SMTS Artifact Description

Matteo Marescotti

August 9, 2018

SMTS is a framework that allows the user to execute, monitor, and guide model-checking and SMT solving in a distributed, high-performance computing environment. This artifact description explains how to use SMTS in a virtual machine. We need to emphasize that the experimental results of the paper are not reproducible in a virtual machine. This is in part due to the number of available CPUs, but in particular to the high memory usage of each model-checking or SMT solving process. However, the user may still fully assess SMTS by using our testing tool *dummy solver* (Sec. B.1), a backend that reports random results, but acts otherwise like a normal solver with respect to the protocol. Alternatively, we successfully run SMTS in a 8-core Mac OSX laptop, executing 4 solvers with lemma sharing.

Username: `smts`

Password: `smts2018`

Working directory: `artifact/smts`

A Introduction

This section provides information on the SMTS set-up, the interaction between the solvers, and how to run SMTS in a single machine.

SMTS consists of five components: the solvers, the lemma server, the server, the client, and the graphical user interface (GUI). The solvers and the lemma server are written in C++; the sources are available in `/src`. The server and the client are written in Python3 available in `/server`, while the GUI is written in NodeJS (Javascript) and is provided in `/gui`.

The executables built from C++ sources are stored in the build directory (by default in `/build`), and include `lemma_server` and the solvers. SMTS currently provides two types of solvers: one based on OpenSMT2 (`solver_opensmt`), and on based on the IC3 engine Spacer (`solver_z3spacer`). While the underlying solver engine is different for the two solvers, their command line interface is the same. The command `./solver_* -h` displays the help of the solver. The reviewer might find useful the argument -s for specifying the server's IP address and port.

The server and the client are respectively smts.py and client.py, both in /server. Both show useful help when executed with the argument -h. The server smts.py is highly configurable through configuration files. The default configuration file is /server/config/default.py. We suggest to not change the default configuration file, but instead provide smts.py with configuration files using the -c argument. The user may provide several configuration files, in which case each one will set the parameters it specifies, leaving everything else unchanged. The current configuration may be checked by running the server with argument -L. When running SMTS with local clients, the arguments -o and -z of smts.py are a convenient way to specify the number of clients (solver_opensmt or solver_z3spacer, respectively). Such approach will not work in a cluster, because the solvers run in different computing nodes. Instead each solver must be run separately, providing the server IP and port through the -s argument.

client.py connects to a running server and can work in two modes: as a Python3 shell or as a system sending problem instances. The former mode allows the user to inspect the current execution by evaluating and executing code while the server is running. The latter mode takes one or more smtlib instances and sends them to the server for solving. Another convenient way to provide instances to the server is to fill them in the files_path list of the provided configuration file. By doing so, smts.py will load them from the file system at startup.

The third Python3 file utils.py lets the user to run the GUI for analysing a past execution (based on the logged events).

B Running SMTS

B.1 The dummy solver.

Every running solver requests a consistent amount of memory and up to a full core of CPU computational resource. For this reason, bugs affecting SMTS when managing a consistent number of solvers might be expensive, both in terms of money (CPU time is expensive) and in terms of time lost in cluster queuing systems. The dummy solver is designed for testing SMTS by simulating several solvers working concurrently. The kinds of bugs the dummy solver can detect are related to protocol compliance, partitioning scheduling among solvers, error and timeout handling.

The dummy solver design is also suitable to make an evaluation of SMTS within a virtual machine possible. The user can assess how SMTS would behave having an arbitrary number of solvers running e.g. in a cluster. Features of particular interest are the functionalities offered by the SMTS GUI, and the partitions scheduling performed by the server.

The Python3 executable /server/dummy_solver.py is designed to simulate an arbitrary number of solvers compatible with any SMT theory. Each solver acts as follows: after receiving an instance from the server it chooses an answer out of SAT, UNSAT or TIMEOUT, based on the weights respectively provided by the arguments `-S`, `-U` , and `-t`. If the chosen answer is TIMEOUT, the solver sleeps forever. Otherwise, if the answer is either SAT or UNSAT, the solver draws a number *t* from a uniform distribution between 0 and the value provided by the argument `-m`, and after *t* seconds, reports the answer.

The dummy solver is useful to evaluate an execution with e.g. 10 solvers on a single core virtual machine. The user can see from the GUI how the resulting tree would look like, visualize the SMT instance, guide the expansion of the partition tree, and also see simulated learned clauses. An example of how to evaluate SMTS using the dummy solver is provided in Sec B.3.

Dummy solver's output. The dummy solver simulates `N` solvers, provided `-n N` as argument. Each simulated solver produces its own output lines:

- `solving [filename][partition]` is printed on the arrival of a new solving request;
- `reporting [status] in [s] seconds` tells immediately when and which status will be sent to the server; and
- `going timeout` means that nothing about the particular instance will be sent to the server.

B.2 The server

The two supported solvers cannot be used simultaneously because they support different SMT theories. In particular OpenSMT supports uninterpreted functions and linear real arithmetics, while Z3 Spacer supports constrained Horn clauses with integers and arrays. To allow the user to evaluate both solvers, we provide two different configuration files, each proving suitable benchmarks for the respective solver. The configuration files `config_{opensmt,spacer}.py` contain some help related to the parameters they set. The configuration files also contain a parameter set with the path to some benchmarks suitable for the respective solver. Those benchmarks are loaded at startup and scheduled for solving as soon as some solvers connects to the server. To run the server and the solvers type one of:

```
$ ./server/smts.py -c config_opensmt.py -l -g -o2
$ ./server/smts.py -c config_spacer.py -l -g -z2
```

WARNING: we discourage from running these commands in a virtual machine because they will result in many CPU and memory intensive processes executing simultaneously, seriously reducing the responsiveness of the system. In particular, the argument `-l` runs the lemma sharing server, `-g` runs the GUI, and `-o2` and `-z2` respectively run two OpenSMT2 and Z3 Spacer processes. To evaluate the GUI and the server on a virtual machine we suggest to use the dummy solver. An example of usage is provided in Sec B.3.

Server's output. Each output line consist of the current time, a level (`INFO`, `WARNING`, or `ERROR`), and a message.

- Each `filename` provided in the `files_path` is loaded by the server and an `INFO` message `new instance "[filename]"` is printed.
- The server attempts to solve instances one by one, until the instance either times out or gets solved. The message `solving instance "`*filename*`"` tells which instance SMTS is currently solving.
- The message `GUI running on 8080` tells the user that the GUI is available through a web browser at `http://127.0.0.1:8080`.
- The message `new <[sname] at [ip:port] idle>` is printed on each new incoming solver connection.
- A solver `sname` reports on instance `fname` by printing a message

  ```
  <[sname] at [ip:port] [fname][part]>:  [report_msg].
  ```

  The partition `part` is described by the index path starting from the root `[]` of the partition tree.

B.3 Examples

The benchmarks are stored in `../benchmarks_{opensmt,spacer}/`. Each folder has a sub-folder `easy/` containing some easy-to-solve benchmarks. These benchmarks are currently listed in the configuration files and will therefore be loaded by SMTS server at startup. We encourage the reader to change the configuration files.

The following example will simulate 5 solvers, each connecting to `smts.py` running locally on the default port, never returning SAT, with 70% probability of returning UNSAT after a simulated run of at most 150 seconds, and with 30% probability of timing out. Note that the weights need not sum up to a particular value, like 10 in the current example.

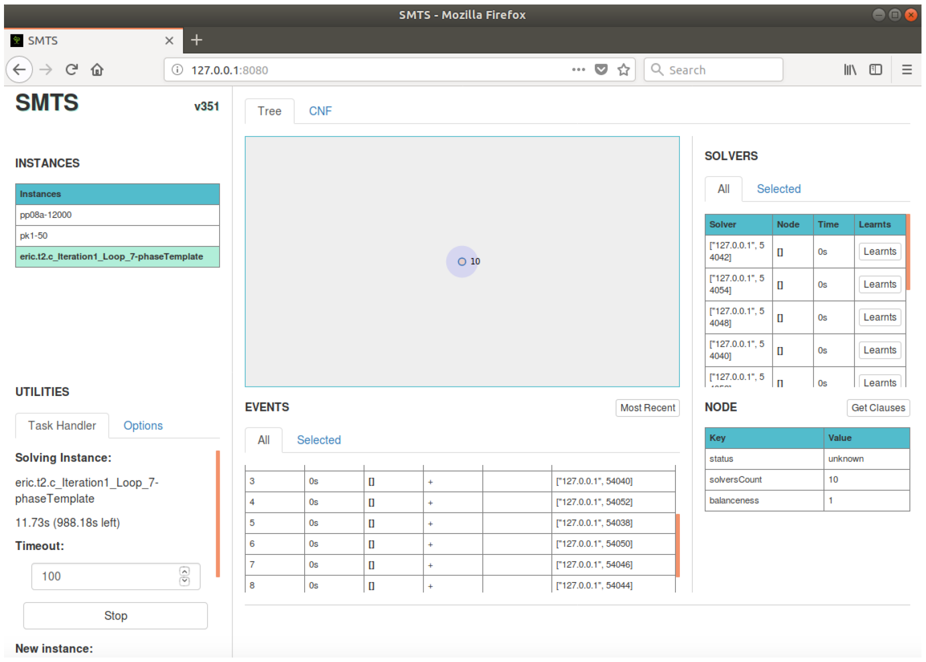
```
$ ./server/smts.py -c config_opensmt.py -g
# on another terminal
$ ./server/dummy_solver.py -S0 -U7 -t3 -m150 -n5 -s127.0.0.1:3000
```

Now the GUI is available by opening the browser at `127.0.0.1:3000`; a screenshot is provided in Fig. 1

The left column provides a list of the solved instances on the top, and information about the currently solving instance (e.g. time spent, time until timeout etc.) on the bottom. The central column consists of the parallelization tree view on the top and the list of solving-related events on the bottom. The number near each node in the tree is the amount of solvers working on the partition which that node represents. Finally, on the right there is a list of currently working solvers on the top, and information about the selected node on the bottom.

The GUI features involve:

- Clicking an event makes the tree to *go back in time* showing how the tree was at that time.
- Double click on a tree node triggers partitioning. Note that partitioning is triggered by the server's config option `partitioning_timeout`, in seconds.
- **Only for server running with** `-c config_opensmt.py`. Clicking the bottom Get Clauses will make the CNF graphical representation to appear in the CNF tab. This feature is only supported by OpenSMT2 solver, and the dummy solver offers that functionality relying on it. For this reason if the dummy solver is simulating solving an IC3 instance (those from `confing_spacer.py`), an error will be returned.
- **Only for server running with** `-c config_opensmt.py`. Clicking the button Learnts will make the CNF graphical representation to appear together with the learned binary clauses displayed by green edges.



Instances
pp08a-12000
pk1-50
eric.t2.c_Iteration1_Loop_7-phaseTemplate

Solving Instance:

eric.t2.c_Iteration1_Loop_7-phaseTemplate

11.73s (988.18s left)

Timeout: 100

New instance:

3	0s	[]	+		["127.0.0.1", 54040]
4	0s	[]	+		["127.0.0.1", 54052]
5	0s	[]	+		["127.0.0.1", 54038]
6	0s	[]	+		["127.0.0.1", 54050]
7	0s	[]	+		["127.0.0.1", 54046]
8	0s	[]	+		["127.0.0.1", 54044]

Solver	Node	Time	Learnts
["127.0.0.1", 54042]	[]	0s	Learnts
["127.0.0.1", 54054]	[]	0s	Learnts
["127.0.0.1", 54048]	[]	0s	Learnts
["127.0.0.1", 54040]	[]	0s	Learnts
["127.0.0.1", 5	[]	0s	Learnts

Key	Value
status	unknown
solversCount	10
balanceness	1

Figure 1: A GUI screenshot