Extended Kronos/CADP tool: minimization, on-the-fly verification and compositionality

Stavros Tripakis
VERIMAG
Centre Équation, 2, rue de Vignate, 38610, Gières, France
E-mail: tripakis@imag.fr

January 1999

Abstract

This is chapter 11 of the PhD thesis [Tri98] where all details concerning the methods implemented in the tool can be found.

1 Tools

KRONOS [1] is a tool suite for the analysis of real-time systems. It has been developed at VERIMAG since 1992 [Yov93, DOY94, DOTY96, Yov97, BDM+98]. The current state of the tool is illustrated in figure 1. The upper part of the figure (enclosed in dotted box) represents what can be called the "first generation" of KRONOS, consisting in a collection of modules acting as *interpreters*, that is, performing the analysis directly on their input model. As of today, the modules of KRONOS are the following:

- `ptg` computes the parallel composition of a set of TA syntactically;
- `kronos` performs TCTL, TBA and reachability model checking (see section 1.1 below);
- `minim` computes the quotient graph of a TA with respect to the strong time-abstracting bisimulation;
- `synth-kro` performs controller synthesis for invariance and reachability, using the fix-point method;
- `optikron` computes the set of active clocks per discrete state of a TA (i.e., the function `act`()), and accordingly optimizes the number of clocks using renaming and cross-clock assignments (see [Daw98] for more details).

The input language of these modules is the basic TA model, that is, finite-state automata with clocks, communicating by label synchronization.

The C-code generator module, called `kronos-open`, represents the next generation of KRONOS. It is based on the *compiler* philosophy of SPIN [Hol91], followed by OPEN-CAESAR [Gar98]. Given an input model, `kronos-open` produces C-code which can be in turn compiled to various executables, which perform the analysis for the specific input model. The interest behind this approach is that it permits to take advantage of the particularities of each input model in order to generate optimized code. Another difference from the first-generation tools is that `kronos-open` accepts a richer input language, namely, TA extended with bounded discrete variables and shared-variable or message-passing communication.

As part of the work for this thesis, we have contributed to the development of KRONOS by:

[1] Named after the Titan of ancient Greek mythology, often indiscernible with *chronos*, which in Greek means "time".

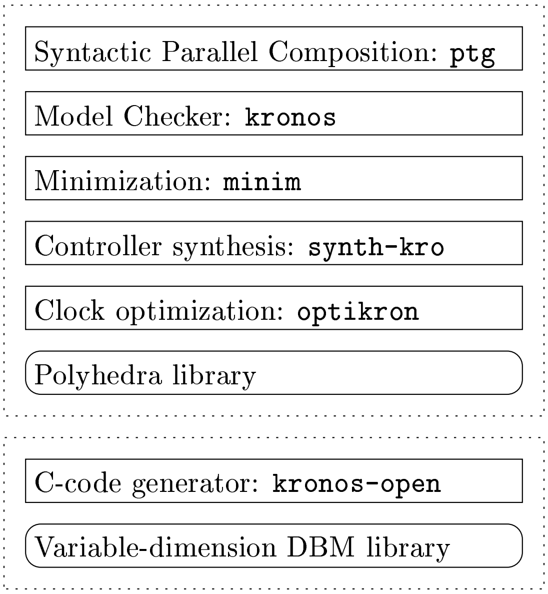

Figure 1: The modules of the KRONOS tool suite.

- extending `kronos` with a module computing the parallel composition of a set of TA on-the-fly;
- extending `kronos` with a module for TBA model checking and reachability based on abstractions, which uses the on-the-fly parallel-composition module above;
- implementing the module `minim`;
- implementing the prototype version of `synth-kro`;
- implementing a library of variable-dimension DBMs, to be used when activity abstraction is applied during the analysis;
- implementing `kronos-open`: to date, it produces code for TBA model checking and reachability and uses the variable-dimension DBM module above.

For the extensions of `kronos` and the implementation of `minim` and `synth-kro`, we have used the parser and DBM library of KRONOS, developed by S.Yovine, A.Olivero and C.Daws. For the implementation of `kronos-open`, we have used the parser of SMI, developed by M.Bozga [Boz97]. The implementation of `synth-kro` has been completed by K.Altisen.

In the following sections, we present `kronos`, `minim`, `synth-kro` and `kronos-open`.

1.1 The model checker `kronos`

The functionalities of `kronos` are shown in figure 2. The tool operates in one of following basic modes:

1. Full-TCTL model checking (top of figure): the system to be verified is given as a TA A (file `.tg`) and the property as a TCTL formula ϕ (file `.tctl`). The tool computes the set of states of A satisfying ϕ (i.e., the characteristic set of ϕ). The output is given in terms of a symbolic state. Since the input is given as a single TA, in case of a system consisting of more than one components, they should be statically composed before the analysis.
2. Safety-TCTL model checking (second from top in the figure): using forward reachability analysis based on simulation graphs, this mode can treat a sub-class of TCTL formulae, such as invariance

($\forall\Box\, p$) and bounded response ($\forall\Box\,(p_1 \Rightarrow \forall\Diamond_{\leq c}\, p_2)$). The input system is given as a single TA, as in the previous mode. The output is a yes/no answer possibly accompanied by a symbolic diagnostic trace.

3. Reachability-TCTL model checking (third from top in the figure): this mode is used to check reachability of discrete states [2] using abstractions and on-the-fly parallel composition of the input system, which is given as a collection of TA. The property is given as a *state formula*, that is, a boolean expression of atomic propositions. As previously, a yes/no answer is returned, plus a symbolic diagnostic trace whenever reachability holds.

4. TBA model checking (bottom of figure): the property here is given in terms of a TBA, the discrete states of which are labeled with boolean expressions of atomic propositions on the system. As in previous mode, parallel composition is computed on-the-fly. Diagnostics are reported in terms of symbolic paths ending in a cycle. The implemented technique is based on the double-DFS algorithm of [CVWY92] to find non-zeno, accepting cycles. This technique is sound but generally incomplete.

Function modes 3 and 4 have been implemented as part of this thesis. We now give examples of their usage, for the verification of the well-known Train Gate Closure system. The input `.tg` files (text) for the three automata are shown below:

```
/* Train.tg */
#states 3
#trans 3
#clocks 1 X

state: 0
prop: far
invar: true
trans:
true => approach; reset{X}; goto 1

state: 1
prop: near
invar: X<=5
trans:
X>2 => in; reset{}; goto 2

state: 2
prop: in
invar: X<=5
trans:
X<=5 => exit; reset{}; goto 0

/* Gate.tg */
#states 4
#trans 4
#clocks 1 Y
```

[2]Any safety property can be reduced to (negation of) reachability. For this, it is sometimes necessary to use an auxiliary automaton to *monitor* the system and move to an error state whenever the property is violated.

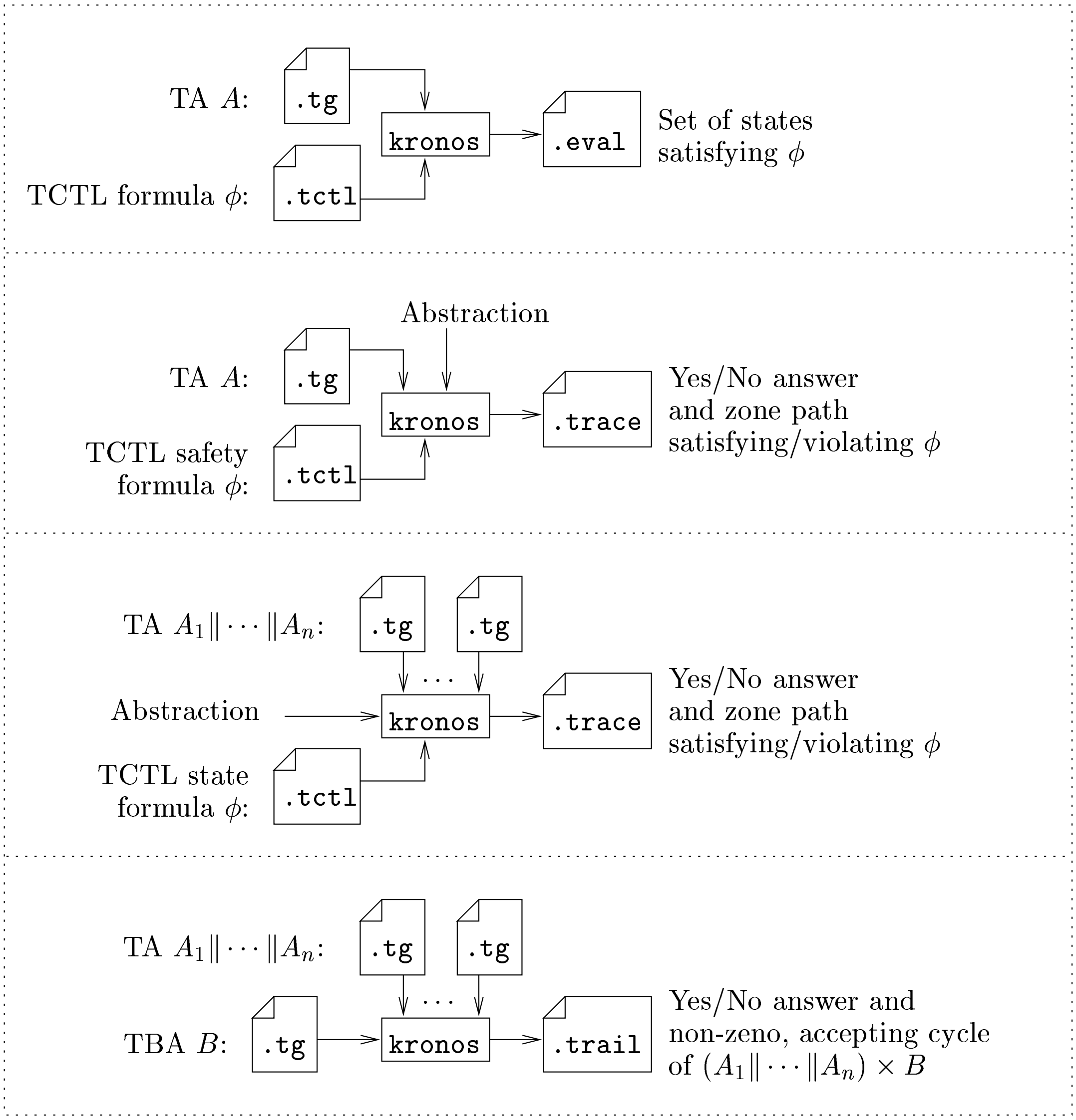

Figure 2: The function modes of the model checker `kronos`.

```
state: 0
prop: up
invar: true
trans:
true => lower; reset{Y}; goto 1

state: 1
prop: coming_down
invar: Y<1
trans:
Y<1 => down; reset{}; goto 2

state: 2
prop: down
invar: true
trans:
true => raise; reset{Y}; goto 3

state: 3
prop: going_up
invar: Y<=2
trans:
Y>=1 => up; reset{}; goto 0

/* Controller.tg */
#states 4
#trans 4
#clocks 1 Z

state: 0
prop: c0
invar: true
trans:
true => approach; reset{Z}; goto 1

state: 1
prop: c1
invar: Z<=1
trans:
Z=1 => lower; reset{}; goto 2

state: 2
prop: c2
invar: true
trans:
true => exit; reset{Z}; goto 3

state: 3
prop: c3
invar: Z<=1
```

```
trans:
Z<=1 => raise; reset{}; goto 0
```

We would like to check the formula $\forall\Box\,(\text{in} \Rightarrow \text{down})$, stating that whenever the train is in the crossing, the gate is down. For this, it suffices to check that the state formula $(\text{in} \wedge \neg\text{down})$ is not reachable. The property can be verified by running `kronos` as follows:

```
kronos -R "in and not down" Train.tg Gate.tg Controller.tg
...
Building synchronization tables...
Using breadth-first search with max symbolic-states set size: 1000
Reachability failed.
Full state space explored: 11 states. Max depth reached: 9
```

The tool reports that reachability has failed, meaning that invariance holds.

As a second example, consider the bounded-response property "whenever the gate is down, it comes up at most 6 time units later". We can verify this property using a TBA encoding the negation of the property, similar to TBA B_2 of figure **??**. The `.tg` file for this automaton (`buchi_bounded.tg`) is shown below:

```
/* Timed Buchi automaton testing bounded response */
#states 4
#trans 4
#clocks 1 W

state: 0
invar: true
trans:
true => go; reset{ W }; goto 1

state: 1
prop: down
invar: true
trans:
true => wait; reset{}; goto 2

state: 2
prop: not up
invar: true
trans:
W>6 => error; reset{}; goto 3

state: 3
prop: accept
invar: true
trans:
true => accept; reset{}; goto 3
```

Then, we can run `kronos` as follows:

```
kronos train.tg gate.tg control.tg buchi_bounded.tg
```

```
...
Building synchronization tables...
On-the-fly model checking by Buchi acceptance.
Using depth-first search with max stack depth: 1000
Search for acceptance cycles successful.
Sample scenario dumped in file: buchi_bounded.trail
State space explored: 19 states. Max depth reached: 18
```

The tool reports a counter-example of length 17, meaning that the property does not hold. In fact, we can get a shorter counter-example by limiting the size of the DFS stack to 12:

```
kronos train.tg gate.tg control.tg buchi_bounded.tg -STACK 12
...
Building synchronization tables...
On-the-fly model checking by Buchi acceptance.
Using depth-first search with max stack depth: 12
Search for acceptance cycles successful.
Sample scenario dumped in file: buchi_bounded.trail
State space explored: 13 states. Max depth reached: 11
```

The buchi_bounded.trail file is shown below:

```
                   Path reaching cycle (length: 10)

0: < 0, 0, 0, 0,    X=Y and X=Z and X=W >
  ---  APPROACH --->
1: < 1, 0, 1, 0,    X=1 and Z=1 and 1<=W and X<=Y and Y=W >
   ---  LOWER   --->
2: < 1, 1, 2, 0,    1<=X and X<2 and X=Y+1 and X=Z and X<=W >
   ---  DOWN    --->
3: < 1, 2, 2, 0,    1<=X and X<2 and X=Y+1 and X=Z and X<=W >
   ---  GO      --->
4: < 1, 2, 2, 1,    2<X and X<=5 and X=Y+1 and X=Z and X<W+2 and W+1<=X >
   ---  WAIT    --->
5: < 1, 2, 2, 2,    2<X and X<=5 and X=Y+1 and X=Z and X<W+2 and W+1<=X >
   ---  IN      --->
6: < 2, 2, 2, 2,    X<=5 and 3<W and X=Y+1 and X=Z and W+1<=X >
   ---  EXIT    --->
7: < 0, 2, 3, 2,    Z<=1 and 4<W and X=Y+1 and X<=Z+5 and Z+4<X and W+1<=X >
   ---  RAISE   --->
8: < 0, 3, 0, 2,    Y<=2 and 6<W and X<=Z+5 and W+1<=X and Y<Z and Z<=Y+1
                         and Z+3<W >
   ---  ERROR   --->
9: < 0, 3, 0, 3,    1<Y and Y<=2 and 6<=W and X<=Z+5 and W+1<=X and Z<=Y+1
                        and Y+4<W >
   ---  UP      --->

                   Cycle (length: 0)

10: < 0, 0, 0, 3,   1<Y and 6<=W and X<=Z+5 and W+1<=X and Z<=Y+1 and Y+4<W >
   ---  ACCEPT  --->
               ... back to node 10 ...
```

1.2 The minimization module `minim`

Figure 3 illustrates the usage of the module `minim`. The tool takes as input a TA [3] and outputs its STa-quotient. Optionally, the initial partition can also be given as input. By default the initial partition consists in a set of zones $(q, \zeta_1), ..., (q, \zeta_m)$ for each discrete state q, where $\zeta_1, ..., \zeta_m$ is the *canonical decomposition* of the guards of edges leaving q: for each zone ζ_i and each guard, ζ_i is either included in the guard or has an empty intersection with it. For example, if $x \leq 1$ and $y > 2$ are the guards, we would obtain four zones, namely, $x \leq 1 \wedge y > 2$, $x \leq 1 \wedge y \leq 2$, $x < 1 \wedge y > 2$ and $x < 1 \wedge y \leq 2$.

The output comes in a various set of formats, including the (untimed) labeled graph format `.aut` of CADP and an extended TA format `.mtg` to represent τ-transitions. Typically, the `.aut` graphs produced by `minim` are given as input to `aldebaran`, in order to be re-minimized or compared with respect to various (untimed) bisimulations or simulations. They can also be visualized (when they are reasonably small) using the module `bcg_edit`, or model checked against μ-calculus formulae using the module `evaluator`.

For example, the STa-quotient of the TGC system (figure **??**) can be minimized with respect to the observational equivalence, yielding the following graph (in `.aut` format):

```
des (0, 9, 8)

(0, APPROACH,2)
(1, APPROACH,7)
(1, UP,0)
(2, LOWER,3)
(3, DOWN,4)
(4, IN,5)
(5, EXIT,6)
(6, RAISE,1)
(7, UP,2)
```

According to lemma **??**, this graph is the Tao-quotient of the TGC system. We can use `bcg_edit` to visualize and transform the graph, which can then be output in postscript format, shown in figure **??**.

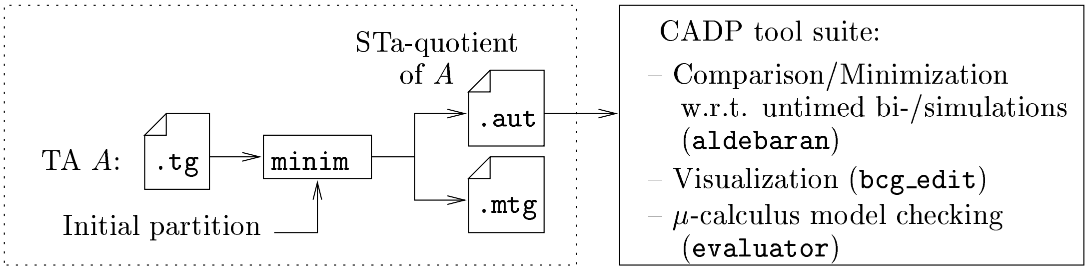

Figure 3: The minimization module `minim`.

1.3 The controller-synthesis module `synth-kro`

This module is presented in figure 4. It takes as input:

- a TA in `.tg` format (the special label `U__` is used to mark uncontrollable edges);

[3] Actually, `minim` accepts as input the parsed `.tg` file. The parsing is done by `kronos`.

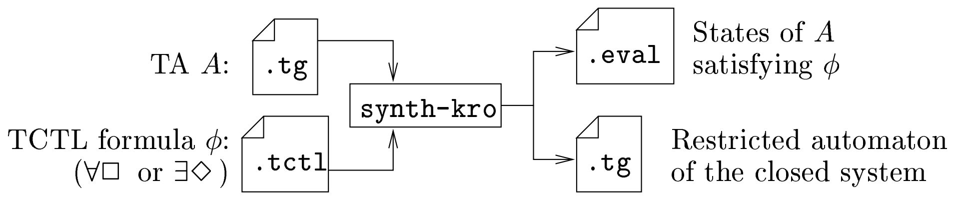

Figure 4: The controller-synthesis module `synth-kro`.

- a TCTL formula in `.tctl` format, of the form $\forall\Box\,\phi$ (invariance) or $\exists\Diamond\,\phi$ (reachability), where ϕ is a boolean expression on atomic propositions.

The tool produces two output files:

- a `.eval` file containing the set of winning states;
- (if the above set is non-empty) a `.tg` file specifying the restriction of the input TA to the set of winning states.

The TGC example of figure **??** is specified by the TA shown below:

```
/* Timed graph generated for the parallel composition of:
        automaton 0: train.tg
        automaton 1: gate.tg
        automaton 2: control.tg
*/
#states 12
#trans  17
#clocks  3
        X /* train */
        Y /* gate */
        Z /* control */


state: 0        /* vector state: < 0, 0, 0 > */
prop: FAR UP C0
invar: TRUE
trans:
1<=X => U__ APPROACH ; RESET{ X Z }; goto 1

state: 1        /* vector state: < 1, 0, 1 > */
prop: NEAR UP C1
invar: X<=5 and Z<=3
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 2
Z<=3 => LOWER ; RESET{ Y }; goto 3

state: 2        /* vector state: < 2, 0, 1 > */
prop: IN UP C1
invar: X<=5 and Z<=3
trans:
Z<=3 => LOWER ; RESET{ Y }; goto 4
```

```
state: 3       /* vector state: < 1, 1, 2 > */
prop: NEAR COMING_DOWN C2
invar: X<=5 and Y<1
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 4
Y<1 => U__ DOWN ; reset{}; goto 5

state: 4       /* vector state: < 2, 1, 2 > */
prop: IN COMING_DOWN C2
invar: X<=5 and Y<1
trans:
Y<1 => U__ DOWN ; reset{}; goto 6
X<=5 => U__ EXIT ; RESET{ X Z }; goto 7

state: 5       /* vector state: < 1, 2, 2 > */
prop: NEAR DOWN C2
invar: X<=5
trans:
2<X and X<=5 => U__ IN ; reset{}; goto 6

state: 6       /* vector state: < 2, 2, 2 > */
prop: IN DOWN C2
invar: X<=5
trans:
X<=5 => U__ EXIT ; RESET{ X Z }; goto 8

state: 7       /* vector state: < 0, 1, 3 > */
prop: FAR COMING_DOWN C3
invar: Y<1 and Z<=1
trans:
Y<1 => U__ DOWN ; reset{}; goto 8

state: 8       /* vector state: < 0, 2, 3 > */
prop: FAR DOWN C3
invar: Z<=1
trans:
Z<=1 => RAISE ; RESET{ Y }; goto 9

state: 9       /* vector state: < 0, 3, 0 > */
prop: FAR GOING_UP C0
invar: Y<=2
trans:
1<=Y and Y<=2 => U__ UP ; reset{}; goto 0
1<=X => U__ APPROACH ; RESET{ X Z }; goto 10

state: 10      /* vector state: < 1, 3, 1 > */
prop: NEAR GOING_UP C1
invar: X<=5 and Y<=2 and Z<=3
trans:
```

```
2<X and X<=5 => U__ IN ; reset{}; goto 11
1<=Y and Y<=2 => U__ UP ; reset{}; goto 1

state: 11       /* vector state: < 2, 3, 1 > */
prop: IN GOING_UP C1
invar: X<=5 and Y<=2 and Z<=3
trans:
1<=Y and Y<=2 => U__ UP ; reset{}; goto 2
```

Running `synth-kro` on the above TA and the TCTL formula $\forall\Box$ (in $\Rightarrow$ down), yields the following restricted TA:

```
/* closed system for AB(IN impl DOWN) */

#states 12
#trans 17
#clocks 3 X Y Z

state: 0
prop: C0 UP FAR
invar: TRUE
trans:
1<=X => APPROACH U__; RESET{ X Z } ; goto 1

state: 1
prop: UP C1 NEAR
invar: X<=1 and Z<=3
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 2
X<=1 and Z<=3 => LOWER; RESET{ Y } ; goto 3

state: 2
prop: UP C1 IN
invar: false
trans:
false => LOWER; RESET{ Y } ; goto 4

state: 3
prop: NEAR C2 COMING_DOWN
invar: X<=5 and Y=1 or Y<=1 and X<=Y+1
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 4
Y<=1 => U__ DOWN; RESET{} ; goto 5

state: 4
prop: IN C2 COMING_DOWN
invar: false
trans:
Y<1 => U__ DOWN; RESET{} ; goto 6
X<=5 => U__ EXIT; RESET{ X Z } ; goto 7
```

```
state: 5
prop: NEAR C2 DOWN
invar: X<=5
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 6

state: 6
prop: IN C2 DOWN
invar: X<=5
trans:
X<=5 => U__ EXIT; RESET{ X Z } ; goto 8

state: 7
prop: FAR COMING_DOWN C3
invar: false
trans:
Y<1 => U__ DOWN; RESET{} ; goto 8

state: 8
prop: FAR DOWN C3
invar: X=0 and Z<=1
trans:
X=0 and Z<=1 => RAISE; RESET{ Y } ; goto 9

state: 9
prop: C0 FAR GOING_UP
invar: Y<=2 and X<=Y or 1<=Y and Y<=2 and Y<=X+1 or 1<=Y and Y<=2 and X<=Y+8
trans:
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 0
1<=X => APPROACH U__; RESET{ X Z } ; goto 10

state: 10
prop: C1 NEAR GOING_UP
invar: Y<=2 and X+1<=Y and Z<=Y+1 or Z<=3 and X+2<=Z and Y+1<=Z and Z<=Y+2
trans:
2<X and X<=5 => U__ IN; RESET{} ; goto 11
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 1

state: 11
prop: C1 IN GOING_UP
invar: false
trans:
1<=Y and Y<=2 => U__ UP; RESET{} ; goto 2
```

Notice that only controllable edges have restricted guards, for instance, the edge `LOWER` of state 1 finds its guard restricted from $z \leq 3$ to $x \leq 1 \wedge z \leq 3$. Also notice that the discrete states which are eliminated by the synthesis algorithm have invariant `false`.

1.4 The connection of Kronos to Open-Caesar

In this section we describe the code-generator `kronos-open` which interfaces Kronos to the verification platform Open-Caesar. The steps of the verification process using `kronos-open` are illustrated in figure 5 and explained in the following paragraphs.

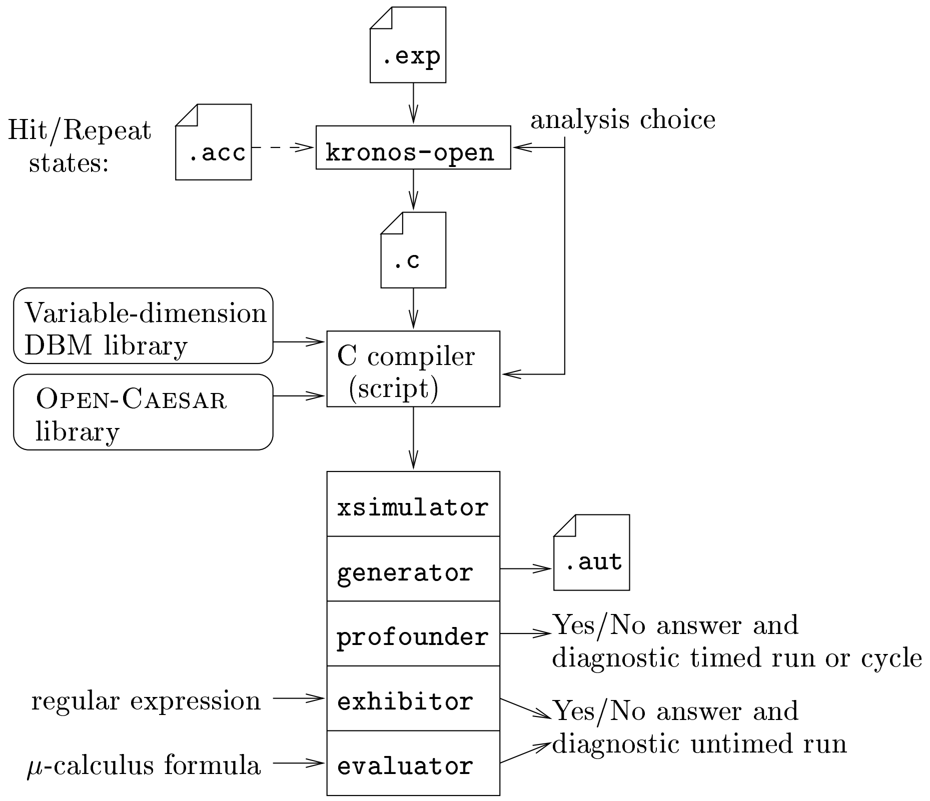

Figure 5: The usage of the module `kronos-open`.

1.4.1 Input

The input is given as a Lotos-like expression specifying the system components and their channel connections (file `.exp`). For example, the `.exp` file for the Bang&Olufsen case study is shown below:

```
Lotos-Behavior

  (
   Bus
   |[ zero, one ]|
   (
    (
     ( Sender_A |[ a_check ]| Detector_A )
     |[ a_frame, a_new_pn, a_reset ]|
       FrameGen_A
    )
    |||
    (
```

```
      ( Sender_B |[ b_check ]| Detector_B )
      |[ b_frame, b_new_pn, b_reset ]|
        FrameGen_B
     )
    )
   )
   |||
   Observer
```

In the above expression, names such as Bus, Sender_A, etc, denote the TA of the system. For each of these names there is a .aut file containing the description of the TA. Names such as zero, one, etc, are *channels*, used for communication between different components. Communication takes place through synchronization of two or more components. To specify which components synchronize on which channels, the notation |[...]| is used, for instance, the Bus synchronizes with the rest of the system on channels |[zero, one]|.

Apart from clocks, the TA can have boolean, bounded-integer, and enumerative-type variables. There is an associated .types file containing type definitions, such as:

```
enum msg {m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10}
```

The variables and the structure of each TA are specified in the corresponding .aut file. Part of this file for the TA Sender_A is shown below:

```
/* variable declaration */

a_c      : clock
a_pf     : bool
a_pn     : bool
a_s1     : bool
a_s2     : bool
a_msg    : msg
...
atomic   : bool
\index{states!atomic}

/* the TA */

des(0,34,19)

/* start */
(0, [~atomic] send a_go a_c:=0, 1)

/* idle */
(1, [(~atomic) and (a_c=781)] send a_start_frame atomic:=true a_c:=0, 2)

/* atomic:ex_start */
(2, [(a_c<=0) and (~b_start)] send b_silent atomic:=false, 3)
(2, [(a_c<=0) and  b_start] send b_sends atomic:=false, 4)

/* ex_silence1 */
(3, [(~atomic) and (a_c=2343)] receive a_zero a_c:=0, 1)
```

```
(3, [(~atomic) and (a_c=2343)] receive a_one a_c:=0, 6)

/* other_started */
(4, [(~atomic) and  b_start and (a_c=3124)] send b_sends a_c:=0, 4)
(4, [(~atomic) and (~b_start) and (a_c=3124)] send b_silent a_c:=0, 3)

/* atomic:goto_idle (removed) */

/* ex_silence2 */
(6, [(~atomic) and (a_c=781)] receive a_zero a_c:=0, 1)
(6, [(~atomic) and (a_c=781)] receive a_one a_c:=0, 7)

/* transmit */
(7, [(~atomic) and (a_c=781)] send a_frame a_err:=e0 a_diff:=false
                                  a_pf:=true atomic:=true a_c:=0, 8)

...

/* until_silence */
(16, [(~atomic) and (a_c=781)] receive a_zero a_c:=0, 16)
(16, [(~atomic) and (a_c=781)] receive a_one a_c:=0, 17)

/* hold */
(17, [(~atomic) and (a_c=28116)] send a_hold a_res:=r0 a_c:=0, 1)

/* jam */
(18, [(~atomic) and (a_c=25000)] send a_jam a_pn:=true
                                  a_start:=false a_res:=r0 a_c:=0, 7)

/* invariants */

[1, a_c<=781]
[2, ~atomic]
[3, a_c<=2343]
[4, a_c<=3124]
...
[17, a_c<=28116]
[18, a_c<=25000]
```

1.4.2 Code generation

`kronos-open` creates a `.c` file which implements the on-the-fly generation of the simulation graph of the input model. The core of the `.c` file consists of the data structures to represent symbolic states (zones) and edges, and the implementation of post(). More precisely:

- A record-like data structure is used to represent zones. The structure has a separate field for each discrete variable, plus an additional field for the convex polyhedron. The size of each discrete-variable field is the number of bits necessary to encode the type of the variable. The field for the polyhedron is a pointer to a variable-dimension DBM. The implementation for the latter is parameterized by the maximal number of clocks (depending on the input model) and is contained in a separate library.

- There is a C function to produce the initial zone.
- `post`() is implemented by a set of C-functions:
  - Two functions for each edge e: the first one takes as input a zone and returns its intersection with the guard of e; the second function performs the assignments on the discrete variables and applies the clock-reset and time-passage operators to the DBM.
  - An iterator function which takes as input a zone and generates its successors one by one. The out-going edges of the zone are computed on-the-fly, based on information stored about the possible channel synchronizations of the input model.

It is worth noticing the main benefit of the compiler approach, compared to the interpreter one: guards, asssignments and clock resets are transformed directly to C code, which results in more efficient execution, than having generic functions for the above operations. In particular, when these operations are trivial (e.g., `true` guard, no assignment) they can be completely skipped.

On the other hand, the approach has the potential disadvantage of explosion of the size of the `.c` file generated, in case there is a very large number of transitions in the input model. Luckily, this is rarely the case, since these are high-level transitions: at the TA level, not at the graph level.

1.4.3 Final output

After the `.c` file has been generated, it is compiled and linked to the OPEN-CAESAR and DBM libraries using a script. As a result, we obtain an executable program performing a certain type of analysis. An option given to the script tells it which type of analysis is to be performed, that is, which type of executable is to be generated. Currently, the following types of executables are available:

- `xsimulator` performs user-guided simulation in a window-based environment.
- `generator` builds the simulation graph of the system in untimed labeled graph `.aut` format.
- `profounder` performs reachability analysis or TBA emptiness. It takes as supplementary input a `.acc` file specifying the discrete states to be reached (*hit* states) or the repeating states (*repeat* states). In case of reachability, `profounder` can generate timed diagnostics.
- `exhibitor` searches for a finite untimed trail matching an input regular expression.
- `evaluator` performs μ-calculus model checking.

Relation to the literature

Apart from KRONOS, perhaps the most successful tool for dense-time verification is UPPAAL [LPY97]. To our knowledge, `synth-kro` is currently the only tool for dense-time controller synthesis.

References

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: a model-checking tool for real-time systems. In *CAV'98*, 1998.

[Boz97] M. Bozga. SMI: An open toolbox for symbolic protocol verification. Technical report, Verimag, March 1997.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992. A preliminary version appeared in the proceedings of CAV'90 (also in Springer Verlag LNCS).

[Daw98] C. Daws. *Méthodes d'analyse de systèmes temporisés: de la théorie à la pratique.* PhD thesis, Institut National Polytechnique de Grenoble, 1998. In french.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In D. Hogrefe and S. Leue, editors, *Proc. 7th. IFIP WG G.1 International Conference of Formal Description Techniques, FORTE'94*, pages 227–242, Bern, Switzerland, October 1994. Formal Description Techniques VII, Champan & Hall.

[Gar98] H. Garavel. Open-Caesar: An open software architecture for verification, simulation and testing. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, LNCS 1384. Springer-Verlag, 1998.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[LPY97] K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[Tri98] Stavros Tripakis. *L'Analyse formelles des Sytèmes temporisés en Pratique.* PhD thesis, Univérsité Joseph Fourier - Grenoble I, December 1998.

[Yov93] S. Yovine. *Méthodes et outils pour la vérification symbolique de systèmes temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, 1993. In french.

[Yov97] S. Yovine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, 1997.