TOPICS - Translation Of Programs Into Counter Systems

User's manual

Laboratoire Spécification et Vérification
CNRS UMR 8643
École Normale Supèrieure de Cachan
61, avenue de président Wilson
F-94245 CACHAN Cedex

EDF R&D
Département STEP
quai Watier
Chatou

Arnaud Sangnier

Contents

Introduction

TOPICS is a tool dedicated to the verification of C-like programs. The particularity of this tool lies in the fact that it can deal with programs manipulating dynamic data structures. In this first version, TOPICS can treat programs working over singly linked lists, but this feature will be extended to more complex data structures. The main feature of TOPICS is the translation of C-like programs into a bisimilar counter system, where counter systems correspond to finite automata extended with (unbounded) integer variables. This translation allows to use tools which verifies counter systems in order to check that the given program do not realize errors as memory violation, memory leakage or out of bound error.

The translation implemented in TOPICS corresponds to the one presented in [BFLS06]. Note that a similar translation has also been proposed in [BBH$^+$06]. The programs given in input of TOPICS are written in a syntaxic fragment of the C programming language. These programs can manipulate integer variables, singly linked lists and arrays of integers and of lists. Whereas there is no restriction on the size of the manipulating lists, the arrays have to be of finite size. The user has also the possibility to give an initial configuration to TOPICS, in which he will characterize the initial state of the memory he wants the programs to begin with.

1 Using TOPICS

In this section, we explain how to use TOPICS.

1.1 Starting TOPICS

TOPICS is distributed a jar file for Java. It has been developed under java 1.4.2, but it should normally work with more recent version of java. Furthermore it is recommended to use a version of java developed by Sun Microsystems. Note that TOPICS use the tools JFLEX [JFl] and CUP [Cup] to parse the input files.

To run TOPICS, use either :

```
java -jar TOPICS.jar filename functionname confinitname
```

or

```
java -jar TOPICS.jar filename functionname
```

In this two case, *filename* is the name of the C file containing the program and *functionname* is the name of the function belonging to the program whose behavior will be translated into a counter system. As said previously, the user has the possibility to give a file containing a description of the initial configuration, this corresponds to the first case, whereas in the second case no initial configuration is given. The syntax for the file containing the program (*filename*) and for the file containing the initial configuration (*confinitfile*) are given in section 3.

1.2 Output of `TOPICS`

`TOPICS` translates the behavior of a function belonging to a C-like program into a counter system. The produced counter system is available in three different formats :

1. in the `dot` format, so that a graphical representation of the counter system can be obtain using the tool `GRAPHVIZ`,
2. in the format of the tool `FAST`,
3. in the format of the tool `ASPIC`.

`GRAPHVIZ` is a tool which , from a structural description of a graph, generates a graphical representation of this graph. This graphical representation can then be exported in different formats. This tool is freely available at the following URL : `http://www.graphviz.org/`.

`FAST`is a tool which automatically verifies counter systems computing their exact reachability set from an initial configuration. Note that since this exact computation cannot always be performed, it might be the case that the algorithm implemented in `FAST`does not terminate, however this tool has proved to be really efficient on many practical case studies. `FAST`is freely available at the following URL : `http://www.lsv.ens-cachan.fr/fast/`. The main characteristic of `FAST`are described in [BFLP03, BLP06, BFLP08].

`ASPIC`is a tool which also verifies counter systems. This tool uses the method of abstract interpretation [CC77] and computes an overapproximation of the reachability set of a given counter system. The algorithm which is implemented in this tool manipulates linear relations and is described in [GH06].

The benefif of this tool is that its computation always terminates, in opposite to the one of `FAST`, but it can be the case that this too cannot solve a reachability problem because of the realized overapproximations and it returns then the answer “I don’t know”. This tool is freely available at the following URL : `http://laure.gonnord.org/pro/aspic/aspic.html`

1.3 How does `TOPICS` work

Given a C-like program, the name of a function of the program and an optional initial configuration, `TOPICS` realizes the following operations :

1. It builds the Control Flow Graph of the function given in input. `TOPICS` produces a representation in the `dot` format of this control flow graph, so that the user can visualize it.

2. From the initial configuration, `TOPICS` builds an initial configuration for the produced counter system.

3. Using the algorithm of translation presented in [BFLS06], `TOPICS` builds a counter system, which is bisimilar to the function given in input. It produces a file in the `dot` format so that the counter system can be visualized. It also produces files in the format of `FAST` and `ASPIC`. The produced counter system might have four special control states, one for each possible errors :

   (a) the control state `SegF` which is reachable if the initialized program realizes a memory violation,

   (b) the control state `MemL` which is reachable if the inititalized program realized a memory leakage,

   (c) the control state `OOBound` which is reachable if the inititalized program realized an overflow of indexes when working over an array,

   (d) the control state `Undef` which is reachable if the initialized program tests during its execution the value of a variable which has not been previously defined.

The file produced into the format of `FAST` contains then a property to check which is whether one of this state is reachable or not. `TOPICS` can generate more than one file for `ASPIC`, it generates one file for each possible error. In fact, the syntax of `ASPIC` only allows to give an initial configuration and a bad configuration, for which `ASPIC` check if it is reachable or not, that is why for each of the special control state present in the generated counter system, `TOPICS` produces a file to the format of `ASPIC`.

Note that when a user wants to verify a given program with `TOPICS`, the verification process is done in two steps. The first step consists in giving the program to `TOPICS` which translates it into a counter system. After this step, some properties might already been established. For instance, if the control state `SegF` is not present in the counter system, then it is sure that the program do not realize any memory violation and there is no need to analyze the counter system with `FAST` or `ASPIC` to test this. In the other case, if one the special control state is present in the counter system, then the user should use `FAST` or `ASPIC` to check if this control state is reachable or not.

2 Example

In this section, we present a simple example of programs to show how `TOPICS` can be used. The figure 1 gives an example of a program written in the syntax of `TOPICS`. Assume this program is store in the file `mainReverse.c`. In this program there are two functions, the function `reverse` and the function `main` which calls the function `reverse`. Note that in the body of the function `main` we use the special symbol `any` to do a non deterministic loop in order to build single linked lists of any size. The user can then choose to analyze one of these two functions.

For the analysis of the function `main` there is no need to give an initial configuration, so the user can executing `TOPICS` doing :

```
java -jar TOPICS.jar mainReverse.c main
```

This will produce the following files :

- `main_automaton.dot` which contains the control flow graph of the function `main`, this control flow graph is given by the figure 2,
- `mainCA.dot` which contains the produced counter system in the `dot` format,
- `main.fst` which contains the produced counter system in format of `FAST`,

```
typedef struct List {
  struct List *next;
}* Liste;

void main(){
  Liste y,z;
  y=NULL;
  while(any){
    z=malloc(sizeof(struct List));
    z->next=y;
    y=z;
  }
  y=reverse(z);
}

Liste reverse(Liste x){
  Liste u,v;
  u=NULL;
  while(!(x==NULL)){
    v=x;
    x=x->next;
    v->next=u;
    u=v;
  }
  return u;
}
```

Figure 1: A C program written in the syntax of `TOPICS`

- `main_Undef_aspic.fst` which contains the produced counter system in format of `ASPIC` to test if the `Undef` state is reachable,
- `main_MemL_aspic.fst` which contains the produced counter system in format of `ASPIC` to test if the `MemL` state is reachable.

Hence `TOPICS`tells us that the function `main` does not realize any memory violation or out of bound error, but the program might realizes a memory leakage or an error due to a test of an undefined variable. in fact, if the loop in the `main` function is never done, then no list is created, and when it calls the function `reverse` it calls it on an undefined list which realizes an error. To test this, the user has to use the tools `FAST` or `ASPIC` with the corresponding files.

The user has also the possibility to analyze the `reverse` function with an initial configuration. For instance, it could use the following initial configuration which characterizes any list of even length :

```
counter k;
abstract node n[k];
abstract node n2[k];
pointer x->n;
succ n=n2;
succ n2=NULL;
```

In fact, with this description, we say that the initial configuration contains two abstract nodes which corresponds to k real nodes, and that the variable x points to the first node. So the entire list has length $2k$ where k is a strictly positive integer. If this configuration is described in a file `conf.init`, then the use launch `TOPICS` with the command :

```
java -jar TOPICS.jar mainReverse.c reverse conf.init
```

3 Syntax of the input files

As we have said `TOPICS` takes two files in input :

1. a program written in a syntaxic fragment of the C programming language, and,
2. a file which contains a description of the initial configuration.

We now give the syntax that should satisfy these two files.

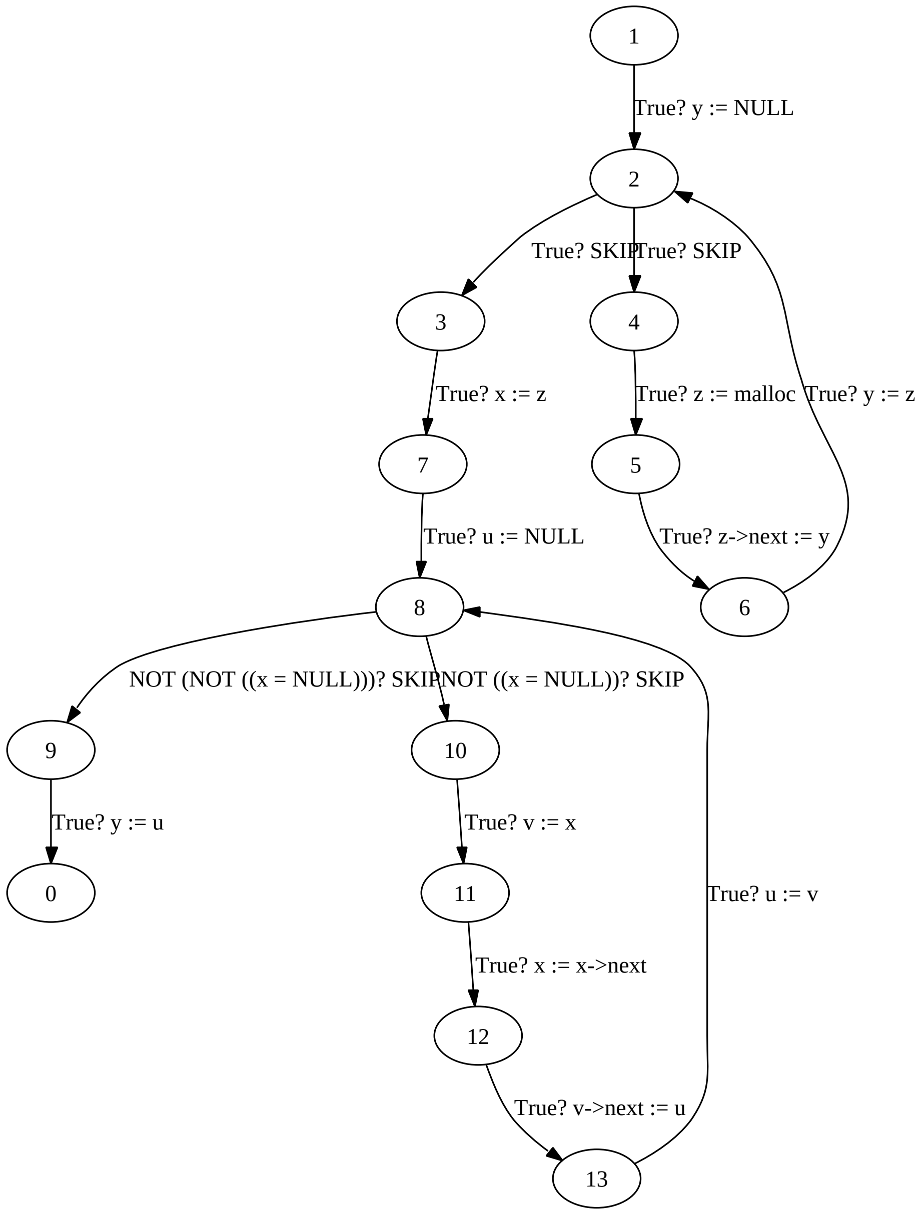

Figure 2: Control flow graph of the function `main` of the program of the figure 1

3.1 Syntax of the programs

In the file corresponding to the program to be verified, the user can declare some types of data structures, some global variables and can give the the definition of functions. The syntax of the programs which can be analyzed with `TOPICS` is given by the figure 3 to 5.

$$
\begin{array}{rcl}
\textit{program} & ::= & \left\{ \textit{declaration} \right\}^{*} \\
\textit{declaration} & ::= & \textit{type-declaration} \\
 & | & \textit{var-declaration} \\
 & | & \textit{function-declaration} \\
 & | & \textit{function-definition} \\
\textit{type-declaration} & ::= & \underline{\textbf{typedef}}\ \textit{type-name}\ \underline{*}\ \textit{identifier}\ \underline{;} \\
 & | & \underline{\textbf{typedef}}\ \underline{\textbf{struct}}\ \textit{identifier}\ \underline{\{}\ \left\{ \text{ struct-field } \right\}^{*}\ \underline{\}}\ \underline{*}\ \textit{identifier}\ \underline{;} \\
\textit{struct-field} & ::= & \textit{type-name identifier}\ \underline{;} \\
 & | & \underline{\textbf{struct}}\ \textit{identifier}\ \underline{*}\ \textit{identifier}\ \underline{;} \\
\textit{type-name} & ::= & \underline{\textbf{int}} \\
 & | & \textit{identifier} \\
\textit{var-declaration} & ::= & \textit{type-name identifier} \left\{ \underline{,}\ \textit{identifier} \right\}^{*} \underline{;} \\
\textit{function-declaration} & ::= & \textit{return-type identifier}\ \underline{(} \left[\textit{fpar} \left\{ \underline{,}\ \textit{fpar} \right\}^{*} \right] \underline{)}\ \underline{;} \\
\textit{return-type} & ::= & \textit{type-name} \\
 & | & \underline{\textbf{void}} \\
\textit{fpar} & ::= & \textit{type-name identifier} \\
\textit{function-definition} & ::= & \textit{return-type identifier}\ \underline{(} \left[\textit{fpar} \left\{ \underline{,}\ \textit{fpar} \right\}^{*} \right] \underline{)}\ \textit{block}
\end{array}
$$

Figure 3: Declarations

Except the use of the special symbol **any**, this syntax corresponds to a restriction of the language C. As we have seen with the previous example, we have introduced this symbol in order to allow the user to implement some non-deterministic tests, which can be useful to give to `TOPICS` a program which corresponds to an abstraction of a more complex program. In fact, when this symbol is used inside a guard, this means that one does not know whether the guard is satisfied or not, and `TOPICS` hence supposed it can be true or false.

With this syntax, a user is able to declare some data structures more complex

$$\begin{array}{rcl}
\textit{block} & ::= & \underline{\{}\ \{\ \textit{statement}\ \}\ \underline{\}} \\
\textit{statement} & ::= & \textit{var-declaration} \\
 & | & /\text{* empty */}\ \underline{;} \\
 & | & \textit{lvalue-expression}\ \underline{=}\ \textit{rvalue-expression}\ \underline{;} \\
 & | & \textit{identifier}\ \underline{=}\ \underline{\textbf{malloc}}\ \underline{(}\ \textit{malloc-expression}\ \underline{)}\ \underline{;} \\
 & | & \underline{\textbf{free}}\ \underline{(}\ \textit{identifier}\ \underline{)}\ \underline{;} \\
 & | & \left[\ \textit{lvalue-expression}\ \underline{=}\ \right]\ \textit{identifier}\ \underline{(}\ \left[\ \textit{term}\ \left\{\ \underline{,}\ \textit{term}\ \right\}^{*}\ \right]\ \underline{)}\ \underline{;} \\
 & | & \underline{\textbf{break}}\ \underline{;} \\
 & | & \underline{\textbf{continue}}\ \underline{;} \\
 & | & \underline{\textbf{goto}}\ \textit{label}\ \underline{;} \\
 & | & \underline{\textbf{return}}\ \left[\ \textit{term}\ \right]\ \underline{;} \\
 & | & \underline{\textbf{if}}\ \underline{(}\ \textit{boolean-expression}\ \underline{)}\ \textit{statement} \\
 & | & \underline{\textbf{if}}\ \underline{(}\ \textit{boolean-expression}\ \underline{)}\ \textit{statement}\ \underline{\textbf{else}}\ \textit{statement} \\
 & | & \underline{\textbf{while}}\ \underline{(}\ \textit{boolean-expression}\ \underline{)}\ \textit{statement} \\
 & | & \textit{label}\ \underline{:}\ \textit{statement} \\
 & | & \textit{block}
\end{array}$$

Figure 4: Instructions

than single linked lists and can also define some recursive functions, anyway `TOPICS` check if the input program satisfied some conditions such as :

- The program has at most three different types of data structures :
    1. A type to define single linked lists in which each cell only contains the pointer to the successor cell.
    2. A type to declare arrays of integers.
    3. A type to declare arrays of single linked lists.
- The defined functions should not be recursive.

Whenever one of these conditions is not satisfied, `TOPICS` detects it and returns an error message to the user. It will be also the case when the given program will not respect the input syntax or some classical rules of programming, such as the use of undeclared variables. Note that `TOPICS` does not verify is the program is well-typed, however an user could check this using a C compiler.

malloc-expression	::=	*size-expression-point*
	|	*integer* $\underline{\texttt{*}}$ *size-expression-tab*
	|	*size-expression-tab* $\underline{\texttt{*}}$ *integer*
size-expression-point	::=	$\underline{\textbf{sizeof}}$ $\underline{(}$ $\underline{\textbf{struct}}$ *identifier* $\underline{)}$
size-expression-tab	::=	$\underline{\textbf{sizeof}}$ $\underline{(}$ *type-name* $\underline{)}$
index-expression	::=	*integer*
	|	*identifier*
lvalue-expression	::=	*identifier*
	|	*identifier* $\underline{[}$ *index-expression* $\underline{]}$
	|	*identifier* $\underline{\texttt{->}}$ *identifier*
term	::=	*lvalue-expression*
	|	*integer*
	|	$\underline{\textbf{NULL}}$
rvalue-expression	::=	*term*
	|	*term* $\underline{+}$ *term*
	|	*term* $\underline{-}$ *term*
	|	$\underline{\textbf{any}}$
boolean-expression	::=	*term* $\underline{==}$ *term*
	|	*term* $\underline{!=}$ *term*
	|	*term* $\underline{<=}$ *term*
	|	*term* $\underline{>=}$ *term*
	|	*term* $\underline{<}$ *term*
	|	*term* $\underline{>}$ *term*
	|	$\underline{\textbf{any}}$
	|	$\underline{!}$ *boolean-expression*
	|	*boolean-expression* $\underline{\&\&}$ *boolean-expression*
	|	*boolean-expression* $\underline{\|\|}$ *boolean-expression*
	|	$\underline{(}$ *boolean-expression* $\underline{)}$

Figure 5: Expressions

3.2 Syntax for the initial configurations

The syntax of the files containing the description of the initial configuration is given by the figure 6. With this language, the user can give the initial values for the pointer variables, the arrays and the integers of the programs. It can also use counters whose values is implicitly strictly positive. To describe the initial state of the memory heap, the user declare some nodes or some abstract nodes which are labeled by counters. These counters tell how many nodes are represented in the list segment characterized by the abstract node. Hence when an abstract node is labeled with a counter k, then this node corresponds to a succession of k nodes. The user can also define constraints over these counters saying that the values of two counters are equal or that the value of a counter is strictly greater than an integer. For what concerned the use of arrays, the user can tell which is the size of an array and define the values of the different elements contained in the array. In the case of array of integers, the user can also use counters if he wants to specify that the value of an elements is any strictly positive value. With the same method, the user can associate a counter to an integer variable. Using these counters, the user can verify programs with parameters.

We have seen with the previous example, how this syntax allows to describe some arithmetic properties over the length of single linked lists. We now present an other example. To describe an initial configuration in which an array `t` is of size 3 and each of its element is any integer strictly greater than its position in the array, the user can use the following description :

```
counter k0;
counter k1;
counter k2;
int tab size t=3;
value t[0]=k0;
value t[1]=k1;
value t[2]=k2;
constraint k0>0;
constraint k1>1;
constraint k2>2;
```

If the initial configuration file uses variables which are not variables of the given program, `TOPICS` will return an error message. This will also happen if the graph given to describe an initial configuration of the memory heap contains some nodes which are not reachable by pointer variables or if indexes used in an array are greater than the size of the array.

```
             configuration  ::=  { description }*
               description  ::=  counter-declaration
                              |  node-delaration
                              |  abstract-node-declaration
                              |  succ-declaration
                              |  pointer-declaration
                              |  tab-int-declaration
                              |  tab-list-declaration
                              |  tab-declaration
                              |  value-declaration
                              |  constraint-declaration
       counter-declaration  ::=  counter identifier ;
            node-delaration  ::=  node identifier ;
 abstract-node-declaration  ::=  abstract node identifier [ identifier ] ;
          succ-declaration  ::=  succ identifier = identifier ;
                              |  succ identifier = NULL ;
       pointer-declaration  ::=  pointer identifier -> identifier ;
                              |  pointer identifier -> NULL ;
       tab-int-declaration  ::=  int tab size identifier = integer ;
      tab-list-declaration  ::=  list tab size identifier = integer ;
           tab-declaration  ::=  tab identifier = identifier ;
         value-declaration  ::=  value identifier [ integer ] = identifier ;
                              |  value identifier [ integer ] = integer ;
                              |  value identifier [ integer ] = NULL ;
                              |  value identifier = identifier ;
                              |  value identifier = integer ;
    constraint-declaration  ::=  constraint identifier >= integer ;
                              |  constraint identifier = identifier ;
```

Figure 6: Syntax of TOPICSfor the initial configuration

References

[BBH$^+$06] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomáš Vojnar. Programs with lists are counter automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531. Springer, 2006.

[BFLP03] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. Fast: Fast acceleration of symbolic transition systems. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BFLP08] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 2008. To appear.

[BFLS06] Sébastien Bardin, Alain Finkel, Étienne Lozes, and Arnaud Sangnier. From pointer systems to counter systems using shape analysis. In *5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, Vienna, Austria, 2006.

[BLP06] Sébastien Bardin, Jérôme Leroux, and Gérald Point. FAST extended release. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 63–66, 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages (POPL'77)*, pages 238–252. ACM, 1977.

[Cup] CUP - LALR Parser Generator in Java. http://www2.cs.tum.edu/projects/cup/.

[GH06] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In *13th Iternational Conference Symposium Static Analysis*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[JFl] JFlex - The Fast Scanner Generator for Java. http://jflex.de/.