VERIFIER, An efficient and computational tool for analysis of message passing programs ⋆

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Gaurav Saini[2], and Subham Modi[3]

[1] Uppsala University, Sweden
[2] Indian Institute of Technology, Ropar
[3] Indian Institute of Technology, Kanpur

Abstract. VERIFIER is a first freely available, open source, computational and an efficient tool for the verification of message passing programs communicating in an asynchronous manner over unbounded imperfect channels. It works on an approach called as *bounded-phase* reachability analysis which verifies the reachability problem of protocols having their phase bounded. Each process in such programs can perform a computation in which the number of phases is bounded. In each phase, a process can perform either send transitions or receive transitions (but not both). However, this doesn't limit the number of context switches between two process but just the number of alternations between receive and send transitions.Currently, VERIFIER can decide reachability problem for three types of channel semantics, namely *lossy, stuttering and unordered* channels. These channels allow messages inside them to be lost, dublicated and re-arranged respectively. It translates bounded phase reachability problem into the satisfiability of quantifier-free Presburger formulas for each of the above mentioned semantics.Since the number of channels, messages and the architecture depends on the protocol used, the verification task is non-trivial and requires uniform as well as efficient translation of bounded phase reachability problem into quantifier-free Presburger formula.

1 Introduction

VERIFIER is an open source, computational and an efficient tool for the verification of message passing programs communicating in an asynchronous manner over unbounded imperfect channels. It is available freely at [3].

The verification problem such as reachability problem is undecidable for programs having perfect channels even if the number of states in each process are finite but it becomes decidable if we have lossy, stuttering or unordered channels [1]. However, to decide the reachability problem in later case we have high complexity obstacles. Thus in order to avoid this, one useful approach, *context bounding* [7], was proposed recently. This idea limits the number of context switches between two processes because of which we have a trade off between the extent of verification and computational complexity.

⋆ This research was in part funded by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the National Science Council of Taiwan project no. NSC-101-2221-E-001-007, and the CENIIT research organization (project 12.04).

VERIFIER verifies the reachability problem of programs or protocols in which each process performs a computation with the phase bounded by some natural number. In each phase, a process can perform either send transitions or receive transitions (but not both). The transition consisting of no operation, but just the change of states, can be performed in either of these phases.

It uses the framework and prototype descibed in [2] which translates bounded phase reachability problem into the satisfiability of quantifier-free Presburger formula in polynomial time. Thus, we can use one of the various SMT solvers which, efficiently, comment on the satisfiabilty of the generated quantifier-free Presburger formula for all the above mentioned semantics. It is created in such a way that it can be applied on a number of message paasing program with promising results. More precisely:

- VERIFIER is an open source [3] verification tool that analyse reachability problem in message passing programs having their phase bounded
- it is complete and sound for protocols having finite number of channels and messages in which processes communicate via unbounded imperfect channels
- it converts bounded phase reachability problem into quantifier-free presburger formula and then call SMT Solver to output the satisfiability results.
- it performs reachability analysis for faulty protocols also which differs from therir original one at some or many transition.
- it provides user with the computational time used both by the constraint generation as well as SMT solver to analyse the complexity of the provided protocol. Details regarding these two extractions are given in section 2.
- it provides user with the satisfiability results in an user-friendly manner by generating a separate tex file. More information regarding this can be found within the tool [3].

Targeted users VERIFIER can be used by these group of users :

1. Various researchers and Computer scientists can use the freely available and open source code (both for using online and downloading) of VERIFIERto optimise it further or improve the implementation techniques thereby extending its features further (some of its extension may include use of some other efficient SMT-solver or using a different data structure for handling the protocols). It can also be used to compare or draw an analogy between various message passing verification tool developed so far.
2. Professors and teachers of computational complexity and verification group can use VERIFIER to analyse reachability of a particular state in any complex but finite message passing programs (working under the above defined restrictions) which are difficult to visualize directly. They can even illustrate the effects of protocal and reachability settings provided on the running time of the tool.
3. Developers who are working on similar verification problem can use VERIFIER to verify their results. It can also be used to check faultiness of a particular protocol by verifying the rechability of an Invalid state.

2 Verifier Tool

VERIFIER performs two different levels of extraction in order to analyse the reachability problem for a given program.

Verification is achieved by taking the examples from [5] which are further described in [6] and [8]. Bounded Retransmission Protocol is also adapted from [4]. Protocols are used in the format of `xml-files` present within the tool repository inside the 'Includes' folder. The protocols, thus, can be modified in a simple manner by changing the fields in xml-files.

2.1 Xml to Automata

The first task of Verifier is to translate the protocols defined in `xml-files` into *Non-Deterministic Finite Automata* (NFA). In order to achieve this, it takes xml-file path of the Protocol as an input and then uses C++ library of `lemon` to translate the protocol into NFA as described below:

```
<rule id="Q0__ack1__INBOUND">
  <pre>
    <current_state>Q0</current_state>
    <received_message>ack1</received_message>
    <channel>c1</channel>
  </pre>
  <post>
    <send_message>mesg0</send_message>
    <next_state>Q1</next_state>
    <channel>c1</channel>
  </post>
</rule>
```

Fig. 1. An example of xml code for ABP Protocol

The above rule adds two transitions from the state `Q0` to the state `Q1`. First transition defines the rule of receiving the message `ack1` from the channel `c1` whereas second transition defines the rule of sending the message `mesg0` into the channel `c0`. Each process in a protocol contain one or more such rules which in together defines the automata for that process within the given protocol.

Now, since the protocols have their phase bounded and each phase contain either send or receive transitions (but not both), we make two automata for each process, one containing all except the receive transitions (send copy of that process) and the other containing all except the send transitions (receive copy of that process).
In this way, we have constructed *2*Number of Process* automata for the given protocol.

2.2 Generating Constraint from the above constructed Automata

Verification of protocol for reachability problem is analysed by generating *Presburger Formula* from the automata constructed as shown in `figure 1` and then using the help of modern SMT solver namely *Z3 theorem prover*. Detailed information regarding the framework showing the translation of reachability for *bounded-phase-automata* into the satisfiability of *quantifier-free Presburger formulas* can be found in [2].

In order to generate quantifier-free Presburger formula, certains variables of a particular *sort* have to be defined and thus, for all the transitions present within each automata, we'll introduce a number of variables as shown below:

Variable name	Variable code
Index-variable	i-var
Occurence-variable	o-var
Match-variable	m-var

Table 1. Variables associated with each transitions

Variable declaration and defination are explained briefly in [2].

For taking *bounded-phase-automata* into account, VERIFIER generates variables for both send and receive copy of each process and then duplicates them *k* times (where k is some natural number denoting the bound for the number of phases within each process) which are then further used to generate Presburger formulas. In this way we have ignored the inefficient process of making multiple copies for each process.

If the result of *Z3 theorem prover* for these set of formulas (one of them being displayed in `figure 2`) is satisfiable (sat), then the *Bad State* is reachable and we have an UNSAFE (U) condition otherwise we have a SAFE (S) condition, ie, *Bad State* is probably not reachable for the given bound.

$$(\texttt{occ}(t) = 1) \wedge (\texttt{occ}(t') = 1) \wedge (\texttt{index}(t) < \texttt{index}(t')) \implies (\texttt{match}(t) < \texttt{match}(t')).$$

Fig. 2. An example of a *quantifier-free Presburger formulas*

3 Implementation

VERIFIER tool is implemented in C++ language with the help of `lemon` and `pugixml` library. Tool is programmed in an user friendly manner and is, thus, easy for any further extension or use.

Optimisations. Various optimisation techniques were implemented to increase the efficiency of the reachability problem.

- *Ignoring multiple copies for each process.* Instead of making 'k' copies for each process (where k is some natural number denoting the bound for the number of phases within each process), we make only two copies per process (send and receive copy) as described in section 2.1.
- *Removal of strongly connected component.* We evaluate all the sets of strongly connected components in the send copies of each processes. From these sets, taken one at a time, we remove all the states and add two new states having transition between them over all the operations associated with the transitions within that particular set. Out of these two added states, initial state will now be the target state for all the transitions entering into this set previously and the other state (final state) will be the source state for all the transition leaving from this set. Therefore, each send copy of each processes is optimised in this way to reduce the number of constraints formed.

4 Experimental Results

Table 2 displays the results of running VERIFIER tool on the set of examples included whereas in Table 3 we show the results of running VERIFIER on the protocols which are intentionally modified to cause an error. All examples are supplied in the form of `xml-files`.

Both the tables should be interpreted as follows :

Column P lists the name of protocol under analysis.
Column Bad lists the name of the process followed by the state in that process which we are trying to reach.
Column Sem lists the channel semantics used for the specific test.
Column channel lists the type of channel used in the message passing.
Column Constraint Generation, SMT and Total lists the time taken to generate constraint, time taken by the SMT solver to analyze the formula and the total time of the analysis respectively.
Column Assert lists the number of assertions fed to the SMT solver
Column Ph. lists the bound on the number of phases, and finally,
Column Res lists the result of the analysis. The results are listed as **U** for Unsafe or Bad state is reachable, ie, **sat** or **S** for possibly safe or Bad state is not reachable, ie, **unsat**.

Results displayed here are for few protocols only. Rest of the results are available online at [3] in the 'verifier_result' folder which is inside the 'verifier-master' directory.

P	Bad	Sem	Channel	Const. gen.	SMT	Total	Assert	Ph.	Res
ABP	RECEIVER Invalid	UCS	process	0.07 sec	74 sec	74.07 sec	4025	4	U(sat)
ABP	RECEIVER Invalid	LCS	process	0.04 sec	1 sec	1.04 sec	2519	3	S(unsat)
ABP	RECEIVER Invalid	SLCS	process	0.03 sec	2 sec	2.03 sec	2519	3	S(unsat)
BRP	RECEIVER Invalid	UCS	process	0.3 sec	2 sec	2.3 sec	23461	3	S(unsat)
BRP	RECEIVER Invalid	LCS	process	0.28 sec	2 sec	2.28 sec	23461	3	S(unsat)
BRP	RECEIVER Invalid	SLCS	process	0.28 sec	1 sec	1.28 sec	23461	3	S(unsat)
STP	A Invalid	UCS	process	0.03 sec	4 sec	4.03 sec	2354	6	U(sat)
STP	A Invalid	LCS	process	0.02 sec	0 sec	0.02 sec	1348	4	S(unsat)
STP	A Invalid	SLCS	process	0.02 sec	0 sec	0.02 sec	1348	4	S(unsat)

Table 2. Verification Results for examples from [6] and [5]

P	Bad	Sem	Channel	Const. gen.	SMT	Total	Assert	Ph.	Res
ABP_F	RECEIVER Invalid	SLCS	process	0.04 sec	1 sec	1.04 sec	1275	2	U(sat)
ABP_F	RECEIVER Invalid	UCS	process	0.04 sec	1 sec	1.04 sec	1275	2	U(sat)
SlidingWindow_F	RECEIVER Invalid	SLCS	process	0.02 sec	0 sec	0.02 sec	913	1	U(sat)
SlidingWindow_F	RECEIVER Invalid	UCS	process	0.03 sec	0 sec	0.03 sec	913	1	U(sat)
Synchronous_F	B Invalid	SLCS	process	0.02 sec	0 sec	0.02 sec	713	3	U(sat)
Synchronous_F	B Invalid	UCS	process	0.02 sec	0 sec	0.02 sec	713	3	U(sat)

Table 3. Verification Results for buggy (faulty) examples

References

1. P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS*, 1993.
2. P. A. Abdulla, M. F. Atig, and J. Cederberg. Analysis of message passing programs using smt-solvers. Uppsala University.
3. P. A. Abdulla, M. F. Atig, S. Modi, and G. Saini. https://github.com/vigenere92/Verifier.
4. P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using forward reachability analysis for verification of lossy channel systems. In *FMSD*, page 25(1):39–65, 2004.
5. A. P. M. Jr., A. Ravn, J. Srba, and S. Vighio. csv2uppaal. https://github.com/csv2uppaal.
6. A. P. Marques, A. P. Ravn, J. Srba, and S. Vighio. Tool supported analysis of web services protocols. In *TTCS*, pages pages 50–64, University of Oslo, 2011.
7. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, 2005.
8. A. P. Ravn, J. Srba, and S. Vighio. Modelling and verification of web services business activity protocol. In *TACAS*, LNCS, pages pages 357–571. Springer, 2011.

A Installation

This appendix is intended to guide the reader to use of VERIFIER . It will first describe installation of VERIFIER , then give a tutorial on using the VERIFIER . VERIFIER can be downloaded from https://github.com/vigenere92/Verifier. The sources are available as a tar ball via the "Downloads" section (recommended) as well as by git clone.

A.1 Requirements

- A C++ compiler supporting C++11. For example g++ version 4.6 or higher.
- Z3 SMT 2.0 Solver.
- Lemon Graph Library.
- pdflatex (Optional).

The Z3 SMT Solver can be downloaded from http://z3.codeplex.com. It should be installed in the bin directory, so that it can be called using "z3 -smt2 filename" from the terminal. Lemon Graph Library can be downloaded from http://lemon.cs.elte.hu. pdflatex is an optional feature. It is required only for creating pdf's for the results for various runs.

A.2 Installation

The VERIFIER Tool can be installed by using the following procedure.

- Copy the verifier_result folder from the Verifier directory to your home folder. This is done to make the installation of the VERIFIER tool consistent irrespective of the location where parent directory is downloaded.
- Using terminal, go to the location of the Verifier folder and execute the following commands.

```
$ touch NEWS README AUTHORS ChangeLog
$ autoreconf --force --install
$ ./configure
$ make
$ sudo make install
```

B VERIFIER Tutorial

This section gives you a short tutorial on the usage of VERIFIER .

B.1 Creating the Settings File

VERIFIER takes path to a Settings file as an input. This file contains the information on the path to the XML file, semantics of the channels, bound, the states for which the reachability is to be checked and the type of channels to be formed for calculating the reachability.The format of Settings file is as follows:

file	path_to_xml_file
semantics	channel_semantics
bound	bound_for_the_processes
bad	bad_process bad_state
channel_type	type_of_channel

path_to_xml_file is path to the xml-files of protocols (message passing programs) present inside the Verifier directory. So before calling the verifier tool, change `path_to_xml_file` argument inside the 'Settings.txt' file.
By default, it is present in the 'Includes/Protocols' folder which is inside the 'Verifier directory'.

B.2 Description of Settings File

- **file**: It refers to the location of the XML file to be verified.
- **semantics**: It refers to the semantics of the channels. It can be lossy, stuttering or unordered.
- **bound**: It refers to the number of bounds for the automata.
- **bad**: It takes pairs of values. First is the name of the process in which the reachability is to be checked and the second is the set of states for which reachability is being checked.
  NOTE: *Bad state for more one process can also be entered one after the other in the same format as diplayed in the example below.*

  Example:

  ```
  bad SENDER Q0 Q1 RECEIVER INVALID
  bad SENDER Q0 RECEIVER INVALID
  bad RECEIVER INVALID
  ```

- **channel_type**: It refers to the type of channel. It can be :

  `prefix` : For making channels based on the first alphabet of the messages.
  `process`: For making one channels for each process.
  `xml` : For making channels as specified in the xml file.

Figure 3 shows a sample Settings file for the Alternating Bit Protocol with its XML file(ABP.xml) in the default 'Includes/Protocols' folder.

file /home/rator/Verifier/Includes/Protocols/ABP.xml
semantics lossy
bound 3
bad RECEIVER INVALID
channel_type process

Fig. 3. An example of Settings file

In case the path to XML file for the protocol is incorrect in the Settings file, you will get the following error message:

```
----------
----------
; loading xml file
Xml File not found!
Exiting
```

If so, please make sure the path to the XML file for the protocol is correct.

B.3 Running the VERIFIER for an Automata

The VERIFIER can be called from the command line using 'verifier' followed by `Path to the Settings file` which is expected as the first argument.
By default, it is present in the 'src' folder which is inside the 'Verifier directory'.Thus Verifier tool can be called as (if Settings file path is not changed)

```
verifier src/Settings.txt
```

In case the Settings file is not found, you will receive the following error message:

```
Settings File not found
```

If so, please make sure the path to Settings file is correct with respect to the current directory.
When the VERIFIER is called for a Settings file, it prints the results in 4 sections. In the first section it prints step by step various operations that are being applied to the Automata. Fig. 4 shows the screen dump, when VERIFIER is run on the Settings file mentioned in Fig. 3.

```
————-
————-
; loading xml file

; adding states for process SENDER
; adding transitions for process SENDER
; adding states for process RECEIVER
; adding transitions for process RECEIVER
; removing send loops
; declaring index and occurence variables for send transitions
; declaring index, occurence and match variables for receive transitions
; declaring index and occurence variables for eps transitions between two phases
; declaring index and occurence variables for send transitions
; declaring index, occurence and match variables for receive transitions
; declaring index and occurence variables for eps transitions between two phases
; adding in and out flow constraint
; Initial state constraint
; Final state constraint
; adding computation order constraint and ensuring connectivity
; adding in and out flow constraint
; Initial state constraint
; Final state constraint
; adding computation order constraint and ensuring connectivity
; declaring distinct index condition for each pair of non unique transitions
; enforcing matching of each occuring receive transitions
; enforcing computation order for transitions pair per channel
```

Fig. 4. Operations applied to the Automata

In the second section, it prints statistics on the number of states and transitions added for the Automata. Fig.5 shows the screen dump for the statistics on Automata.

```
————-AUTOMATA STATISTICS————-
Total Number of non-unique transitions: 249
Total Number of unique transitions: 6
Total Number of states: 156
Number of Assertions : 2519
```

Fig. 5. Automata Statistics

In the third section, it prints the time taken for generation of the preburger formula and the time taken by Z3 SMT solver.
Fig.6 shows the screen dump for the time statistics.

```
————-TIME STATISTICS————-
Automata and Constraint genertaion time for generating Presburger Formula: 0.05 seconds
SMT Solver time for generating output by Presburger_formula theorem Prover: 1 seconds
Total time elapsed: 1.05 seconds
```

Fig. 6. Time Statistics

In the fourth and the final section, it prints the result, whether the given state is reachable or not. Fig. 7 shows the screen dump for the result.

```
————-RESULT————-
Protocol_name = ABP
Bad_state_name = RECEIVER Invalid
Semantics = LCS
Channel_Type = By process
Constraint Generation Time = 0.05 sec
SMT Time = 1 sec
Total Time = 1.05 sec
Assertions = 2519
Bound = 3
Result = S(unsat)
Thus, the bad State is not reachable for lossy semantics within the given bound : 3
————-
————-
```

Fig. 7. Reachability Result

B.4 Other Optional Arguments

A pdf containing the results of the Verifier can also be generated if pdflatex is installed in the system.This can be done using `new` or `old` arguments in addition to the one which we were using before. It can be used as follows :

1. Using the new option:

A new pdf containing the results can be generated using this option. The pdf is placed in the `verifier_result/Result` folder in the home directory. It is used as follows :

```
verifier path_to_settings_file new
```

2. Using the old option:
The result for this run of the Verifier is appended to the previously genereated pdf. The pdf is placed in the `verifier_result/Result` folder in the home directory. It is used as follows :

```
verifier path_to_settings_file old
```