A User's Guide to $\mathbb{Z}$ot

Matteo Pradella

CNR IEIIT, Milano, Italy
pradella@elet.polimi.it
http://home.dei.polimi.it/pradella/

December 2009

Contents

1 Overview

$\mathbb{Z}$ot is an agile and easily extendible bounded model checker, which can be downloaded at http://home.dei.polimi.it/pradella/.

The tool supports different logic languages through a multi-layered approach: its core uses PLTL, and on top of it a decidable predicative fragment of TRIO [8] is defined. An interesting feature of $\mathbb{Z}$ot is its ability to support different encodings of temporal logic as SAT problems by means of plug-ins. This approach encourages experimentation, as plug-ins are expected to be quite simple, compact (usually around 500 lines of code), easily modifiable, and extendible. At the moment, a variant of the eventuality encoding presented in [2] is supported, (approximated) dense-time MTL [5], and a bi-infinite encoding [12], [13].

$\mathbb{Z}$ot offers three basic usage modalities:

1. *Bounded satisfiability checking (BSC)*: given as input a specification formula, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies the specification. An empty history means that it is impossible to satisfy the specification.

2. *Bounded model checking (BMC)*: given as input an operational model of the system, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies it.

3. *History checking and completion (HCC)*: The input file can also contain a partial (or complete) history H. In this case, if H complies with the specification, then a completed version of H is returned as output, otherwise the output is empty.

The provided output histories have temporal length $\leq k$, the bound given by the user, but may represent infinite behaviors thanks to the loop selector variables, marking the start of the periodic sections of the history. The BSC/BMC modalities can be used to check if a property *prop* of the given specification *spec* holds over every periodic behavior with period $\leq k$. In this case, the input file contains spec $\wedge \neg$prop, and, if prop indeed holds, then the output history is empty. If this is not the case, the output history is a counterexample, explaining why prop does not hold.

2 Installation

ℤot's core is written in Common Lisp (with ASDF packaging http://www.cliki.net/asdf). It can be used under Linux, Windows, or MacOS X, but has been tested only under Linux and Windows XP, using the following Common Lisps[1]:

- SBCL (http://www.sbcl.org),
- CLISP (http://clisp.cons.org),
- CMUCL (http://www.cons.org/cmucl/),
- ABCL (http://common-lisp.net/project/armedbear/),
- Clozure CL (http://www.clozure.com/clozurecl.html),

This approach makes ℤot an open system, as it uses Common Lisp also as internal scripting language of the tool, both to define complex verification activities, and to add new constructs and languages on top of the existing ones.

Typically, to install ℤot in a Debian system (or Ubuntu), the user must install a Common Lisp (e.g. one of the packages `clisp`, `sbcl`, `cmucl`, ...), and the `common-lisp-controller` package. To perform a system-wide install of the ℤot packages, just put symbolic links to its .ads files in the /usr/share/common-lisp/systems/ directory. Note that it is possible to avoid a system-wide installation, but in this case the user has to work inside the main ℤot directory.

ℤot works with external SAT-solvers. The supported SAT-solvers are MiniSat (default) [3], MiraXT [9], PicoSAT [1], and zChaff [10]. ℤot assumes that executable files called `minisat`, `MiraXTSimp` (optional), `picosat` (optional), `zchaff` (optional), are system-wide installed.

A pre-packaged all-inclusive version for Windows (*WinZot*, based on Cygwin-compiled binaries and SBCL) is available from the author.

All ℤot's components are available as open source software (GPL v2).

[1] SBCL and CMUCL are usually the fastest implementations, for running Zot.

3 Languages

Being an open system, ℤot supports different languages. At present, the main native language is PLTL (linear temporal logic with future and past operators). The other main layer based on PLTL is the metric temporal logic TRIO.

ℤot scripts are written in Common Lisp, so a basic knowledge of the language is required. It is very easy to find online a lot of tutorials and short presentations[2].

3.1 PLTL

Propositional operators are written as: `&&` (and), `||` (or), `!!` (not).

Predicates and propositional letters e.g., proposition Q is written (-P- Q); predicate Pred(1,2) is written as (-P- Pred 1 2).

Quantifications $\exists t \in \{One, Two\} : Formula(t)$ is written `(-E- t '(One Two) Formula(t))`. -A- is the universal quantifier.

Term comparisons and conditions are available through Common Lisp (e.g. eql, equal, <, <=, and, or, not, . . .)

Temporal operators The following temporal operators are supported: `until`, `since`, `release`, `trigger`, `next`, `yesterday`, `zeta`. The last one is the dual of `yesterday`, and is used only in the mono-infinite semantics.

For the semantics of these operators, see e.g. [2] (which describes the implementation of the mono-infinite encoding in details).

3.2 TRIO

ℤot was originally born as a satisfiability checker for the TRIO metric temporal logic [8].

The list of supported operators (and their correct "ℤot spelling") is the following:

```
dist
futr
past
lasts    lasts_ee    lasts_ie    lasts_ei    lasts_ii
lasted   lasted_ee   lasted_ie   lasted_ei   lasted_ii
withinf  withinf_ee  withinf_ie  withinf_ei  withinf_ii
withinp  withinp_ee  withinp_ie  withinp_ei  withinp_ii
lasttime lasttime_ee lasttime_ie lasttime_ei lasttime_ii
```

[2]e.g. http://gigamonkeys.com/book/ is a good and freely available text.

```
nexttime nexttime_ee nexttime_ie nexttime_ei nexttime_ii
somf     somf_e      somf_i      som
somp     somp_e      somp_i
alwf     alwf_e      alwf_i      alw
alwp     alwp_e      alwp_i
until    until_ie    until_ee    until_ii    until_ei
since    since_ie    since_ee    since_ii    since_ei
```

Bounded version of since and until are written as:

```
(until_ie_<=_<= t1 t2 A B)
B will be true at t instants in the future with t1<=t<=t2
(until_ie_>= t1 A B)
B will be true at t instants in the future with t>=t1
since_ie_<=_<=
since_ie_>=
```

Caveat emptor! The default `until` is PLTL's (which is usually called `until_ie` in TRIO). For example, the following model satisfies `(until A B)` at 0:

```
0                    B
----------------------
AAAAAAAAAAAAAAAAAAAAA
```

B may appear at 0.

For MTL users:

1. $\Diamond_{=t}A$ (or $\Box_{=t}A$)) is written `(futr (-P- A) t)`;
2. $\Box_{\leq t}A$ is written `(lasts (-P- A) t)`;
3. $\Diamond_{\leq t}A$ is written `(withinf (-P- A) t)`;
4. $\blacklozenge_{=t}A$ (or $\blacksquare_{=t}A$)) is written `(past (-P- A) t)`;
5. $\blacksquare_{\leq t}A$ is written `(lasted (-P- A) t)`;
6. $\blacklozenge_{\leq t}A$ is written `(withinp (-P- A) t)`;

with $t > 0$.

3.3 Operational constructs

$\mathbb{Z}$ot offers some simple facilities to describe operational systems.

```
(define-item  <varname> <domain>)
```

is used to define variables à la Von Neumann over finite domains (e.g. counters).

```
(define-array <varname> <index-domain> <domain>)
```

is used to define mono-dimensional arrays.

Example usage:

```
(define-item  cont (loop for i from 0 to 9 collect i))
(define-array arr  (loop for i from 0 to 9 collect i)
                   '(on off unknown))
```

In the spec, the user can e.g. write `(cont= 6)`; `(arr= 6 'off)`.

Caveat: both define-item and define-array have side effects. It is therefore wrong to "define-items" after a zot main procedure call, since successive calls may work with spurious constraints. It is therefore recommended to perform `(clean-up)` before defining items or arrays.

Typically, to define an operational model means to constraint operational variables and arrays. This can be done either by using simple next-time formulae, i.e. containing only the `next` temporal operator, or by using the two dual constructs `and-case` and `or-case` [14].

To give the reader an idea of their semantics, here is an automatic translation made by $\mathbb{Z}$ot on two simple examples.

```
(and-case (x '(1 2) y '(3 4))
          ((-P- P x) (-P- Q x))
          ((-P- R y) (-P- R1 y))
          (else (-P- R2 x)))
```

expands to

```
(-A- X '(1 2)
    (-A- Y '(3 4)
     (&& (-> (-P- R Y) (-P- R1 Y)) (-> (-P- P X) (-P- Q X))
      (-> (&& (!! (-P- R Y)) (!! (-P- P X))) (-P- R2 X)))))
```

and

```
(or-case (x '(1 2) y '(3 4))
          ((-P- P x) (-P- Q x))
          ((-P- R y) (-P- R1 y))
          (else (-P- R2 x)))
```

expands to

```
(-E- X '(1 2)
    (-E- Y '(3 4)
     (|| (&& (-P- R Y) (-P- R1 Y)) (&& (-P- P X) (-P- Q X))
      (&& (!! (-P- R Y)) (!! (-P- P X)) (-P- R2 X)))))
```

3.4 MTL

There is an experimental plug-in (called *ap-zot* for using a variant of dense-time MTL through approximation (see [5], and [4]).

Here is a list of the time operator defined in ap-zot.

```
until-b   until-b-v   until-b-^
since-b   since-b-v   since-b-^
release-b release-b-^ release-b-v
trigger-b trigger-b-^ trigger-b-v

until-b-inf   until-b-v-inf   until-b-^-inf
since-b-inf   since-b-v-inf   since-b-^-inf
release-b-inf release-b-^-inf release-b-v-inf
trigger-b-inf trigger-b-^-inf trigger-b-v-inf

diamond   diamond-inf
diamond-p diamond-inf-p
box       box-inf
box-p     box-inf-p
```

The plug-in offers the following operations

```
normalize
basicize
compute-granularity
over-approximation
under-approximation
nth-divisor
```

To compute over- and under-approximations, an axiom must be prepared through the two functions *basicize* and *normalize*
(e.g. with `(setf ax1 (normalize (basicize ax1)))`).

The two functions *over-approximation* and *under-approximation* are used to compute the approximated formulae, while *compute-granularity* is used to set the ρ parameter (see [5] for details).

The interested reader may find a complete example in `coffee.lisp`.

3.5 Timed Automata

Timed Automata (TA) are supported through a *very* experimental plug-in called *ta-zot* (see [6], [7]), which is based on the approximations offered by *ap-zot*.

First, here is a list of the added operators, and approximations procedures:

```
white-tri
white-tri/3
black-tri
black-tri/3

timed-automaton-under-formula
timed-automaton-over-formula

timed-automata-under-formula
timed-automata-over-formula
```

Here is the main data structure used to represent TA's, together with its interface:

```
(defstruct timed-automaton
  alphabet
  states
  initial-states
  clocks)

(defgeneric add-trans (autom from to lamb constr))
(defgeneric add-label (autom state list-of-symbols))
(defgeneric alpha (autom state))
(defgeneric get-trans-from-states (autom from to))
(defgeneric all-connected-pairs (autom))
(defgeneric all-unconnected-pairs (autom))
(defgeneric get-all-trans (autom))
(defgeneric get-trans-from-clock-reset (autom clock))
```

The interested reader may find a complete example in

`trans_prot.lisp.`

4 Usage

4.1 SAT-solvers

The supported SAT-solvers are MiniSat [3] (which is used by default), MiraXT [9], and zChaff [10].

To use the zChaff SAT-solver, the user has to set the *zot-solver* parameter. For example:

```
(setq sat-interface:*zot-solver* :zchaff)
```

MiraXT is a multi-threaded solver, so to use it we also have to choose the maximum number of threads that it will use:

```
(setf sat-interface:*zot-solver* :miraxt)
(setf sat-interface:*n-threads* 3)
```

4.2 Model Checking

To perform Bounded Model Checking, the user must provide the model through as argument :transitions. Important: every variable used must be declared implicitly by e.g. an initialization formula as the second argument of $\mathbb{Z}$ot.

Here is a simple example: mutex3 (a simple mutual exclusion protocol with three processes).

The first part is used to load the mono-infinite plug-in, and defines the used variables. The first line loads the mono-infinite plug-in, called *eezot*. (*bezot* is the bi-infinite one.)

```
(asdf:operate 'asdf:load-op 'eezot)
(use-package  :trio-utils)

(defvar state-d '(N T C))
(defvar turn-d  '(1 2 3))


(define-array state turn-d state-d)
(define-item  turn  turn-d)

(defconstant decl ; optional declarations, just for checking usage
  (append
   (loop for x in state-d append
         (loop for y in turn-d collect (state= y x)))
   (loop for x in turn-d collect (turn= x))))
```

Then, we define the system initialization and transitions:

```
(defvar init   ; system initialization (at 0)
  (&& (-A- x turn-d (state= x 'N))
      (turn= 1)))

(defvar trans  ; list of model constraints
  (list
   (-A- p turn-d
        (or-case (x state-d)
                 ((state= p 'N)
                  (next (state= p 'T)))

                 ((&& (state= p 'T)
                      (|| (-A- p1 turn-d (-> (not (equal p p1))
                                             (state= p1 'N)))
                       (turn= p)))
                  (next (state= p 'C)))

                 ((state= p 'C)
                  (next (state= p 'N)))

                 (else
                  (&& (state= p x)
                      (next (state= p x))))))

   (or-case (x turn-d)  ; -- schedule --

            ((&& (state= 1 'N) (state= 2 'T) (state= 3 'N))
             (next (turn= 2)))
            ((&& (state= 1 'T) (state= 1 'N) (state= 3 'N))
             (next (turn= 1)))
            ((&& (state= 1 'N) (state= 1 'N) (state= 3 'T))
             (next (turn= 3)))


          ; --- random choice policy ---
            ((&& (state= 1 'T)(state= 2 'T))
             (next (|| (turn= 1)(turn= 2))))
            ((&& (state= 1 'T)(state= 3 'T))
             (next (|| (turn= 1)(turn= 3))))
            ((&& (state= 2 'T)(state= 3 'T))
             (next (|| (turn= 2)(turn= 3))))

            (else
             (&& (turn= x) (next (turn= x)))))))
```

As the reader may see, the transitions are defined as a list of constraints, which must hold on every instant of the time domain.

We then write a simple property we wish to check on the system:

```
(defvar spec
    (alw
      (&&
        (-> (turn= 1) (somf (|| (turn= 2)(turn= 3))))
        (-> (turn= 2) (somf (|| (turn= 1)(turn= 3))))
        (-> (turn= 3) (somf (|| (turn= 1)(turn= 2)))))))
```

The main procedure is called *zot*, and has two arguments: the time bound and the formula to be satisfied (plus some optional switches, e.g. :transitions, :declarations, :loop-free).

To check if spec-0 holds for a time bound of 30, we perform:

```
(eezot:zot 30               ; time bound
      (&& (yesterday init) ; initialization (init must hold at 0)
          (!! spec))       ; (negated) property
      :transitions trans   ; list of model constraints
      :declarations decl   ; (optional) declarations
      )
```

UNSAT means that the desired property holds. If the output is SAT, then *spec* does not hold and ℤot returns a counter-example.

4.3 Completeness

A switch of the *zot* procedure (:loop-free, `nil` by default) is used to check completeness. In the previous example, we can check completeness by performing:

```
(eezot:zot 30            ; time bound
     (yesterday init)   ; initialization (init must hold at 0)
     :transitions trans ; list of model constraints
     :declarations decl ; (optional) declarations
     :loop-free t       ; check completeness
     )
```

UNSAT means that the completeness bound is reached.

The *zot* procedure returns `t` if the spec is satisfiable, `nil` otherwise. So, it is possible to write a loop to actually find the completeness bound, e.g.:

```
(format t "Found: ~s~%"
        (loop for bound from 2 unless
             (eezot:zot bound
                       (yesterday init)
                       :transitions trans
                       :declarations decl
                       :loop-free t
                       )
           return bound))
```

4.4 Satisfiability Checking

Let us now consider a simple example to show how satisfiability checking can be performed with ℤot.

The first line loads the bi-inifinite plug-in.

```
(asdf:operate 'asdf:load-op 'bezot)
(use-package  :trio-utils)
```

We then define the timed lamp spec:

```
(defconstant delta 5)

; Alphabet
; on:  the "on"  button is pressed
; off: the "off" button is pressed
; L:   the light is on

(defconstant init
  (&& (!! (|| (-P- on)(-P- off)(-P- L)))))

(defconstant the-lamp
  (alw (&&
         (<->
          (-P- L)
          (|| (yesterday (-P- on))
              (-E- x (loop for i from 2 to delta collect i)
                   (&& (past (-P- on) x)
                       (!! (withinP_ee (-P- off) x))))))
         (!! (&& (-P- on) (-P- off))))))
```

To obtain a history compatible with the spec, we perform:

```
(bezot:zot  10
    (&& init the-lamp))
```

This is an example history generated by ℤot, where **LOOP**, and **POOL** are the loop selector variables (**POOL** towards the past, **LOOP** towards the future):

```
------ time 0 ------

------ time 1 ------
  **LOOP**
  ON

------ time 2 ------
  ON
  L

------ time 3 ------
  ON
  L

------ time 4 ------
  OFF
  L

------ time 5 ------
  OFF

------ time 6 ------
  OFF

------ time 7 ------
  OFF

------ time 8 ------
  OFF

------ time 9 ------
  **POOL**
  OFF

------ time 10 ------

------ end ------
```

4.5 Temporary data

ℤot uses four files to save temporary data during the verification activity:

1) `output.cnf.txt`
2) `output.sat.txt`
3) `output.hist.txt`

(1) contains the resulting boolean formula of the system (in the standard DIMACS CNF format); (2) is the output of the SAT-solver; (3) is the resulting trace of the system (e.g. a TRIO history).

5 Architecture

ℤot's architecture is based on a PLTL-to-SAT core, which interacts with the "outside world" through a TRIO-based interface and different plug-ins. The core itself is structured as a plug-in, so that different encodings can be defined and used.

More recently (May 2009), we added two plugins to ℤot, natively supporting metric operators (like *lasts, withinf*). These native metric plugins are called *meezot* (mono-infinite), and *mbezot*. Their usage is exactly the same as *eezot* and *bezot* [11].

5.1 PLTL-to-SAT encodings

As said before, ℤot's core is based on encoding PLTL into SAT. At present two main encodings are available in the standard distribution: *eezot*, which is a standard eventuality-based encoding on a mono-infinite time domain (ℕ, see e.g. [2]), and the bi-infinite one, *bezot* [12] on ℤ.

The two encodings are packaged (as asdf systems) in the following files:

```
eezot.lisp   eezot.asd
bezot.lisp  bezot.asd
```

The file `kripke.lisp` contains the basic data structure and the definition of the generics.[3]

```
(defclass kripke ()
  (; time bound i.e. [0..k]
   (the-k       :accessor kripke-k)

   ; number of used prop. variables
   (numvar      :accessor kripke-numvar)

   ; formula -> integer data structure (hash-table)
   (the-list    :accessor kripke-list)

   ; integer -> formula data structure (hash-table)
   (the-back    :accessor kripke-back)

   ; list of propositional letters
   (sf-prop     :accessor kripke-prop)

   ; list of used boolean subformulae
   (sf-bool     :accessor kripke-bool)
```

[3] *kripke* does not actually contain a Kripke structure - names of data structures and generics come from previous, forsaken incarnations of the tool-set.

```
; list of used future-tense subf.
(sf-futr     :accessor kripke-futr)

; list of used past-tense subf.
(sf-past     :accessor kripke-past)

; n. of props used in the encoding
(max-prop    :accessor kripke-maximum)

; resulting SAT formula
(the-formula :accessor kripke-formula)))
```

There is also an old variant of *eezot*, called *ezot*, which supports virtual unrollings (as presented in [2], usually called δ), so its data structure is extended through inheritance. The user may change the default behavior (i.e. $\delta = 0$), by setting ezot:*FIXED-DELTA* to nil, which tells eezot to actually compute δ, or (s)he may change to set it to a fixed meaningful value.

The *call* generic translates a formula/proposition and a time instant into an integer (the SAT-solver proposition); *self* must be an instance of kripke (or of a subclass).

```
(defgeneric call (self obj the-time &rest other-stuff))
```

The *back-call* generic is used to translate an integer in $[0..k]$ into the corresponding subformula; self must be an instance of kripke (or of a subclass).

```
(defgeneric back-call (self x))
(defgeneric back-call-time (self x))
```

5.2 Main Interface

There are two interfaces:

```
sat-interface.lisp
```

the first one is with the SAT-solver, and it is used to send the output of the PLTL encoding to it; then, to parse its output and get a counter-example, if any.

The other one,

```
trio-utils.lisp
```

is the basic interface with the user, and is based on TRIO (see Section 3.2) augmented with the operational constructs covered in Section 3.3.

5.3 Other modules and plug-ins

At present just *ap-zot* and *ta-zot* are available. Please refer to Sections 3.4, 3.5, and the related papers.

The two plug-ins are implemented and packaged (as asdf systems) in

```
ap-zot.lisp  ap-zot.asd
ta-zot.lisp  ta-zot.asd
```

ta-zot is based on *ap-zot*, which uses TRIO as underlying language (through the *trio-utils* interface).

Acknowledgments

I thank the following people: Stefano Riboni for his work on the CNF translator; Davide Casiraghi for the metric plugins (meezot and mbezot).

References

[1] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[2] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5):1–64, 2006.

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT Conference*, volume 2919 of *LNCS*, pages 502–518. Springer-Verlag, 2003.

[4] C. A. Furia, M. Pradella, and M. Rossi. Dense-time MTL verification through sampling. Technical Report 2007.37, DEI, Politecnico di Milano, April 2007.

[5] C. A. Furia, M. Pradella, and M. Rossi. Dense-time MTL verification through sampling. In *Proceedings of FM'08*, volume 5014 of *LNCS*, 2008.

[6] C. A. Furia, M. Pradella, and M. Rossi. Practical automated partial verification of multi-paradigm real-time models. Technical Report arXiv.org 804.4383, April 2008.

[7] C. A. Furia, M. Pradella, and M. Rossi. Practical automated partial verification of multi-paradigm real-time models. In *10th International Conference on Formal Engineering Methods (ICFEM)*, October 2008.

[8] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[9] M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th Conf. on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[11] M. Pradella, A. Morzenti, and P. S. Pietro. A metric encoding for bounded model checking. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 741–756. Springer, 2009.

[12] M. Pradella, A. Morzenti, and P. San Pietro. The symmetry of the past and of the future: Bi-infinite time in the verification of temporal

properties. In *Proc. of The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE*, Dubrovnik, Croatia, September 2007.

[13] M. Pradella, A. Morzenti, and P. San Pietro. Benchmarking model- and satisfiability-checking on bi-infinite time. In *5th International Colloquium on Theoretical Aspects of Computing (ICTAC 2008)*, Istanbul, Turkey, September 2008.

[14] M. Pradella, A. Morzenti, and P. San Pietro. Refining real-time system specifications through bounded model- and satisfiability-checking. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, L'Aquila, Italy, September 2008.