Zot Quick Installation Guide

Table of contents

1 Architecture of the Zot tool

1.1 A brief introduction to the Zot approach to formal verification

Zot is a Bounded Model/Satisfiability Checker that takes as input specifications written in a variety of temporal logics, and determines whether they are satisfiable or not. Through this basic mechanism, it can perform verification of user-defined properties for desired models. Zot can also be used for trace generation purposes.

More precisely, let us call *S* the temporal logic-based model of the system being designed. If *S* is fed to the Zot tool "as is", Zot will look for an execution trace of the modelled system; if it does not find one (i.e., if the model is *unsatisfiable*), then the model is contradictory, hence it contains some flaw that makes it unrealizable in practice.

Now, suppose we introduce a further temporal logic formula, *P*, which formalizes a user-defined property to be checked for the system. We can ask Zot to check whether the formula "S and (not P)" is satisfiable or not. If "S and (not P)" is *unsatisfiable*, this means that there is no execution trace that satisfies the system (i.e., S), that also satisfies "not P", that is, that *violates* the property P. If no system trace violates property P, then the latter actually holds for the system. If, on the other hand, "S and (not P)" is *satisfiable*, this means that there is at least one system trace that satisfies both S and "not P"; that is, there is at least one execution trace of the system that violates the property, so the property *does not* hold for the system. If Zot determines that "S and (not P)" is satisfiable, the tool produces an execution trace that satisfies it, i.e., a *counterexample* trace that is compatible with the system model but that violates the property.

1.2 Architecture of the Zot tool

Zot performs the checks outlined in Section 1.1 by encoding temporal logic formulae into the input language of various solvers. In particular, Zot supports two kinds of solver: SAT solvers and SMT solvers.

SAT solvers are capable of taking as input formulae written in *Propositional Logic*, and determine whether they are satisfiable or not. SMT solvers do the same (determine whether logic formulae are satisfiable or not), but they accept as input formulae written in logics (fragments of First-Order Logic) that are richer than simple Propositional Logic.

Over the last few years both SAT solvers and SMT solvers have made great strides in terms of their performances (i.e., their ability to handle large-sized formulae in a reasonable amount of time), so that they have become viable engines for fully automated logic-based verification approaches such as the one realized by Zot. In addition, most SAT/SMT solvers accept inputs written in a standard format (the DIMACS format [DIMACS] for SAT solvers, and the SMT-LIB format [SMT-LIB] for SMT solvers), which makes them easily interchangeable. This is very useful, since different solvers implement different heuristics, and the "best" solver does not exist in absolute terms, but only on a model-by-model basis.

At its core, Zot encodes specifications written in a variety of temporal logics into the input languages of SAT/SMT solvers. The tool supports several different encodings, which the user can choose by setting suitable options in the verification scripts.

Zot is plugin-based: every encoding is realized by a *plugin*, and to select the encoding to be used the user selects the corresponding plugin.

Figure 1 depicts the architecture of the Zot tool. At its core there are the third-party SAT and SMT solvers. Suitable modules interface Zot with the underlying solvers. Since the two kinds of solvers accept different inputs and have different features, the interface modules are also different depending on the solver: "sat-interface" and "zot2cnf" interface Zot with SAT solvers, while "smt-interface" does the same for SMT solvers.

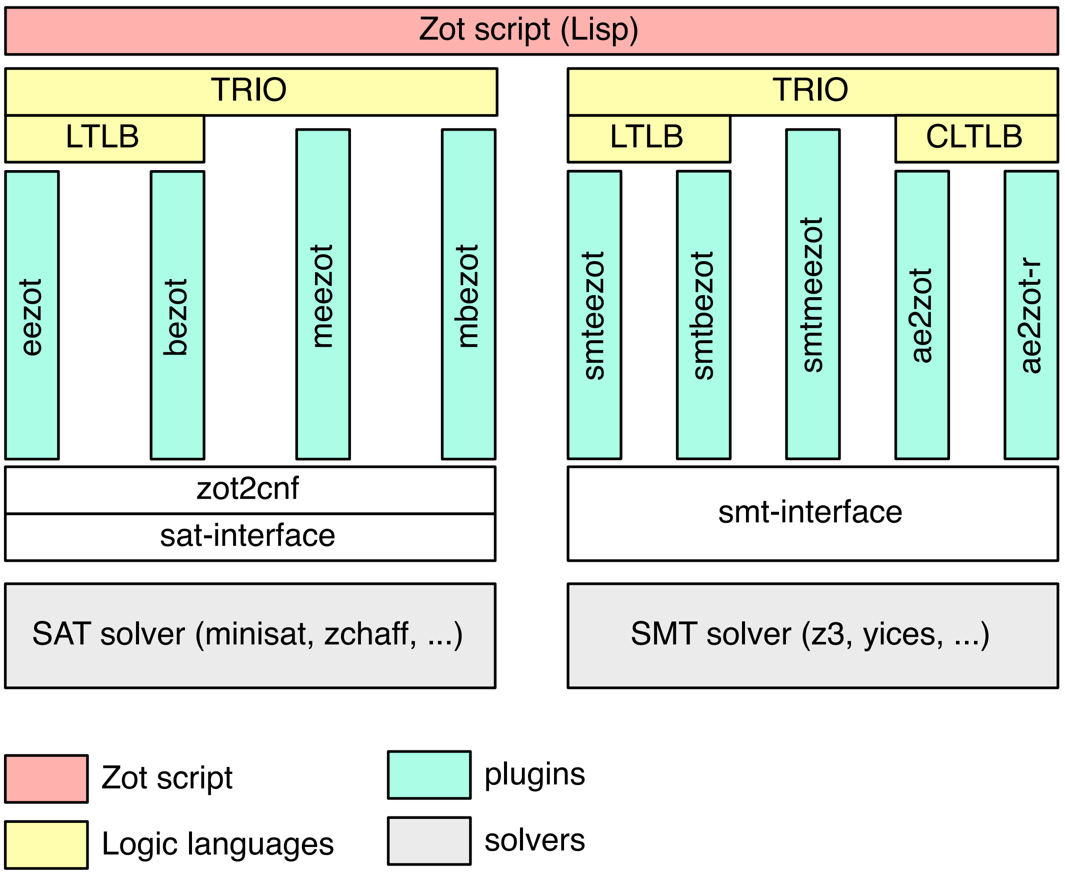

Figure 1 Architecture of the Zot Bounded Model/Satisfiability Checker.

The plugins implementing the different encodings define how temporal logic formulae are translated into the input languages of SAT/SMT solvers. The temporal logics supported by Zot are depicted in the yellow boxes. They are:

- LTLB: propositional Linear Temporal Logic with Both future and past operators, whose main temporal operators are "next" (X), "yesterday" (Y), "until" (U) and "since" (S).
- CLTLB: Constraint LTLB, which has the same temporal operators as LTLB, but which admits, as atomic elements, not only propositional letters, but also a limited set of first-order constraints (e.g., "x < 3").

- TRIO: a very expressive linear temporal logic with a metric notion of time, whose basic temporal operator is *Dist*, which is used to express metric constraints such as "event e will occur at a distance of 3 time instants from now".

Since both LTLB and CLTLB are subsets of the more expressive TRIO, to encode TRIO formulae into the input languages of SAT/SMT solvers some plugins (e.g., *eezot* in Figure 1) first translate TRIO formulae into LTLB/CLTLB formulae, then they encode the latters into the language of the corresponding solver.

Zot plugins implement the primitives that are used in Zot scripts to define the models to be analysed. Since the plugins are implemented as Lisp modules, the primitives are Lisp declarations. Then, Zot scripts, which contain both the model to be analysed and the necessary commands to invoke the desired solver, are a collection of Lisp statements.

The available Zot plugins are the following:

- ***eezot***, which encodes LTLB formulae in the input language of SAT solvers; *eezot* also translates TRIO formulae into LTLB ones, by transforming TRIO the metric operator *Dist* into a series of next/yesterday operators (e.g., "event e will occur at a distance of 3 time instants from now" becomes "the next instant of the next instant of the next instant event e occurs").
- ***meezot***, which encodes TRIO formulae in the input language of SAT solvers; unlke *eezot*, *meezot* natively encodes the metric operator *Dist*, and it is usually faster than *eezot* when the input model has many metric constraints.
- ***smteezot***, which encodes LTLB formulae in the input language of SMT solvers; *smteezot* is akin to *eezot* (for example as far as the translation of the TRIO *Dist* operator is concerned), but it exploits the features of SMT solvers to produce a more compact encoding than *eezot*.
- ***smtmeezot***, which encodes TRIO formulae in the input language of SMT solvers; this plugin, which is the dual of *meezot*, requires an SMT solver that supports the "mod" operator (among those mentioned in Figure 1, only z3 has this feature).
- ***ae2zot***, which encodes CLTLB formulae in the input language of SMT solvers; *ae2zot* differs from *smteezot* in that it supports arithmetic constraints over *integer* numbers such as "x < 3" (*smteezot*, like *eezot*, inly supports propositional LTL formulae). However, it is similar to *smteezot* in that it translates the TRIO *Dist* operator as a sequence of "next"/"yesterday" operators.
- ***ae2zot-r***, which is similar to *ae2zot*, but it supports arithmetic constraints over *real* numbers.

Zot includes also other plugins (*bezot*, *mbezot*, *smtbezot*) which are variants of the plugins above that use a bi-infinite notion of time (i.e., where the time domain are the integers Z).

Determining which plugin (and which solver) to use for verification purposes is often not clear-cut. Experiments carried out with the various plugins have shown that the fastest combination plugin/solver changes from model to model. Hence, the best choice of plugin/solver depends heavily on the shape and the features of the specific model to be analysed, often in an unpredictable way. As a consequence, from a practical point of view, after the designer has defined model to be analysed, we suggest the following two options: either (i) run different combinations plugin/solver on different computers (or on different cores of the same computer) if these are available, then collect the result from the verification instance that terminates first; or (ii) try different combinations, determine which one is the fastest, then continue to use that for all subsequent analyses of the model[1].

The next section describes how the various components needed to run Zot scripts are installed.

[1] Assuming, of course that a model is analysed more than once, which is however the most common situation in a model$\rightarrow$verify$\rightarrow$modify/refine loop.

2 Installation Guide

2.1 Prerequisites

The Zot verification tool relies on the following components:

- Lisp compiler
- SAT/SMT solver

In the following, we outline the installation instructions for the components listed above.

2.1.1 Lisp compiler

As mentioned in Section 1, Zot scripts are collections of Lisp declarations and, as such, must be compiled through a Lisp compiler. Different compilers are suitable for this task. Two possibilities are the Steel Bank Common Lisp (sbcl, http://www.sbcl.org/), and GNU clisp (http://www.clisp.org/). They are both available for a number of platforms, including Windows, Linux, and Mac OS X. In the following we will assume that SBCL is the Lisp compiler of choice. In addition, the scripts included in the distribution of the zot tool are also targeted to SBCL.

SBCL is available for all major platforms. In the following be briefly outline how it can be installed on Windows, Linux, and Mac OS X platforms. For the Zot tool to correctly function the folder in which SBCL is installed is not important; all that matters is that the *sbcl* executable is in the path. All procedures outlined below do not need additional steps to put the *sbcl* executable in the path.

- **Windows**. In this case it is enough to download the *.msi* installer from the SBCL download page (http://www.sbcl.org/platform-table.html), then launch it on the target platform. The installer automatically sets up the path to include the SBCL folder.
- **Linux**. Precompiled binaries for a generic Linux platform are also available from the download page mentioned above. In alternative, SBCL is available in the main Ubuntu repository, so it can easily be installed either with the following command
  apt-get install sbcl
  or via *aptitude* or some other package management system.
- **Mac OS X**. A precompiled binary is also available for the MAC OS X platform (see link above). In addition, SBCL is available through MacPorts (http://www.macports.org/), or through Homebrew (http://mxcl.github.com/homebrew/).

2.1.2 SAT/SMT solver

As mentioned in Section 1, Zot relies on third-party verification engines which can be either SAT solvers or SMT solvers. There are many solvers available which accept the standard DIMACS (for SAT solvers) and SMT-LIB (for SMT solvers) input formats. However, since the outputs are not standardized, in this document we will mention some of those which are handled by the SAT/SMT interfaces of Zot.

- **Minisat** is an open source SAT solver that is available from the web page: http://minisat.se/. It can be compiled from the sources for any platform. The distribution of the zot tool includes the pre-compiled executable of the 2.2 version for the following platforms: Windows (32bit, compiled for the cygwin environment), Mac OS X (10.6), Linux (64bit).
- **Z3** is an SMT solver developed by Microsoft Research. It is available free of charge for the Windows and Linux platforms, and a Mac OS X version has recently been added. It can be downloaded from the website http://research.microsoft.com/en-us/um/redmond/projects/z3/. After installation, the z3 executable must be manually put in the default path (for example by adding the z3 folder to the environment variable PATH on Windows systems, or simply by putting the z3 executable in a folder already in the path, e.g., usr/local/bin on Linux and MacOS systems).
- **Yices** is an SMT solver developed by SRI International. It is also available free of charge for all major platforms (Windows, Linux, Mac OS X). It can be downloaded from the website http://yices.csl.sri.com/. Similarly to z3, after installation the executable must be put into the default path (which in this case simply corresponds to copying the *yices* executable in a folder that is already in the path).

2.2 Installing Zot

The Zot tool can be downloaded from the Zot GoogleCode repository:

http://zot.googlecode.com/files/zot_201202.zip

After downloading the zip file from the Zot GoogleCode repository, to install the Zot tool it is enough to extract the *zot* folder contained in the zip file in some location, then make the *zot* script included in the *zot/bin* subfolder available system-wide by putting it in the default path.

More precisely, the *zot/bin* subfolder contains two scripts:

- *zot*, which is a *bash* script that assumes that the Zot tool was extracted to the */usr/local/zot* folder
- *zot.bat*, which is a Windows batch file that assumes that the Zot tool was extracted to the *C:\zot* folder.

Both scripts assume that the *sbcl* executable is available system-wide in the default path. The *bash zot* script works "as is" for Linux and Mac OS X systems, provided, as mentioned above, that the Zot tool has been extracted to the */usr/local/zot* folder. It also works on Windows systems that have the UNIX-like *cygwin* environment installed (www.cygwin.com), by launching it from the cygwin shell (assuming the cygwin shell sees the *zot* folder as */usr/local/zot*). The *zot.bat* script, instead, works "as is" from the standard command line of Windows system, without needing to install cygwin.

Changing the default installation folder for Zot is possible by modifying the *zot* (resp. *zot.bat*) script, and the support lisp file *start.lisp* (resp. *start_win.lisp*). It is enough to substitute in the two files all instances of */usr/local/zot* (resp. *C:\zot*) with the desired folder.

As mentioned in Section 2.1.2 the *zot/bin* subfolder of the zip file contains the pre-compiled executable file for the minisat SAT solver for the three main platforms: *minisat_win.exe* is the Windows executable compiled under the cygwin environment; *minisat_mac* is the Mac OS X (10.6) executable; *minisat_x64* is the Linux executable (64 bit). The executable corresponding to the chosen installation platform must be renamed simply as *minisat* for Zot to be able to use it. Since the windows executable has been compiled under the cygwin platform, the *cygwin1.dll* is needed to run it if the cygwin environment is not installed, hence the dll has also been included in the distribution of the tool; of course, the user is free to download and recompile the minisat executable for his/her environment of choice.

3 References

[CCC+99] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. ACM TOSEM, 8(1):79-113, 1999.

[DIMACS] http://www.satcompetition.org/2011/format-benchmarks2011.html

[SMT-LIB] http://www.smtlib.org/

[Zot] zot.googlecode.com