASK_COUNTER_EXAMPLE_NOT_GENERATED (0x00)
    #define MASK_RESULT_PARAMETRIC_ANALYSIS (0x20)
    #define MASK_RESULT_NO_PARAMETRIC_ANALYSIS (0x00)
    #define MASK_REACHABILITY_RESULT (0xF00)
    #define FLAG_REACHABILITY_UNDECIDED 0
    #define FLAG_NOT_REACHABLE (0x100)
    #define FLAG_REACHABILITY_INCONCLUSIVE (0x200)
    #define FLAG_REACHABILITY_DETECTED (0x300)
    #define MASK_LFP_TASK_TYPE (0xF000)
    #define MASK_LFP_TASK_RISK (0x1000)
    #define MASK_LFP_TASK_GOAL (0x2000)
    #define MASK_LFP_TASK_SAFETY (0x3000)
    #define MASK_LFP_TASK_ZENO (0x5000)
    #define MASK_LFP_TASK_DEADLOCK (0x6000)
```
This data structure encapsulates the result of a reachability analysis, backward or forward. The data structure declaration allows the users to directly access the analysis result without having to first dump it to a file and then read it in with a parser that you have to develop. We explain the attributes of the data structure as follows.

- **status**: This attribute contains the following flags.
  - Flags indicating how the reachability analysis finished.
    - *MASK_REACHABLE_RETURN*: a mask for extracting flag values about how the reachability analysis is finished.
    - *FLAG_RESULT_EARLY_REACHED*: a flag value saying that the reachability analysis finished when the first computation that confirmed the reachability was detected.
    - *FLAG_RESULT_FULL_FIXPOINT*: a flag value saying that the reachability analysis finished when full reachability was reached.
    - *FLAG_RESULT_DEPTH_BOUND_FINISHED*: a flag value saying that the reachability finished when the bound on the number of timed transition steps set by the users is reached.
  - Flag values indicating whether a counter example (a run that shows the reachability) has been constructed.
    - *FLAG_COUNTER_EXAMPLE_GENERATED*: a counter has been constructed.
    - *FLAG_COUNTER_EXAMPLE_NOT_GENERATED*: no counter example constructed.
  - Flag values indicating whether predicate for parametric reachability has been constructed.
    - *FLAG_RESULT_PARAMETRIC_ANALYSIS*: This flag value is meaningful only for linear hybrid automatas. It shows that reachability has been confirmed with the condition on the parameters described in the diagram variable `risk_parameter`.
    - *FLAG_RESULT_NO_PARAMETRIC_ANALYSIS*: This flag value means that no predicate has been constructed for the diagram variable `risk_parameter`.
- **iteration_count**: This attribute tells us how many iterations of least fixpoint are used to calculate the reachability predicate.
• **counter_example_length**: This attribute tells us the length of the constructed counter example, if any.

• **counter_example**: This attribute is a pointer to a list with node type `counter_example_node_type`. This list represents a counter-example, i.e., a computation from an initial state to a goal state that confirms the reachability. Each node in the list represents a synchronous transition in the model, and the precondition of those states in the counter-example to the corresponding synchronous transition through time passage.

• **reachability**: This attribute is a diagram that represents the set of states reached in the reachability analysis.

• **risk_parameter**: this attribute is a diagram that represents the constraint on input parameter variables that supports the reachability. This is meaningful only for linear hybrid automatas.

This procedure may incur autonomous garbage collection. Note that the diagrams used in the return result are not automatically protected from garbage-collection. The users need to push them to stack or mark them specifically.

```c
/* Forward reachability analysis */
struct reachable_return_type *red_reach_fwd(
  redgram I, // a diagram for the initial states
  redgram P, // a diagram for the path states
  redgram G, // a diagram for the goal states
  int flag_task,
  int flag_parametric_analysis,
  int flag_game_roles,
  int flag_full_reachability,
  int flag_reachability_depth_bound,
  int flag_counter_example,
  int flag_time_progress,
  int flag_normality,
  int flag_action_approx,
  int flag_reduction,
  int flag_state_approx,
  int flag_symmetry,
  int flag_print
)
```
The interpretation of all the arguments is the same as the one for procedure `red_reach_bck()` in page 63. Specifically, the explanation of argument `flag_state_approx` is the same as the one for procedure `red_abstract()` in page 51. This procedure returns a BDD-like diagram that characterizes the space of states satisfying \( P \) and can be reached from some states in \( I \) through a path of states satisfying \( P \) with respect to the declared behavior structure.

This procedure may perform autonomous garbage-collection. Note that the diagrams used in the return result are not automatically protected from garbage-collection. The users need to push them to stack or mark them specifically.

The commonly adopted verification framework of risk analysis and safety analysis can all be fulfilled with this procedure, pending on the computing resources. Deadlock and Zeno analysis can also be fulfilled by respectively using the return values of `red_query_deadlock()` (page 79) and `red_query_zeno()` (page 79) as the goal state predicate argument \( G \).

### 11.2 Model-checking with REDLIB

```c
/* Procedure for model-checking */
struct model_check_return_type *red_model_check(
    redgram I,
    redgram P,
    int flag_normality,
    int flag_action_approx,
    int flag_reduction,
    int flag_state_approx,
    int flag_zeno,
    int flag_print,
    char *f,
    ...
)
```

This procedure can only be used for timed automata. Here \( P \) represents the space in which the model-checking is to be done. This procedure returns a structure for the model-checking result for the TCTL formula (henceforth the specification) constructed out of string \( f \) and the variable-length arguments in the space of \( P \) with respect to the declared model structure from initial state \( I \). The explanation of arguments `flag_normality`,
flag_action_approx, flag_reduction, flag_state_approx, flag_zeno, flag_print is exactly the same as that for procedure red_reach_bck() in page 63. Argument flag_zeno says that whether we should only consider those runs with divergent time values in the model-checking. There are three values of this argument.

- **RED_PLAIN_NONZENO**: this flag value means that only runs with divergent time values will be considered in the model-checking. This option may incur significant computation.
- **RED_APPROX_NONZENO**: this flag value means that REDLIB uses an abstraction technique to evaluate whether there is a divergent run from a state. This option may not yield the precision for the correct checking of inevitability properties.
- **RED_ZENO_TRACES_OK**: this flag means that REDLIB will not check whether a run is Zeno or with divergent time values. This option may not yield the precision for the correct checking of inevitability properties.

This procedure is also with variable-length arguments. Those variable-length arguments are used to fill in the place-holder values in string f. This arrangement allows the users to write parameterized code.

The procedure returns a structure of type structure model_check_return_type declared as follows.

```c
struct red_predicate_type {
    struct red_type *red, *original_red;
};

struct ps_bunit_type {
    struct ps_exp_type *subexp;
    struct ps_bunit_type *bnext;
};

struct ps_bexp_type {
    int len;
    struct ps_bunit_type *blist;
};

struct ps_rexp_type {
    char *clock_name;
    int clock_index;
    struct parse_variable_type *var;
    struct ps_exp_type *child;
};

struct ps_qexp_type {
    char *quantifier_name;
    int value;
    struct ps_exp_type *quantification, *child;
};
```
struct ps_fairness_link_type {
    int status, occ_vi;
    struct parse_variable_type *occ_var;
    struct ps_exp_type *fairness;
    struct red_type *red_fairness;
    struct ps_fairness_link_type *next_ps_fairness_link;
};

struct ps_mexp_type {
    int time_lb, time_ub,
        strong_fairness_count, weak_fairness_count;
    struct ps_exp_type *path_child, *dest_child;
    struct red_type *red_early_decision_maker;
    struct ps_fairness_link_type *strong_fairness_list, *weak_fairness_list;
};

union ps_union {
    struct red_predicate_type rpred;
    struct ps_bexp_type bexp;
    struct ps_rexp_type reset;
    struct ps_qexp_type qexp;
    struct ps_mexp_type mexp;
};

struct ps_exp_type {
    int type, /* EXISTS_UNTIL, EXISTS_ALWAYS, RED,
        * AND, OR, NOT, RESET, FORALL, EXISTS,
        */
        status,
    #define FLAG_TCTCTL_INSIDE (0x04000000)
    #define FLAG_GFP_EARLY_DECISION (0x08000000)
    lineno;
    union ps_union u;
    struct ps_exp_type *parent,
        *original_form; // For atomic formulas,
        // original_form is identical to
        // the formula itself.
        // For other formulas, this
        // is a new copy.
        // This can be used
        // for educational purpose.
        // It can also be used for
        // model-checking analysis.
    struct red_type *diagram_label;
};

struct model_check_return_type {
    int status;
    #define FLAG_MODEL_CHECK_SATISFIED 1
    #define FLAG_MODEL_CHECK_UNSATISFIED 0
    redgram initial_state_diagram, failed_state_diagram;
    struct ps_exp_type *neg_formula;
};

/
/* model_check_return_type */
Attribute \textit{status} in a structure \texttt{model\_check\_return\_type} is a set of flags. At this moment, two flag values are \texttt{FLAG\_MODEL\_CHECK\_SATISFIED} and \texttt{FLAG\_MODEL\_CHECK\_UNSATISFIED} for the analysis result of the model-checking. The former says that the model satisfies the specification while the latter says the model does not.

Attribute \textit{initial\_state\_diagram} is basically $I$. Attribute \textit{failed\_state\_diagram} is a diagram that describes those states that does not satisfy the specification. Attribute \textit{neg\_formula} is a parsing tree for the negated specification.

Each node in the parsing tree is of type structure \texttt{ps\_exp\_type}. Attribute \textit{type} specifies the nine different types of the parsing tree nodes: \texttt{EXISTS\_UNTIL}, \texttt{EXISTS\_ALWAYS}, \texttt{RED}, \texttt{AND}, \texttt{OR}, \texttt{NOT}, \texttt{RESET}, \texttt{FORALL}, and \texttt{EXISTS}. Attribute \textit{status} tells us if the subformula corresponding to the node is a TCTCTL formula, a special subclass of TCTL. Also when the corresponding subformula is an $\exists \Box$-formula and evaluated as “not satisfied”, the flag also tells us whether the analysis answer was obtained with the early decision technique on greatest fixpoint evaluation. Attribute union \texttt{u} is for the structures of the corresponding subformula. Attribute \textit{original\_form} is for the original form of the corresponding subformula. Users can print out the original form with procedure \texttt{red\_print\_ps\_exp()}. Attribute \textit{diagram\_label} is a diagram that describes the set of states that satisfy the corresponding subformula.

\begin{example}
We may want to check what is the minimum integer values of $c$ in [0, 100] that makes the following TCTL formula satisfied the declared model.

\[
\forall \exists c \quad (\text{collision} [1] \Rightarrow \forall \text{ eventually } (c \leq c \text{ idle} [1]))
\]

Note that the above formula is not allowed in \texttt{REDLIB} since $c$ is not a constant. We can write the following code for the TCTL model-checking task.
\end{example}
This procedure may incur autonomous garbage collection. Note that the diagrams used in the return result are not automatically protected from garbage-collection. Procedure `red_push(d)` pushes diagram `d` to a stack maintained by `REDLIB` so that the diagram will not be claimed by garbage-collection. It returns the stack frame index `k` for the diagram `d`. We later call procedure `red_stack(k)` to refer to the diagram. The users need to push them to stack or mark them specifically.

Procedure `red_query_diagram_initial()` returns the diagram for the initial condition declared in the model structure.

### 11.3 Simulation & bisimulation-checking with REDLIB

```c
/* Procedure for simulation-checking */
struct sim_check_return_type *red_sim_check(
    redgram I,
    redgram S,
    int flag_complete_greatest_fixpoint,
    int flag_fixpoint_iteration_bound,
    int flag_counter_example,
    int flag_time_progress,
    int flag_normality,
    int flag_action_approx,
    int flag_reduction,
    int flag_state_approx,
    int flag_symmetry,
    int flag_zeno,
    int flag_print,
    char *R,
    ...
)
```

This procedure calculates the branching simulation between two timed automata against a common environment timed automata. Specifically, `REDLIB` checks if the model timed
automata (specified in string $R$ and the variable-length arguments) is simulated by the
specification timed automata (also specified in string $R$ and the variable-length arguments).
The procedure can only be used for timed automata at the moment.
Here $S$ represents the initial description of the set of state pairs for the branching simula-
tion. $I$ represents the state pairs for initial states in the branching simulation of the two
timed automata.

There are some new flag arguments that need explanation. Flag `flag_complete_greatest_fixpoint`
tells the procedure whether to pursue complete greatest fixpoint evaluation or not in
the simulation checking. If this flag is set to `RED_NO_COMPLETE_GREATEST_FIXPOINT`,
`REDLIB` stops the greatest fixpoint evaluation if it sees that some initial states of the
model automata are not simulated by any initial states of the specification automata in the
greatest fixpoint image of the simulation relation at the present fixpoint iteration. If it is
set to `RED_COMPLETE_GREATEST_FIXPOINT`, `REDLIB` will continue the greatest fixpoint
evaluation until the greatest fixpoint image does not change with new iterations.
Flag `flag_fixpoint_iteration_bound` tells `REDLIB` a bound on the number of the
greatest point iterations to be executed. If `flag_fixpoint_iteration_bound` is set to -1,
then the argument tells the procedure to do the greatest fixpoint evaluations without any
bound on the number of iterations.
Flag `flag_counter_example` tells `REDLIB` whether to construct a counter example or
not. However the argument is at the moment neglected by `REDLIB` since we have not
implemented the counter-example capability for branching simulation yet.
Flag `flag_zeno` tells `REDLIB` whether to take Zeno computations into consideration.
This is a new feature of `REDLIB`. If this flag is set to `RED_ZENO_TRACES_OK`, the procedure
computes for the traditional branching simulation. If it is set to `RED_PLAIN_NONZENO`, then
only those states of the model timed automata that start a non-Zeno computations will
be checked for simulation by a state of the specification timed automata. If it is set to
`RED_APPROX_NONZENO`, then only those states of the model timed automata that start a run
that looks like a non-Zeno computation in the abstraction will be checked for simulation
by a state of the specification timed automata.

The explanation of the other flag arguments are the same as for the ones for procedures
`red_abstract()` in page 51, `red_reach_bck()` in page 63, and `red_model_check()` in
page 70.
$R$ is a string for the specification of roles of the processes. Its explanation is the same as
the same-name argument for procedure `red_abstract()` in page 51.
This procedure returns a structure of type `sim_check_return_type` for the analysis result
struct sim_check_return_type {
    int status;
    #define MASK_REACHABLE_RETURN (0xF)
    #define FLAG_RESULT_EARLY_REACHED 1
    #define FLAG_RESULT_FULL_FIXPOINT 2
    #define FLAG_RESULTDEPTH_BOUND_FINISHED 4
    #define FLAG_COUNTER_EXAMPLE_GENERATED (0x10)
    #define FLAG_COUNTER_EXAMPLE_NOTGENERATED (0x00)
    #define FLAG_NO_SIMULATION 0
    #define FLAG_NO_BISIMULATION 0
    #define FLAG_SIMULATION_EXISTS (0x100)
    #define FLAG_BISIMULATION_EXISTS (0x100)
    int iteration_count;
    redgram initial_state_pair_diagram,
    final_sim_relation_diagram;
    #define bisim_relation_diagram sim_relation_diagram
    redgram *iteratively_removed_diagram;
    char*temp;
}; /* sim_check_return_type */

Attribute status is a set of flags. At this moment, two flag values FLAG_NO_SIMULATION (or
FLAG_NO_BISIMULATION) and FLAG_SIMULATION_EXISTS (or FLAG_BISIMULATION_EXISTS)
tell us whether a simulation (or simulation) exists. Attribute iteration_count tells us
in how many greatest fixpoint iterations, the simulation relation is calculated. Attribute
initial_state_pair_diagram is basically I.
Attribute final_sim_relation_diagram is a diagram for the description of simulation re-
lation at the end of the greatest fixpoint evaluation. The interpretation of the this attribute
is subject to the flag values of FLAG_RESULT_EARLY_REACHED, FLAG_RESULT_FULL_FIXPOINT,
and FLAG_RESULT_DEPTH_BOUND_FINISHED.
Attribute iteratively_removed_diagram records an array of iteration_count diagrams
indexed 1, ..., iteration_count. For any \( i \in [1, \text{iteration_count}] \), the \( i \)’th element in
the array is the diagram for those state pairs removed from the greatest fixpoint image at
the \( i \)’th iteration of the fixpoint loop.
The last attribute is named temp and is not used at the moment.
This procedure may perform autonomous garbage-collection. All diagrams in the returned
structure have been protected from garbage collection with procedure red_static_protect().
Users need to call red_static_unprotect() to make them recyclable in garbage collec-

Example 22 : For the example in figure 2 and in subsection 3.2, we may have the following verification task invocation.

```c
               red_query_declared_invariance_diagram(),
               "%1d; %1d;", 2, 3);
```

The verification task is to check whether the model automaton of processes 1 and 2 is simulated by the specification automaton of processes 1 and 3. The initial greatest fixpoint image of the simulation is constructed with `red_query_declared_invariance_diagram()`, the initial conditions of the model and the specification are respectively `wait[2] && x[2]==0` and `wait[3] && x[3]==0`.

```c
/* Procedure for bisimulation-checking */
struct sim_check_return_type *red_bisim_check(
    redgram I,
    redgram S,
    int flag_complete_greatest_fixpoint,
    int flag_fixpoint_iteration_bound,
    int flag_counter_example,
    int flag_time_progress,
    int flag_normality,
    int flag_action_approx,
    int flag_reduction,
    int flag_state_approx,
    int flag_symmetry,
    int flag_zeno,
    int flag_print,
    char *R,
    ...
)
```

This procedure calculates the branching bisimulation between two timed automata against a common environment timed automata. Specifically, REDLIB checks if the model timed
automata (specified in string $R$ and the variable-length arguments) is simulated by the specification timed automata (also specified in string $R$ and the variable-length arguments). The procedure can only be used for timed automata at the moment.

Here $S$ represents the initial description of the set of state pairs for the branching simulation. $I$ represents the state pairs for initial states in the branching simulation of the two timed automatas.

The explanation of the flag arguments are the same as for the ones for procedures `red_sim_check()` in page 74.

$R$ is a string for the specification of roles of the processes. Its explanation is the same as the same-name argument for procedure `red_abstract()` in page 51.

Similar to procedure `red_sim_check()` in page 74, this procedure returns a structure of type `sim_check_return_type` for the analysis result of branching bisimulation checking.

Attribute `status` is a set of flags. At this moment, two flag values `FLAG_NO_SIMULATION` (or `FLAG_NO_BISIMULATION`) and `FLAG_SIMULATION_EXISTS` (or `FLAG_BISIMULATION_EXISTS`) tell us whether a simulation (or simulation) exists.

This procedure may perform autonomous garbage-collection. All diagrams in the returned structure have been protected from garbage collection with procedure `red_static_protect()`. Users need to call `red_static_unprotect()` to make them recyclable in garbage collection.

12 Miscellaneous operations

12.1 Special constants

```
/* Procedure that returns the upper-bound of all clock constants. */
int red_clock_oo()

This procedure returns the upper-bound of all clock timing constants used in a zone. Note that the constant returned actually is two times the maximum timing constants plus 1.
```

```
/* Procedure that returns the maximum of the enumerators all hybrid constants. */
int red_hybrid_oo()

This procedure returns the upper-bound of the enumerators of all hybrid timing constants used in a convex polyhedra. Note that the constant returned actually is two times the maximum enumerators of all hybrid constants plus 1.
```
12.2 Special diagrams

Many of the diagrams that returned here are derived from the model structure. With proper use, they could be very useful in analyzing the system behavior.

\begin{verbatim}
redgram *red_process_modes(pi)
    int pi; /* a process index */
    Procedure that returns characterization of the union of invariance constraints of those modes reachable in the control flow graph of process pi.

redgram *red_process_xtions(pi)
    int pi; /* a process index */
    Procedure that returns characterization of the union of triggering constraints of those transitions firable in the local control flow graph of process pi.

redgram *red_query_deadlock()
    Procedure that returns characterization of states that do not allow the execution of any transition in the model description.

redgram *red_query_urgent()
    Procedure that returns characterization of states that do not allow any time-progress.

redgram *red_query_zeno()
    Procedure that returns characterization of states that can only lead to Zeno computations in the model description.

redgram *red_query_nonzeno()
    Procedure that returns characterization of states that may lead to non-Zeno computations in the model description.

redgram *red_var_active(vi)
    int vi; /* a variable index */
    This procedure returns a BDD-like diagram for the constraint of the variable’s value to be active in affecting the computation. That is, if the constraint is not satisfied, then the value of the variable has no effect on the computation of the model.

redgram *red_var_inactive(vi)
    int vi; /* a variable index */
    This procedure returns a BDD-like diagram for the constraint of the variable’s value to be inactive in affecting the computation. It is simply the complement of red_var_inactive(
\end{verbatim}
12.3 Role specification

```c
void red_input_roles(R, ...)
    char *R; /* a string for role specification. */
    ...; /* a sequence of arguments of either type string or type integer. */

The procedure tells REDLIB which roles each process plays. A role specification string
is a string of the following form.

"m1, m2, ..., mk; s1, s2, ..., sj;"
Here m1, m2, ..., mk, s1, s2, ..., and sj are process indices. We require that the sets
of {m1,m2,...,mk} and {s1,s2,...,sj} are disjoint. Suppose that the process count is
M. a role specification string tells REDLIB that the model automaton is constructed of
processes with indices in

{p | 1 ≤ p ≤ M, ∩1≤i≤j p ≠ si}
while the specification automaton is constructed of processes with indices in

{p | 1 ≤ p ≤ M, ∩1≤i≤k p ≠ mi}

The format string is like a role specification string except that there could be place-holder
strings like "%s" and "%d" for strings and integers respectively. The format string R is
like those in procedure printf(). The procedure substitutes the i’th arguments in the
variable-length argument list for the i’th place-holder strings in F.

Example 23: Suppose we have a system with 5 processes and we want to calculate the reach-
ability of process pair 1 and i, with i ∈ [2, 5]. Then we can use the following piece of code to do
the job.

```
Note that the loop is executed four times for each process pair of (1,2), (1,3), (1,4), and (1,5).

12.4 Garbage collection

/* Procedure for user-invoked garbage-collection */
int red_garbage_collect(op)

int op//; option for garbage collection

This procedure reclaims all those diagram nodes and arcs that can not be referenced through those system-used diagrams and those diagrams in the stack. There are the following two values for parameter op.

#define RED_GARBAGE_SILENT 0
#define RED_GARBAGE_REPORT 1

If op is RED_GARBAGE_SILENT, it does not print out any message. If op is RED_GARBAGE_REPORT, it print out some summary memssage. The procedure returns the size in bytes of the memory used by the data-structures related to REDLIB diagrams.

/* Procedure for pushing a diagram to a stack to escape from garbage-collection. */
int red_push(D)

redgram *D//; the diagram to be pushed to the stack.

This procedure pushes diagram D to the stack so that D will not be reclaimed in garbage-collection. The procedure returns an index to the stack frame that D stays in. This index can be used in the future to access D in the stack. Also, when we want to pop D out of the stack, we also want to use the index to check whether we have pop the stack frames in a correct ordering. If the number returned is less than zero, then it shows that something was wrong in the operation.

/* Procedure for popping a diagram from a stack that protects * its content diagrams from garbage collection. */
redgram *red_pop(i)

int i; \\ index to the top frame.
This procedure pops the top frame of the stack and returns the diagram saved in the top frame. It double checks whether \( i \) is the index of the top frame. If it is not, NULL is returned.

NOTE that the argument frame index \( i \) is not for REDLIB to pop a particular frame indexed \( i \). This violates the stack operation. Still, only the stack top frame can be popped. The argument is meant for REDLIB to double check whether a pop operation respects the stack usage.

```c
/* Procedure for referencing a diagram in a stack that protects its content diagrams from garbage collection. */
redgram red_stack(int i)
    int i; \ \ \ \ index to the top frame.
    This procedure returns the diagram stored in stack frame \( i \). If \( i \) is either less than zero or greater than the index of the top frame, NULL is returned.
    The above-mentioned procedures that allows us to control the garbage collection with a stack. Such stack operations sometimes can be difficult to manipulate since we may not want to protect the diagrams according to the ordering of the stack frames. REDLIB also supports alternatives in this regard for the users’ convenience. The following procedures allows us to protect some diagrams from garbage collection without pushing them to the stack.

    /* Procedure for marking a diagram for protection from garbage collection */
    void red_protect_from_gc(redgram D)
        This procedure marks all nodes used in \( D \) as protected from garbage collection with a special bit.

    /* Another procedure for marking a diagram for protection from garbage collection */
    void red_protect_aux_from_gc(redgram D)
        This procedure marks all nodes used in \( D \) as protected from garbage collection with yet another special bit.
        Note that in the implementation, the two procedures use different bits to mark their diagrams to be protected. This gives us some flexibility when we want to have some control in protecting some diagrams while not protecting some others.
```
/* Procedure for unprotecting all diagrams from garbage collection */
void red_unprotect_all_from_gc()

This procedure marks all diagrams that were marked protected with procedure red_protect_from_gc() as now unprotected.

/* Another procedure for unprotecting all diagrams from garbage collection */
void red_unprotect_all_aux_from_gc()

This procedure marks all diagrams that were marked protected with procedure red_protect_aux_from_gc() as now unprotected.

Note that a diagram that has been protected with both red_protect_from_gc() and red_protect_aux_from_gc() can only be released to the garbage collector when it has been marked unprotected by both red_unprotect_all_from_gc() and red_unprotect_all_aux_from_gc().

12.5 Diagram string representation procedures

/* Procedure for customized diagram manipulation with single diagram parameter */
char *red_diagram_string(D)
    redgram *D; // the diagram to be translated to string.

    This procedure returns a string for the global (or event) constraint represented with D.
The string can be again input to procedure red_diagram() to construct a diagram. The space for the string is dynamically allocated by REDLIB. It is up to the users when to free the space. If NULL is returned, it means something has gone wrong.

/* Procedure for customized diagram manipulation with single diagram parameter */
char *red_initial_condition_string()

    This procedure returns a string for the initial condition declared in the model structure. If it has not been declared, then NULL is returned. The space for the string is dynamically allocated by REDLIB. It is up to the users when to free the space.

/* Procedure for customized diagram manipulation with single diagram parameter */
char *red_risk_condition_string()

    This procedure returns a string for the risk condition declared in the model structure. If it has not been declared, then NULL is returned. The space for the string is dynamically
allocated by REDLIB. It is up to the users when to free the space.

12.6 Checking, selecting, and executing process transitions in reachability graph construction

To support the construction of reachability graphs with various purposes, we have implemented the following procedures. Conceptually, we figure that in each step of the reachability graph construction, we need to know what set of synchronous transitions a user want to choose. Thus we need to let the users know what the sets of synchronous transitions are available for execution are from a set of states. The following routines let the users check and select the sets of synchronous transitions in a process-by-process way.

```c
/* Procedure for starting a session for checking, selecting, and executing process transitions for backward precondition calculation. */
redgram red_begin_sync_xtion_bulk_restriction_bck(
    redgram ddst
)

This procedure starts a session for checking, selecting, and executing process transitions in backward precondition calculation. Necessary data structures are arranged. The diagram for all the synchronous transitions is returned as the implicit bulk representation for synchronous transitions that can reach states in ddst. Note that this diagram is protected from garbage collection by REDLIB automatically. This implicit diagram is also returned to the user application program for further manipulation.

/* Procedure for starting a session for checking, selecting, and executing process transitions for forward postcondition calculation. */
redgram red_begin_sync_xtion_bulk_restriction_fwd(
    redgram dsr
)

This procedure starts a session for checking, selecting, and executing process transitions in forward postcondition calculation. Necessary data structures are arranged. The diagram for all the synchronous transitions is returned as the implicit bulk representation for synchronous transitions that be fired from states in dsr. Note that this diagram
```
is protected from garbage collection by REDLIB automatically. This implicit diagram is also returned to the user application program for further manipulation.

/* Procedure for ending a session for checking, selecting, and executing process transitions. */
void red_end_sync_xtion_bulk_restriction()

This procedure ends a session for checking, selecting, and executing process transitions. Necessary data structures are released.

/* Procedure for restricting the implicit bulk representation of all synchronous transitions. */
redgram red_restrict_sync_bulk(pi, xi)

int pi; // a process index.
int xi; // a transition index.

This procedure restricts the implicit bulk representation of synchronous transitions with transition xi to be executed by process pi. The restricted bulk representation becomes the new implicit bulk representation. Note that this diagram is protected from garbage collection by REDLIB automatically. This implicit diagram is also returned to the user application program for further manipulation.

/* This procedure is related to an implicit list that records all the process and transition pairs with game roles specified in flag_game_roles in the implicit bulk representation of synchronous transitions. For each combination of game roles, REDLIB has such a list. */
int red_first_pxpair_for_roles(int flag_game_roles)

The use of this game role flag argument is exactly the same as the one for procedure red_sync_xtion_bck() described in page 55 and specifies a set of game roles to be processed with this procedure. The procedure positions an implicit working pointer to the head of the list for role combination flag_game_roles. It returns 1 if there is such a non-null list. Otherwise, it returns 0.
* with game roles specified in flag_game_roles in
* the implicit bulk representation of synchronous transitions.
* For each combination of game roles, REDLIB has such a list.
*/

int red_next_pxpair_for_roles(int flag_game_roles)

The use of this game role flag is exactly the same as the one for procedure
red_first_pxpair_for_roles() described in page 85 and specifies a set of game roles to
be process with this procedure. The procedure advances an implicit working pointer by
one link in the list for role combination flag_game_roles. It returns 0 if no more pointer
advancement is possible. Otherwise, it returns 1.

/* This procedure is related to an implicit list
* that records all the process and transition pairs
* with game roles specified in flag_game_roles in
* the implicit bulk representation of synchronous transitions.
* For each combination of game roles, REDLIB has such a list.
*/

int red_query_current_pi_for_roles(int flag_game_roles)

The use of this game role flag is exactly the same as the one for procedure
red_first_pxpair_for_roles() described in page 85 and specifies a set of game roles to
be process with this procedure. The procedure returns the process index of the process
transition pair pointed to by the implicit position pointer for role combination flag_game_roles.
If the implicit position pointer for the list for role combination flag_game_roles is already
NULL, then RED_FLAG_UNKNOWN is returned.

/* This procedure is related to an implicit list
* that records all the process and transition pairs
* with game roles specified in flag_game_roles in
* the implicit bulk representation of synchronous transitions.
* For each combination of game roles, REDLIB has such a list.
*/

int red_query_current_xi_for_roles(int flag_game_roles)

The use of this game role flag is exactly the same as the one for procedure
red_first_pxpair_for_roles() described in page 85 and specifies a set of game roles to
be process with this procedure. The procedure returns the transition index of the
process transition pair pointed to by the implicit position pointer for role combination
flag_game_roles. If the implicit position pointer for the list for role combination flag_game_roles is already NULL, then RED_FLAG_UNKNOWN is returned.

/* This procedure is related to an implicit bulk representation for synchronous transitions manipulated with the other procedures in this subsection. */

redgram red_execute_sync_bulk_restriction(
    redgram dpath,
    int flag_game_roles,
    int flag_time_progress,
    int flag_action_approx
)

If the whole session for checking, selecting, and executing process transitions are started with red_begin_sync_xtion_bulk_restriction_bck(), then this procedure sends the arguments, together with the implicit bulk representation for synchronous transitions, to routine red_sync_xtion_given_bulk_bck() to evaluate the (timed) backward precondition. Otherwise, it sends the arguments, together with the implicit bulk representation for synchronous transitions, to routine red_sync_xtion_given_bulk_fwd() to evaluate the (timed) forward precondition.

To show how we can use the procedures presented in this subsection for a solution to construct successive post-conditions with the synchronous transitions selected by the users, we have the following example of program code. The procedure uses processes 2 and 3 as the model processes, process 4 as the specification, and the others as the environment. The first command-line argument is used to declare the process count, the second to declare the number of steps to execute in path construction, and the third for the input model file name. In each iteration, it asks for the users to select some model processes and transitions to be executed by the model processes. Then the program evaluate the next state condition diagram.

/**************************
* This program accepts command line as follows.
* 
*    pathex process_count step_count model_input_file_name
* /
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <float.h>

#include "redlib.h"
#include "redlib.e"

main(argc, argv)
int argc;
char **argv;
{
    int process_count, step_count, i, flag, pi, xi;
    redgram pre;
    FILE *inputfile;

    if (argc < 4) {
        printf("Input file for successive execution not specified!\n");
        exit(0);
    }
    process_count = atoi(argv[1]); // the process count
    if (process_count < 3) {
        fprintf(RED_OUT, "No this example needs at least 6 processes. \n\n");
        exit(0);
    }
    step_count = atoi(argv[2]);
    red_begin_session(RED_SYSTEM_TIMED, argv[3], process_count);

    red_input_model(argv[3]); // argv[3] is the name of the input model file.
    fprintf(RED_OUT, "\nAfter the input model.\n");
    red_input_roles("2,3;4;"); // processes 1,2,3 are the model, 4 is the spec,
    // and the others are the environment.
    fprintf(RED_OUT, "\nAfter the input roles.\n");
    for (i = 0, pre = red_query_diagram_initial();
        i < step_count || pre == red_false();
        i++) {
        printf("\n** step %1d **\nfrom:\n%s\n",
            i, red_diagram_string(pre));
        red_begin_sync_xtion_bulk_restriction_fwd(pre);
        for (pi = 1; pi > 0; ) {
            printf("Now you have the following choices:\n");
            for (flag = red_first_pxpair_for_roles(RED_SIM_MODL);
                flag;
                flag = red_next_pxpair_for_roles(RED_SIM_MODL)) {
                printf("pi:%1d, xi:%1d, %s\n",
                    red_query_current_pi_for_roles(RED_SIM_MODL),
                    red_query_current_xi_for_roles(RED_SIM_MODL),
                    red_query_string_xtion(
                        red_query_current_xi_for_roles(RED_SIM_MODL),
                        red_query_current_pi_for_roles(RED_SIM_MODL))
                );
            }
        }
    }
}

88
printf("\nPlease select your processes (-1 for stop): ");
scanf("%d", &pi);
if (pi <= 0)
    break;
printf("Please select your transition (0 for no-op): ");
scanf("%d", &xi);
red_restrict_sync_bulk(pi, xi);

pre = red_execute_sync_bulk_restriction(
    red_query_diagram_global_invariance(),
    RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
    RED_TIME_PROGRESS,
    RED_NORM_ZONE_CLOSURE,
    RED_NO_ACTION_APPROX,
    RED_REDUCTION_INACTIVE,
    RED_OAPPROX_MODL_GAME_DIAG_MAG |
    RED_OAPPROX_SPEC_GAME_DIAG_MAG |
    RED_OAPPROX_ENVR_GAME_DIAG_MAG |
    RED_OAPPROX_GLOBAL_GAME_DIAG_MAG,
    RED_NO_SYMMETRY
);
red_end_sync_xtion_bulk_restriction();
red_end_session();
fclose(inputfile);

/* main() */

12.7 Diagram profiling

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_diagram_node_count(D)
    redgram *D; // the diagram to be analyzed.

    This procedure returns the number of nodes in D.

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_diagram_arc_count(D)
    redgram *D; // the diagram to be analyzed.

    This procedure returns the number of arcs in D.

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_diagram_size(D)
    redgram *D; // the diagram to be analyzed.

    This procedure returns the number of nodes and arcs in D.
/* Procedure for customized diagram manipulation with single diagram parameter */
int red_diagram_path_count(D)
    redgram *D; // the diagram to be analyzed.
    This procedure returns the number of root-to-terminal paths in D.

12.8 Session run-time profiling

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_cpu_time()
    This procedure returns the CPU time in seconds used by the current REDLIB session.

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_system_time()
    This procedure returns the system time in seconds used by the current REDLIB session.

/* Procedure for customized diagram manipulation with single diagram parameter */
int red_space()
    This procedure returns the number of bytes used by the current REDLIB session.

12.9 Print-out procedures

/* Procedure for printing out information of varaibles */
void red_print_variables()
    This procedure prints out the variable table in the behavior model for the users’ reference.

/* Procedure for printing out a variable */
void red_print_variable(vi)
    This procedure prints out the table information for variable vi in the behavior model for the users’ reference.
/* Procedure for printing out declared transitions */
void red_print_xtions()

This procedure prints out the transition table in the behavior model for the users’ reference.

/* Procedure for printing out a declared transition */
void red_print_xtion(xi)

This procedure prints out the table information for declared transition $xi$ in the behavior model for the users’ reference.

/* Procedure for printing out synchronous transitions */
void red_print_sync_xtions()

This procedure prints out the synchronous transition table in the behavior model for the users’ reference.

/* Procedure for printing out a synchronous transition */
void red_print_sync_xtion(sxi)

This procedure prints out the table information for synchronous transition $sxi$ in the behavior model for the users’ reference.

/* Procedure for printing out information of declared modes */
void red_print_modes()

This procedure prints out the mode table in the behavior model for the users’ reference.

/* Procedure for printing out information of a mode */
void red_print_mode(mi)

This procedure prints out the table information for mode $mi$ in the behavior model for the users’ reference.

/* Procedure for printing out some summary information of the current session */
void red_print_summary()
This procedure prints out the CPU time, system time, and the memory usage of the current REDLIB session.

/* Procedure for printing out a diagram as a tree */
void red_print_graph(D)

This procedure prints out diagram D in a tree structure that shows the sharing in the diagram.

/* Procedure for printing out a diagram as a global (or event) constraint in a line */
void red_print_line(D)

This procedure prints out diagram D as a global (or event) constraint in a single line with parentheses.

/* Procedure for making a comment in the model structure file. */
red_comment(com)

char *com; // a string for the comment.

This procedure can only be used in the model construction after the invocation of procedure red_begin_declaration() and before that of red_end_declaration(). It prints out a comment line to the model file constructed by REDLIB.

13 Examples of using REDLIB

To show how REDLIB may be used for various applications, we present two examples.

13.1 Precondition & postcondition construction of untimed complex program

Remember that the statements of redlib transition rules can be composed of five types of statement structures.

- Atomic statements like "x=y+3z+5;" where x, y, z are discrete variables.
- Concatenation statements like S1 S2 where S1 and S2 are some statement structures.
- While-loops like "while (E) S" where E is a formula and S is a statement structure.
- If-then and If-then-else statements like "if (E) S1" or "if (E) S1 else S2."
- Parenthesized block statement like "{ S }" where S is a statement structure.
With these five types of statement structures, we can use REDLIB to conveniently construct preconditions and postconditions of complex program structures.

13.2 Sudoku solver

Sudoku is a popular game with one player. According to the dkm software webpage (http://www.dkmsoftware.com/sudoku/), a Sudoku game is as follows.

Sudoku is a puzzle with a grid containing nine large blocks. Each block is divided into its own matrix of nine cells. The rules for solving Sudoku puzzles are very simple: each row, column and block must contain one of the numbers from “1” to “9”. No number may appear more than once in any row, column, or block. When you have filled the entire grid, the puzzle is solved.

In the following, we present a program that solves a Sudoku game with REDLIB. Note the program is purely for showing a typical application of REDLIB. REDLIB is considered a heavy-weight library that aims at performance for dense-time system verification. Thus the performance of this example program may not be comparable with implementations with those solvers specifically designed for propositional logics.

Anyway, here is the program. This program takes one parameter as the file name for input Sudoku games. The program reads in 81 digits to make a game. After a game is solved, the program then reads in another 81 digits for another game. The loop continues until an asterisk is read. In fact, anytime when the program reads an asterisk, it exits.

We declare night processes and night local discrete variables d1, d2, d3, d4, d5, d6, d7, d8, and d9. Thus for any \(i, j \in [1, 9]\), \(d[i][j]\) records the value of cell \(i, j\).

```c
#include <stdio.h>
#include <stdlib.h>
#include "redlib.h"
#include "redlib.e"

char *s[9][9];
int count_print = 0;

struct red_type *slot_constraint(d, i, j)
struct red_type *d;
int i, j;
{
    int value, h, k, nc, ac, gs;
    redgram conj;
```
for (value = 1; value <= 9; value++) {
for (h = 0; h < 9; h++) {
    if (h != i) {
        conj = red_diagram("not(\%s==\%d &\%s==\%d)",
                        s[i][j], value, s[h][j], value);
        if (count_print < 100 && gs < 30) {
            fprintf(RED_OUT, "\nMutual exclusion to value:\%id at \%s and \%s:\nconstraint:\n",
                     value, s[i][j], s[h][j]);
            red_print_line(conj);
            fprintf(RED_OUT, "\n");
            red_print_diagram(conj);
            fprintf(RED_OUT, "input diagram:\n");
            red_print_line(d);
            fprintf(RED_OUT, "\n");
            red_print_diagram(d);
        }
        d = red_and(d, conj);
    }
    if (h/3 == i/3) {
        for (k = 0; k < 9; k++) {
            if ((k/3) == (j/3) && (k != j)) {
                d = red_and(d,
                            red_diagram("~(\%s==\%d &\%s==\%d)",
                            s[i][j], value, s[h][k], value));
            }
        }
    }
    if ((h/3) == (i/3)) {
        for (k = 0; k < 9; k++) {
            if ((k/3) == (j/3) && (k != j)) {
                d = red_and(d,
                            red_diagram("~(\%s==\%d &\%s==\%d)",
                            s[i][j], value, s[h][k], value));
            }
        }
    }
    if (h != j) {
        d = red_and(d,
                    red_diagram("~(\%s==\%d &\%s==\%d)",
                    s[i][j], value, s[h][j], value));
    }
}
}
}
/*
   fprintf(RED_OUT, "%s && %s, after one conjunction.\n", s[i][j], s[i][h]);
   red_print_graph(d);
   fprintf(RED_OUT, "\n");
   fflush(RED_OUT);
*/
}
}
}

h = red_push(d);
red_garbage_collect(RED_GARBAGE_SILENT);
red_pop(h);
return(d);
} /* slot_constraint() */

#define FLAG_MULTIPLE 1
#define FLAG_SINGLE 0

main(argc, argv)
int argc;
char **argv;
{
    char a, *start, *condition, *p;
    redgram sol, cube;
    int i, j, k, u, v, h, m, c[9][9], lb[9][9], ub[9][9], flag;
    char stop[30];
    FILE *datafile;

    if (argc < 2) {
        printf("Input file for sudoko not specified!\n");
        exit(0);
    }
    datafile = fopen(argv[1], "r");
    if (datafile == NULL) {
        printf("Can’t open the data file of sudoko!! \n");
        exit(0);
    }

    //RED_input_file("sample.d");
    red_begin_session(RED_SYSTEM_UNTIMED, "sudoku", 2);
    red_begin_declaration(); /* each process for a row on the board. */

    for (i = 0; i < 9; i++) {
for (j = 0; j < 9; j++) {
    s[i][j] = malloc(4);
    sprintf(s[i][j], "d%i%d", i, j);
    red_declare_variable(RED_TYPE_DISCRETE, 1, 9, "%s", s[i][j]);
}

red_end_declaration();

red_print_variables();
red_print_xtions();
red_verify();

/* Declaration of the 81 slot variable indices as a 9X9 arrays.
 * Note this is not for the array of slot variable values.
 * This is only for the slot variable indices.
 */
do {
    /* After the completion of the loop, sol should be the MBDD+CRD
     * that records the solutions to your input board specification.
     * Initially, we set sol to TRUE.
     */
    sol = red_true();
    /* Please fill in the code here to calculate the solution for
     * the given suokdo puzzle.
     * The details of the main processing body, that you should fill in,
     * include:
     * 1. Read in the values of those given slots on the board and
        construct the logic formula with the REDLIB procedures.
     * 2. Iteratively restrict the formula you got in step 1 with
        all the suokdo rules.
     */
    for (i = 0; i < 9; i++) {
        for (j = 0; j < 9; j++) { 
            a = getc(datafile);
            if (a == '
')
                a = getc(datafile);
            sprintf(stop, "%c\0", a);
            c[i][j] = atoi(stop);
            if (c[i][j] != 0) {
                sol = red_and(sol, red_diagram("%s==%d", s[i][j], c[i][j]));
                sol = slot_constraint(sol, i, j);
            }
        }
    }
}
for(j = 0; j < 9; j++) {
    if (c[i][j] == 0)
        sol = slot_constraint(sol, i, j);
}
/*
   -------------------------------
   After the main processing body of the loop,
   * we are now ready to print out the solution board specification.
   * We do this in two ways.
   * In the first way, we print the board as a 9X9 char array.
   * We do this by first invoking the depth-first traversing procedure
   * red_process_DFS() for MBDD+CRD.
   * For each node in the MBDD+CRD, red_process_DFS() process the
   * node with procedure max_rec() and each arc of the node with
   * procedure arc_noop().
   * In the second way, we print the MBDD+CRD of sol as a single-line
   * Boolean formula.
   */

red_print_diagram(sol);
/*
i = red_diagram_discrete_model_count(sol);

switch (i) {
    case 0:
        fprintf(RED_OUT, "\nNo solutions!\n");
        break;
    case 1:
        fprintf(RED_OUT, "\nOnly 1 solution\n");
        break;
    default:
        fprintf(RED_OUT, "\nTotal %1d solutions!\n", i);
        break;
}
cube = red_first_cube(sol);
for (; cube != red_false(); cube = red_next_cube(sol)) {
    fprintf(RED_OUT, "\n\nOne solution:\n");
    flag = FLAG_SINGLE;
    for (i = 0; i < 9; i++) {
        for (j = 0; j < 9; j++) {
            red_get_cube_discrete_value(
                cube, s[i][j], &(lb[i][j]), &(ub[i][j])
            );
            if (lb[i][j] < ub[i][j])
                flag = FLAG_MULTIPLE;
            fprintf(RED_OUT, "%1d", lb[i][j]);
        }
    }
}
13.3 A safety analyzer

In the following, we have a safety analyzer implemented with REDLIB. The program expects up to 4 parameters. The first parameter specifies a file for model structure in RED format. The second parameter is a number. If it is specified, then it is used as the number of processes. Otherwise, we use the process number specified in the input model file. The third parameter is a string for a global constraint for the initial condition. The fourth is a string for a global constraint for the safety condition.

/****************************
 * A command line accepted by the program is as follows.
 *
 *   safe in-file-name process-count initial-condition risk-condition
 */
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include "redlib.h"
#include "redlib.e"

main(argc, argv)
int argc;
char **argv;
{
    int ini, risk, cur, w, sxi, space, step;
    struct reachable_return_type *rr;

    if (argc < 2) {
        printf("Model file for safety analyzer not specified!\n");
        exit(0);
    }
    // The 2nd parameter is used for the number of processes.
    if (argc < 3 || (w = atoi(argv[2])) <= 0)
        red_begin_session(RED_SYSTEM_TIMED, "safety analyzer", -1);
    else
        red_begin_session(RED_SYSTEM_TIMED, "safety analyzer", w);
    red_input_model(argv[1]); // the 1st parameter for the input file name.
    // print out the model structure tables.
    red_print_variables();
    red_print_xtions();
    red_print_sync_xtions();
    red_print_diagram(red_query_diagram_xtion_sync_bulk());

    if (argc < 4) // use the initial condition in the input file.
        ini = red_push(red_query_diagram_initial());
    else // use the 3rd parameter for the initial condition.
        ini = red_push(red_diagram(argv[3]));

    if (argc < 5) // use the risk condition in the input file.
        risk = red_push(red_query_diagram_spec_risk());
    else // use the 4th parameter for the safety condition.
        risk = red_push(red_not(red_diagram(argv[4])));

    // Call three abstraction routines to calculate the abstract reachability.
    rr = red_reach_fwd(
        red_stack(ini),
        red_query_diagram_declared_invariance(),
        red_false(), // redgram for the goal condition
        RED_TASK_RISK,
        RED_NO_PARAMETRIC_ANALYSIS,
        RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
        RED_FULL_REACHABILITY,
        -1,
        RED_NO_COUNTER_EXAMPLE,
        RED_NO_TIME_PROGRESS,
        RED_NORM_ZONE_NONE,
        RED_ACTION_APPROX_UNTIMED,
RED_REDUCTION_INACTIVE,
    RED_OAPPROX
  | RED_OAPPROX_MODL_GAME_UNTIMED
  | RED_OAPPROX_SPEC_GAME_UNTIMED
  | RED_OAPPROX_ENVR_GAME_UNTIMED
  | RED_OAPPROX_GLOBAL_GAME_UNTIMED,
    RED_NO_SYMMETRY,
    RED_NO_PRINT
);
space = red_push(rr->reachability);
fprintf(RED_OUT, "\nAfter untimed reachability:\n")
red_print_diagram(red_stack(space));

rr = red_reach_fwd(
    red_stack(ini),
    red_stack(space),
    red_false(), // redgram for the goal condition
    RED_TASK_RISK,
    RED_NO_PARAMETRIC_ANALYSIS,
    RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
    RED_FULL_REACHABILITY,
    -1,
    RED_NO_COUNTER_EXAMPLE,
    RED_TIME_PROGRESS,
    RED_NORM_ZONE_MAGNITUDE_REDUCED,
    RED_ACTION_APPROX_NOXTIVE,
    RED_REDUCTION_INACTIVE,
    RED_OAPPROX
  | RED_OAPPROX_MODL_GAME_MAGNITUDE
  | RED_OAPPROX_SPEC_GAME_MAGNITUDE
  | RED_OAPPROX_ENVR_GAME_MAGNITUDE
  | RED_OAPPROX_GLOBAL_GAME_MAGNITUDE,
    RED_NO_SYMMETRY,
    RED_NO_PRINT
);
red_set_stack(space, rr->reachability);
fprintf(RED_OUT, "\nAfter magnitude reachability:\n")
red_print_diagram(red_stack(space));

rr = red_reach_fwd(
    red_stack(ini),
    red_stack(space),
    red_false(), // redgram for the goal condition
    RED_TASK_RISK,
    RED_NO_PARAMETRIC_ANALYSIS,
    RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
    RED_FULL_REACHABILITY,
    -1,
    RED_NO_COUNTER_EXAMPLE,
    RED_TIME_PROGRESS,
    RED_NORM_ZONE_MAGNITUDE_REDUCED,
    RED_NO_ACTION_APPROX,
RED_REDUCTION_INACTIVE,
  RED_OAPPROX
| RED_OAPPROX_MODL_GAME_DIAG_MAG
| RED_OAPPROX_SPEC_GAME_DIAG_MAG
| RED_OAPPROX_ENVR_GAME_DIAG_MAG
| RED_OAPPROX_GLOBAL_GAME_DIAG_MAG,
  RED_NO_SYMMETRY,
  RED_NO_PRINT
);
red_set_stack(space, rr->reachability);
fprintf(RED_OUT, "\nAfter diagonal-magnitude reachability:\n\n");
red_print_diagram(red_stack(space));

// Each loop for iteration of the least fixpoint calculation.
w = red_push(red_false());
// The least fixpoint loop for the backward reachability.
for (step = 1; red_stack(w) != red_stack(risk); step++) {
  fprintf(RED_OUT, "\n****** Step %1d *******\n", step);
  red_set_stack(w, red_stack(risk));
  // Calculate the timed precondition of each indexed synchronous transition.
  for (sxi = 0; sxi < red_query_sync_xtion_count(); sxi++) {
    cur = red_push(red_sync_xtion_bck( 
      red_stack(risk),
      red_stack(space),
      sxi,
      RED_SIM_MODL | RED_SIM_SPEC | RED_SIM_ENVR,
      RED_TIME_PROGRESS,
      RED_NORM_ZONE_CLOSURE,
      RED_NO_ACTION_APPROX,
      RED_REDUCTION_INACTIVE,
      RED_NOAPPROX,
      RED_NO_SYMMETRY
    ));
    fprintf(RED_OUT, "***** After step %1d, sx %1d, a new precondition:\n", 
      step, sxi);
    red_print_diagram(red_stack(cur));
    red_set_stack(risk, red_or(red_stack(risk), red_stack(cur)));
    red_pop(cur);
  }
  // Check if the intersection with initial condition is empty.
  cur = red_push(red_and(red_stack(risk), red_stack(ini)));
  if (red_norm(red_stack(cur), RED_NOAPPROX) != red_false()) {
    printf("The system is not safe\n");
    exit(0);
  }
  red_pop(cur);
}
printf("The system is safe with the backward reachability:\n");
red_print_diagram(red_stack(risk));
red_end_session();

101
The program first reads in the model structure, the initial condition, and the risk condition. Then it calls `red_reach_un timed_fwd()`, `red_reach_magnitude_fwd()`, and `red_reach_diagonal_fwd()` to calculate an abstract representation of the forward reachability.

Then in each iteration of the outer for-loop, we calculate the timed preconditions of those preconditions to the synchronous transitions. All the calculated precondition are disjuncted with and saved in `red_stack(risk)`. In each iteration, we also check whether the newly constructed precondition intersects with the initial condition. If it does, we conclude that the system is unsafe. The loop repeats until a fixpoint is reached, i.e., when no more change to `red_stack(risk)` is possible.

When the fixpoint is reached and does not contain any initial state, we exit and conclude that the system is safe.

References


Another way to model the CSMA/CD system with API

/* (1)*/ FILE *out;
/* (2)*/ redgram *red_ini;
/* (3)*/ int ini, inv, rch;
/* (4)*/
/* (5)*/ red_begin_session(RED_SYSTEM_TIMED, "CSMA-CD", 3);

/* (6)*/ // start all the declaration.
/* (7)*/ red_begin_declaration();

/* (8)*/ // define constants used in RED descriptions.
/* (9)*/ red_comment("Three constants used in the specification.");
/* (10)*/ red_define_const("A", 26);
/* (11)*/ red_define_const("B", 52);
/* (12)*/ red_define_const("LAMBDA", 808);
/* (13)*/ red_comment("3 modes for the bus.");
/* (14)*/ red_define_const("bus_idle", 0);
/* (15)*/ red_define_const("bus_active", 1);
/* (16)*/ red_define_const("bus_collision", 2);

/* (17)*/ red_comment("3 modes for the senders.");
/* (18)*/ red_define_const("sender_wait", 0);
/* (19)*/ red_define_const("sender_transm", 1);
/* (20)*/ red_define_const("sender_retry", 2);

/* (21)*/ // declare variables
/* (22)*/ red_comment("One local clock.");
/* (23)*/ red_declare_variable(RED_TYPE_DISCRETE, "bus", 0, 2);
/* (24)*/ red_declare_variable(RED_TYPE_CLOCK, "x0", 0, 0);
/* (25)*/ red_declare_variable(RED_TYPE_DISCRETE, "sender1", 0, 2);
/* (26)*/ red_declare_variable(RED_TYPE_CLOCK, "x1", 0, 0);
/* (27)*/ red_declare_variable(RED_TYPE_DISCRETE, "sender2", 0, 2);
/* (28)*/ red_declare_variable(RED_TYPE_CLOCK, "x2", 0, 0);

/* (29)*/ // declare synchronizers, which are also global variables
/* (30)*/ red_comment("4 synchronizers.");

/* (31)*/ // start declaring the optional model structure.
/* (32)*/ // modes for the bus.
/* (33)*/ red_comment("Invrairance for the system.");
/* (34)*/ invariance = "
    ( bus==bus_idle
    || bus==bus_active
    || (bus==bus_collision&&x0<A)
    )
    
    && ( sender1==sender_wait
    || (sender1==sender_transm&&x1<=LAMBDA)
    || (sender_1==sender_retry&&x1<B)
    )
    
    && ( sender2==sender_wait

105
(sender2==sender_transm&&x2<=LAMBDA) \n|| (sender_2==sender_retry&&x2<B) 
); 

/*(35)*/ red_comment("transitions for the system."); 
/*(36)*/ red_comment("bus idle"); 

/*(37)*/ xtion[0] = "when (bus==bus_idle & sender1==sender_wait) \n may x0=0; x1=0; bus=bus_active; sender1=sender_transm;"; 
/*(38)*/ xtion[1] = "when (bus==bus_idle & sender2==sender_wait) \n may x0=0; x2=0; bus=bus_active; sender2=sender_transm;"; 

/*(39)*/ red_end_mode(); 
/*(40)*/ red_begin_mode("active", "true"); 
/*(41)*/ red_transition("?end (true)", "x= 0; goto idle;" ); 
/*(42)*/ red_transition("!busy (x >= A)", ";" ); 
/*(43)*/ red_transition("?begin (x < A)", "x= 0; goto collision;" ); 
/*(44)*/ red_end_mode(); 
/*(45)*/ red_begin_mode("collision", "x < A"); 
/*(46)*/ red_transition("!cd :cd (x < A)", "x= 0; goto idle;" ); 
/*(47)*/ red_end_mode(); 

/*(48)*/ // modes for the senders. 
/*(49)*/ red_comment("3 modes for the senders."); 
/*(50)*/ red_begin_mode("wait", "true"); 
/*(51)*/ red_transition("!begin (true)", "x= 0; goto transm;" ); 
/*(52)*/ red_transition("?cd (true)", "x= 0;" ); 
/*(53)*/ red_transition("?cd (true)", "x= 0; goto retry;" ); 
/*(54)*/ red_transition("?busy (true)", "x= 0; goto retry;" ); 
/*(55)*/ red_end_mode(); 
/*(56)*/ red_begin_mode("transm", "x <= LAMBDA"); 
/*(57)*/ red_transition("!end (x==LAMBDA)", "x= 0; goto wait;" ); 
/*(58)*/ red_transition("?cd (x>B)", "x= 0; goto retry;" ); 
/*(59)*/ red_end_mode(); 
/*(60)*/ red_begin_mode("retry", "x < B"); 
/*(61)*/ red_transition("!begin (x < B)", "x= 0; goto transm;" ); 
/*(62)*/ red_transition("?busy (true)", "x= 0;" ); 
/*(63)*/ red_transition("?cd (x < B)", "x= 0;" ); 
/*(64)*/ red_end_mode(); 

/*(65)*/ // finish all the declaration and start constructing tables. 
/*(66)*/ red_end_declaration(); 

/*(67)*/ // print out some tables to file 'out'. 
/*(68)*/ red_print_variables(out); 
/*(69)*/ red_print_xtions(out); 
/*(70)*/ red_print_sync_xtions(out); 
/*(71)*/ // print out those transitions to be executed in a bulk. 
/*(72)*/ red_print_diagram(out, red_bulk_xtions()); 

/*(73)*/ red_init = red_diagram( 
 "idle[1] & x[1]==0 & forall i:i>1, (wait[i] & x[i]==0)" ); 
/*(74)*/ ini = red_push(red_init); 
/*(75)*/ red_print_line(out, red_stack(ini));
// get an abstract image of the forward reachability.
inv = red_push(red_query_declared_invariance_diagram());
red_set_stack(inv, red_reach_unimed_fwd(red_stack(inv), red_stack(ini)));
red_set_stack(inv, red_reach_magnitude_fwd(red_stack(inv), red_stack(ini)));

// risk analysis.
red_set_stack(rch, red_reach_bck(red_stack(inv), red_stack(rch)));
if (red_normal(red_and(red_stack(rch), red_stack(ini))) != RED_FALSE())
fprintf(out, "The system is risky.\n");
else
fprintf(out, "The system is safe.\n");
red_pop(rch);
red_pop(inv);
red_pop(ini);
red_end_session("CSMA-CD");

B Syntax of RED input file format