Model Checking Executable Specification for Reactive Components

Bruno Blašković
Faculty of Electrical Engineering and Computing
Zagreb, Croatia
Email: bruno.blaskovic@fer.hr

***Abstract*—Finding design errors in the earliest phase of software developments is still challenging area of research. This paper deals with model checking of executable specification. Executable specification is introduced as *C* program. After that, *C* program is transformed into an input model for the Spin model checker. At the end, an example for the `Zune30` bug is presented.**

***Keywords*-executable specification; reactive component; software model checking; model transformation**

I. Introduction

Telecommunication network software system can be modeled as the set of hierarchically connected communicating finite state automata (FSA). The basic unit of behavior is reactive component, modeled as FSA and referred as model $\mathcal{M}$ in this paper. FSA is implemented in C language subset. Such approach provides executable specification for component behavior analysis. In this paper, analysis is focused on component model checking. Executable specification can also serve as starting point for test cases definition, component simulator and target code skeleton generation. Component quality assurance is provided through *safety* ("Bad things will never happen") and *liveness* ("Good things will eventually happen") properties verification.

If property do not hold, component exhibits illegal behavior. Model checking approach define the model and check the properties of the model by means of assertions (invariants) and temporal logic formulas. In the case of illegal behavior, model checker provides counterexamples. Counterexample consists of a set of actions that describe paths (sequence of actions) to the errors.

FSA transitions describe dynamic behavior: C instructions are abstractions for internal actions like method calls or external actions like message sending/receiving events. There are no pointers or arrays in C code yielding straightforward translation; there is always the same FSA with different syntax representation.

First, designer defines FSA as C program. After that, C program is transformed to the form suitable for model checking. In short, FSA is designer's viewpoint about component behavior.

In order to model check or verify component behavior additional commands like *assertions* (`assert(<condition>)`) and *labels* are included into the program source. If all *assertions* are true, or if they are never false, program satisfies *"liveness"* property. The problem is where to put *assertions*: false *assertion* are hard to detect because that part of the code can be unreachable. Even the more, desired behavior can include another kind of *assertions* that can be true "many times", "infinitely often" or "only once". Such "*assertions*" are expressed with **L**inear **T**ime **L**ogic (LTL) formula. In Section VI-A, an example of linear time logic formula usage is presented.

Labels are used to check regular behavior and illegal behavior, respectively. Program is *"safe"* if "error" labels are always be unreachable, and "end" labels are eventually reachable. Program testing will not find all false *assertions*, so additional efforts are required. Model checker Spin has built–in facilities to detect assertion violations and unreached labels. In this paper we use model checker Spin to find unreached "end–state" and "non–progress" loops. Spin can also check concurrent errors from the specification. We expect separate study in order to extend specification with concurrent issue and to improve the abstraction for unbounded data values. Another set of problems are space–time limits. Space time limits are known as state explosion problem, because number of states grows exponentially. If only part of the system, consisting of several components is under consideration, state explosion problem can be under control.

This paper is organized as follows. After Introduction, in Section II related work is described. Scope and motivations are in Section III and theoretical background is presented in Section IV. After that, sketches of ψ–algorithms for model transformations are introduced in Section V. Description of a *C* source to the *Promela* code is in Section VI. The results of an experiment are given in Section VI-A, and, at the end, are conclusion and further research directions.

II. Related work

First, we introduce three approaches where specification in various formalism is translated into an model for model checkers. The origin of formal specification with Statechart is introduced by Harel in [1]. Statechart is an extension of FSA. Every statechart user defines specific properties that are hard to express in unique specification language, because statechart have no unique and clear semantic.

Object–oriented statechart semantic based on Statemate tool is formalized as labeled transition system in [2]. Graphic editing tool TCM can produce output to SMV, NuSMV and KRONOS model checkers. Bharadwaj and Heitmeyer [3] uses SCR (Software Cost Reduction) tabular notation as specification language. SCR specifications is transformed into an input model for Spin and SMV model checker. In all mentioned approaches [1] [2] [3], model definition phase prepares specification for model checkers. Our approach uses FSA encoded in C language. With such an approach, we avoid problems with semi–formal statechart semantic and the usage of specification languages outside UML set of diagrams.

Translating C programs to Promela [4] is another approach that checks C programs. A similar approach exists in [5] where Promela model is extended with direct inclusion of C code. Both approaches [4] and [5]) addresses C–code model checking. Our approach targets model checking of specification modeled as C program.

In [6] [7], `cbmc` C program model checker is described. C source is abstracted as Boolean program that is checked with satisfiability (SAT) tool. Our approach uses model extractor that is the part of `cbmc`. Extracted FSA follows BNF definition for FSA introduced later in Figure 4. We find this combination of tools useful because `cbmc` extracts models and Spin can check properties expressed in linear time temporal logic. At the end of this Section, the model checking fundamentals are introduced in [8]. Explicit state model checker Spin with industrial strength experience is described in [9]. Spin modeling language is called *Promela*.

III. Scope and motivation

It is well known fact that design errors like deadlock states, non–progress loops, illegal program termination, and message buffers overflow must be discovered as early as possible during the software life cycle. This paper is focused on executable specification analysis and the model transformation of executable specification to *Promela* model. For that purposes an illustrative example regarding `zune30 bug` has been selected from [10]. In our case, real scale example were components for e–Invoice service where an infinite loop has been discovered. `Zune30 bug` example has similar features as find in real scale examples, like unreached code or infinite loop. Similar piece of code with "small" programming mistake in telephone switch software canceled 50% of 133 million long distance calls.

The approach introduced within the paper bridges the gap between the tools capable of finding design errors and semi–formal specification. Usual approach for system specification is textual or semi–formal form, using UML or SDL+MSC diagrams. This paper starts from the C language model $\mathcal{M}$ as executable specification of state–transition system. It is designers responsibility to provide component model as much as possible close to the original. For that purpose, C language specification uses only small part of C language constructs that have direct implementation in Promela, because there is no need for pointers, complex data–type structures or arrays. After translation to Promela model, Spin [9] builds `pan` validator where checking procedures take place. Besides that, after model checker has proved desired properties, executable specification can be transformed to code skeleton (target language implementation).

Another possibility is to model specification as statechart and directly transform to the model that can understand the model checker. This approach yields several design inconsistencies:

– the semantic for a Statechart model of specification and the semantic of Promela model for the same specification is in general case different because specification can be interpreted in different ways,
– introducing executable specification as an intermediate representation (Figure 3.) provides the "simulator" for real application yielding information about overall system semantic and behavior, avoiding design inconsistencies,
– target code and model checking results are inconsistent without executable specification.

Instructions from the *C* language executable specifications are transitions that represents the real system behavior. Single transition is model or abstraction that describe method call, indivisible sequence of method calls, FSA execution or network of connected FSA executions. Although the model M, in most cases, describes single FSA, we can easily compose communicating FSA to the single higher level FSA using asynchronous product of FSA. Asynchronous product of FSA is built in feature of Spin (for details see [9] Appendix A). Each FSA is separate process in Promela model. Generic model $\mathcal{M}$ for reactive component or generic FSA or proces from Promela are syntactically different but semantically equivalent basic building block for component specification and definition. Transition $t_{\mathcal{M}}$ from Figure 3. describes the position of executable specification within the generic model, models behavior and unify transition semantic between all models. Each transition has the same form as Mealy FSA but with extended transition semantic (1).

$$\frac{input\_event}{output\_action} \quad (1)$$

Input_events are:

– guards, control–flow instructions, i.g., `if`
– message receiving events

Output_events are:

- message sending events,
- method calls,
- assignments,
- call of another FSA or FSA network.

IV. Theoretical background

Theoretical background is based on model $\mathcal{M}$ transformations [11] and consists of the following parts :

(1) "*Triptych*" environment for two–phase model transformations (Figure 2): (1) from high–level specification or requirements to executable specification and (2) from executable specification to verification (*Promela*–prml) or (2) to implementation code.
(2) model $\mathcal{M}$ for generic reactive component (Figure 3). Component is finite state automaton (FSA) with C language or Promela `proctype` construct representation,
(3) *C* program as executable specification (Section V-C). Executable specification can also be tested like any piece of C code,
(4) $\mathcal{M}_{tr}$ model transformation as framework for model checking executable specification (Figure 1),
(5) ψ algorithms for $\mathcal{M}_{tr}$ model transformations. Due to restricted instruction set in C specification, model transformations are simple `Perl` scripts.

We perform model checking for model $\mathcal{M}$ for property φ. Our approach follows usual approach [8] for model checking as described in Equation 2:

$$\mathcal{M}_{\texttt{FSA}} \models \varphi_{\texttt{LTL}} \qquad (2)$$

A model $\mathcal{M}$ is an executable specification expressed as state–transition system or more precisely as extended FSA (eFSA). Extended FSA models:

- single eFSA,
- network of communicating FSA (cFSA),
- hierarchical network of communicating eFSA (hcFSA).

There is no universal approach for model checking executable specification. That means every domain is specific regarding designers or users requirements. As a consequence, we focus our attention to reactive software components generic model $\mathcal{M}$ (Figure 3) as the basic building block for FSA, extended FSA (eFSA), communicating FSA (cFSA) and hierarchical FSA (h(cFSA)).

From initial state s_0 (Figure 3) transition t_o initiates the component. There are two possible end states, regular (`end_OK`) and illegal (`end_NOK`), respectively.

Regular behavior is abstracted within the single transition $t_{\mathcal{M}}$. As previously said $t_{\mathcal{M}}$ can abstract the behavior of cFSA or h(cFSA). Illegal behavior is executed within $t_{\neg\mathcal{M}}$ transition. In regular cases FSA returns to initial state s_0 with

$$\begin{tikzcd} \mathcal{M}^{\alpha}_{\text{spec}} \arrow[r, "\psi_1"] & \mathcal{M}^{\omega}_{\mathcal{C}} \arrow[r, "\psi_2\text{:c2cfg;cfg2prml}"] & \mathcal{M}^{\pi}_{prml} \end{tikzcd}$$

Figure 1. Model transformation sequence

t_{OK} transition while in illegal cases FSA returns to initial state with t_{NOK} transition (represented with dashed line on Figure 3), respectively. Following Equation 2. we introduce model $\mathcal{M}$ as triple in Floyd–Hoare logic and properties φ for *safety* and *liveness*:

$$\mathcal{M} \equiv \langle\{\texttt{INV}\ \mathit{pre}\}\ \mathit{code}\ \{\texttt{INV}\ \mathit{post}\}\rangle \longrightarrow \langle\varphi \equiv \Diamond\,\Box\,\texttt{np\_}\rangle \qquad (3)$$

Introduced linear time logicformula is checked with the `pan` analyzer of model checker Spin [9]. Executable specification

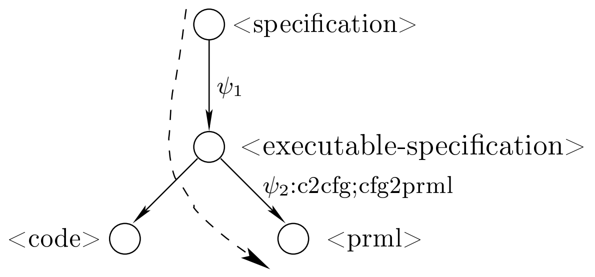

Figure 2. Triptych

is derived from top level semi–formal specification as described on Figure 2. (`<specification>` labeled circle). Top level specification (M_{spec} on Figure 1.) is transformed to executable specification with ψ_1 algorithm. In this paper

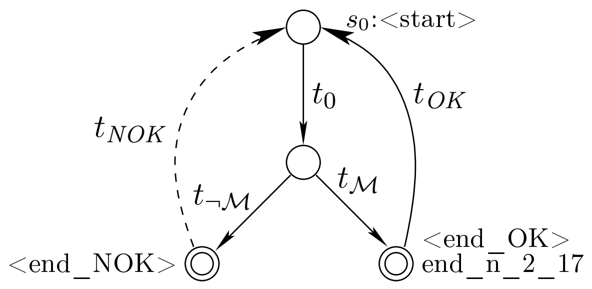

Figure 3. eFSA generic model $\mathcal{M}$ for Reactive Component

we focus our attention on translation of an executable specification to the *Promela* model. *Promela* model is the input to Spin model checker suitable for analysis of property φ from 3.

Code generation and transformation to executable specification are not the subject of this paper. The sequence of model transformation is summarized on Figure 1. In order to unify syntax representation for all internal FSA model transformation BNF representation is introduced in Figure 4.

V. ψ-algorithms

First, we introduce the definition of `ccfg`–c control flow graph. We shall refer `ccfg` simply as control flow graph `cfg`. A `cfg` is triple (S, T, L) where:

S	set of states s_i, $s_i \in S$;
T	(or $\longrightarrow$) is the set of transitions such that $T \subseteq S \times L \times S$
L	is labeling functions (assign `C` instruction to the label l_j, $l \in L$

```

<eFSA> ::= <header> <body> | <comment>
 <header> ::= [h|H]  (.)* '\n'
 <comment> ::= # (.)* '\n'
 <body> ::= <keyw> <records>
  <records> ::= <record> '\n'
   <record> ::= <fields> <separator>
    <fields> ::= '[\w-_\(\)]'+
  <separator> ::= '\s'+ | '\t'+ | ':' | ','
  <keyw> ::= INIT | STATES | LABELS |
             TRANSITIONS} | FINAL | SL | LT

```

Figure 4. BNF for $\mathcal{M}$ FSA

Control flow graph `cfg` follows previously mentioned BNF syntax for `eFSA`, `cfg` derived from *C* source is presented in lisp–like form as the set of *state–label* pairs and the set of *state–arrow–next-state* triples, respectively:

$$(s_i, l_j)$$
$$(s_i \longrightarrow s_{i+i})$$

A. ψ_2–c to cfg

This algorithm (`c2cfg`) is model extraction [7] for `C` program. We use `goto-cc` model extractor introduced in [6].

B. ψ_2–cfg to prml

Control flow graph translation to *Promela* model (`cfg2prml`) algorithm consists of the following steps:

(1) substitute arrow $\rightarrow$ with label: $(s_i \rightarrow s_{i+i}) \longrightarrow (s_i\, l_j\, s_{i+i})$
(2) abstract label l_j: $l_j \longrightarrow < l_j >$: **abstraction is already in C source.**
(3) $\forall s_i \in S$ substitute s_i with *Promela* `if` block or label abstraction
(4) "End of function" $\rightarrow$ `end_`

ψ_2–cfg to prml translation is realized as`Perl` script.

C. Example

As an example we present model $\mathcal{M}$ of `Zune30` bug. C program has been taken from [10] and translated to $\mathcal{M}_{prml}$ *Promela* model. This *C* program serves as executable specification model. Similar models $\mathcal{M}_{spec}$ of executable specification are derived from semi formal specification of distributed web applications, business processes and control software. For inputs like 366, 10593 `zune30.c` program enter endless loop. `Assertion` from line `15` with `Q1:` label is never executed in C program, yielding no *assert–violation*:

"$\mathcal{M}_C$ C source for zune30"

```
1
2 /* BUG: issue ./zune30 366, 10593 */
3 /* and have endless loop         */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <assert.h>
8
9 int zune30(int days) {
10
11   int year = 1980;
12   while (days > 365) {
13     if ((year % 4) == 0){
14       if (days > 366) {
15 Q1:        assert(1);
16         days = days - 366;
17         year = year + 1;
18       }
19 /*       else { */
20 /*       }      */
21     }
22     else {
23       days = days - 365;
24       year = year + 1;
25     }
26   }
27   printf("%d\n", year);
28   return 1;
29 }
30
```

After transformation *C* source ($\mathcal{M}_C$) is translated to *Promela* model ($\mathcal{M}_{prml}$). Analysis of *Promela* model $\mathcal{M}_{prml}$ gives the sequence of instructions that raise undesired behavior.

"$\mathcal{M}^{\pi}_{prml}$: the Promela model"

```

int UNKNOWN;
int year;
int days;
int cprntf;
int cgoto;
int creturn;
int cassertif;
int cassertFALSE;
int cassertTRUE;

active proctype acz() {

#if DAYS
   days=DAYS;
#endif

n_2_0: UNKNOWN=0; -> goto n_2_1;

n_2_1: year = 1980; ->  goto n_2_2;

n_2_2:
if
:: !(days > 365) -> goto n_2_15;   //true
::  (days > 365) -> goto n_2_3;   // false
fi;

n_2_15: cprntf=1 -> goto n_2_16;

n_2_3:
if
:: !(year % 4 == 0) -> goto n_2_12;  // true
::  (year % 4 == 0) -> goto n_2_4;   // false
fi;

n_2_16: creturn=2 -> goto end_n_2_17;
n_2_12: days = days - 365 -> goto n_2_13;

n_2_4:
if
:: !(days > 366) ->  goto n_2_11;  // true
::  (days > 366) -> goto n_2_5;    //  false
fi;

n_2_13: year = year + 1 -> goto n_2_14;
n_2_11: cgoto=3 -> goto n_2_14;

n_2_5:
if
:: cassertif=4 ->  goto n_2_8;   //true
:: cassertif=5 ->  goto n_2_6;   //false
fi;

n_2_14: cgoto=6 -> goto n_2_2;
n_2_8: cassertFALSE=7 -> goto n_2_9;
n_2_6: cassertTRUE=8 -> goto n_2_7;
n_2_9: days = days - 366 -> goto n_2_10;
n_2_7: cgoto=9 -> goto n_2_9;
n_2_10: year = year + 1 -> goto n_2_11;

end_n_2_17: skip;    // End of Function
}
```

Next section will explain transformation from *C* source to *Promela* model.

VI. Model transformation: from *C* to *Promela*

Model transformation is performed following the theoretical concepts from the Section IV and Figure 1. The first step is call to `goto-cc` that implements transformation of *C* source to control flow graph `cfg`. ($\mathcal{M}^{\omega}_{cfg} \longrightarrow \mathcal{M}^{\omega}_{cfg}$). Transformation is implemented in ψ:`c2cfg` algorithm.

Vertexes from Figure 5 are executable instructions and edges are "connections" between instructions, respectively. Nodes are assignments like `year=year+1` or `if` statements (for example: `if(days >365)`. In real situations additional assignments are method calls. We assume that methods are safe and live, thus always return desired values. That means methods have *assume-guarantee* property that is checked separately.

The result of the transformation is coded in lisp–like syntax:

– "$\mathcal{M}^{\omega}_{cfg}$ cfg for zune30 in lisp--like syntax" –

```
SL
(n_2_0  UNKNOWN)
(n_2_1  "year = 1980;")
(n_2_2  "!(days > 365)?")
(n_2_15  "PRINTF("%d\n",year)")
(n_2_3  "!(year % 4 == 0)?")
(n_2_16  "return 1;")
(n_2_12  "days = days - 365;")
(n_2_4  "!(days > 366)?")
(n_2_17  "End of Function")
(n_2_13  "year = year + 1;")
(n_2_11  Goto)
(n_2_5  "!(_Bool)1?")
(n_2_14  Goto)
(n_2_8  "Assert(FALSE)")
(n_2_6  "(void)0;")
(n_2_9  "days = days - 366;")
(n_2_7  Goto)
(n_2_10  "year = year + 1;")

LT
(n_2_0 -> n_2_1)
(n_2_1 -> n_2_2)
(n_2_2 -> n_2_15  true)
(n_2_2 -> n_2_3   false)
(n_2_15 -> n_2_16)
(n_2_3 -> n_2_12 true)
(n_2_3 -> n_2_4  false)
(n_2_16 -> n_2_17)
(n_2_12 -> n_2_13)
(n_2_4 -> n_2_11 true)
(n_2_4 -> n_2_5  false)
(n_2_13 -> n_2_14)
(n_2_11 -> n_2_14)
(n_2_5 -> n_2_8 true)
(n_2_5 -> n_2_6 false)
(n_2_14 -> n_2_2)
(n_2_8 -> n_2_9)
(n_2_6 -> n_2_7)
(n_2_9 -> n_2_10)
(n_2_7 -> n_2_9)
(n_2_10 -> n_2_11)
```

For example, vertexes `n_2_1` is assignment for *C* statement `year=1980` and transitions between vertexes are triples `(n_2_1` $\longrightarrow$ `n_2_2)`, respectively.

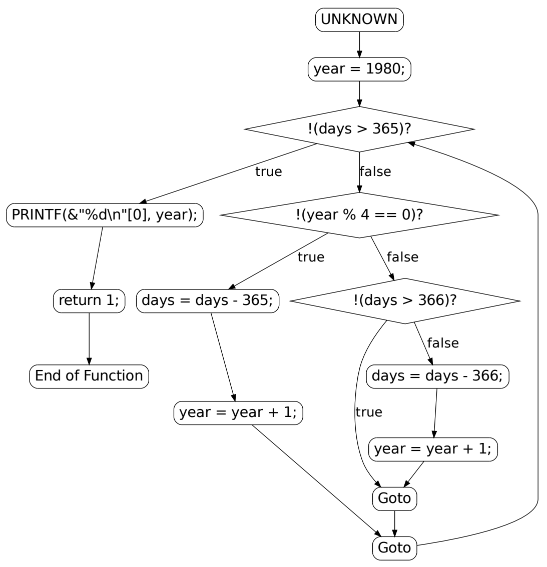

Figure 5. C Control flow graph `cfg` for zune30 example

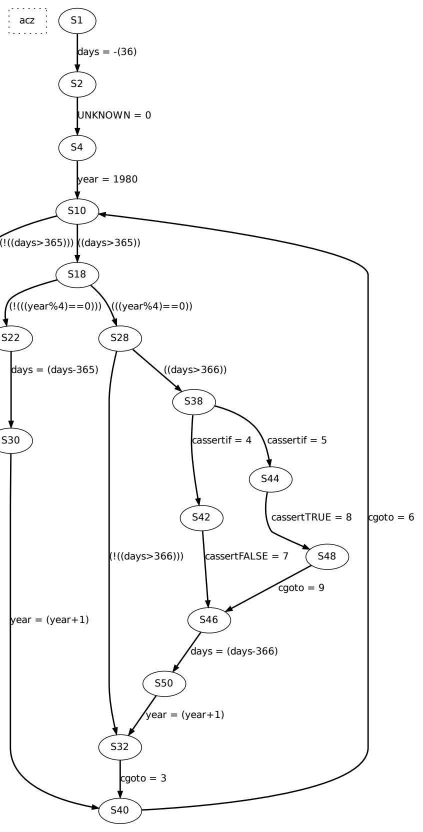

Figure 6. FSA for Promela model $\mathcal{M}^{\pi}_{prml}$

After that, algorithm ψ_2:*cfg2prml* is applied, resulting in *Promela* model $\mathcal{M}^{\pi}_{prml}$ as presented in Section. V-C. The algorithm translates control flow graph to *Promela* code $\mathcal{M}^{\pi}_{cfg} \longrightarrow \mathcal{M}^{\pi}_{prml}$. The *Promela* model is another invariant form of $\mathcal{M}^{\pi}_{C}$ model, Figure 6 visualize it as an finite state automaton. Spin‘s verifier `pan` has options that enable visualization of *Promela* models as automaton.

There are significant difference from `cfa` from figure 5, instructions l_j are placed on transition labels and many instructions have abstracted form $< l_j >$, for example, `assert` is replaced with `cassertif=4` abstracted form.

Next step is *Promela* model analysis of liveness and safety properties.

A. *Experiment result analysis*

The analysis of *Promela* model from Section V-C yields the following results:

– there are unreached portions of code
– there are endless loops

In order to achieve this results two verifier runs are required:

– Non–progress cycles (loops) are detected with linear temporal logic formula: $\Diamond\Box$`np_`, where `np_` is *Promela* built–in variable for marking the progress of global system state status. In our example, formula is *false* producing counterexample with non–progress cycle.
– unreached instruction from *Promela* model (“dead–code”) is standard built–in function into the *pan* verifier.

The output from the *pan* verifier is counterexample with the path to the error. Each row presents the line number of instruction from the *Promela* model presented in Section V-C. Non–progress loop is sequence of instructions with line numbers `24 32 40 45 53 24 32 40 ...`

```
"non-progres loops"

  z30.ltg.prml:14          [days = 366]
               days = 366
  z30.ltg.prml:17          [UNKNOWN = 0]
  z30.ltg.prml:19         [year = 1980]
        year = 1980
  <<<<<START OF CYCLE>>>>>
  z30.ltg.prml:24          [((days>365))]
  z30.ltg.prml:32  [(((year%4)==0))]
  z30.ltg.prml:40          [(!((days>366)))]
  z30.ltg.prml:45          [cgoto = 3]
        cgoto = 3
  z30.ltg.prml:53         [cgoto = 6]
        cgoto = 6
spin: trail ends after 16 steps
               year = 1980
               days = 366
```

Counterexample pointing unreached code use `pan` verifier built in options for unreached code detection. Each row presents the line number of unreached instruction from *Promela* model presented in Section V-C (27, 35, 36, 44, . . .).

```
"unreached end--state"

unreached in proctype z30

  z30.ltg.prml:27,  "cprntf = 1"
  z30.ltg.prml:35,  "creturn = 2"
  z30.ltg.prml:36,  "days = (days-365)"
  z30.ltg.prml:44,  "year = (year+1)"
  z30.ltg.prml:44,  "year = (year+1)"
  z30.ltg.prml:49,  "cassertif = 4"
  z30.ltg.prml:49,  "cassertif = 5"
  z30.ltg.prml:54,  "cassertFALSE = 7"
  z30.ltg.prml:55,  "cassertTRUE = 8"
  z30.ltg.prml:56,  "days = (days-366)"
  z30.ltg.prml:57,  "cgoto = 9"
  z30.ltg.prml:58,  "year = (year+1)"
  z30.ltg.prml:61,  "-end-"
        (11 of 53 states)
```

VII. Conclusion and further work

We have presented model checking of specification as software model checking for C language.

We find that Spin model checker is feasible solution because Spin finds deadlocks, unreached code, assertion violations, invalid end states, and analyze linear time logicformula. In the same time, executable specification can be analyzed, tested as every C program. Our approach avoids complex and long term development of model extractor with tools like CIL. Another benefit is the application of linear time logicformula on C specification. Usual approach puts assertions in the code in the place according to the designer's discretion. Sometimes it is necessary that assertion is true to "some point in the future infinitely often" which can be expressed as temporal logic formula. With our approach linear time logic formula is the part of Promela model and consequently also the part of C specification. State explosion and designer mistakes during specification definition are still problems that needs improvements. "Designers will never use it" syndrome is always the problem when introducing development paradigms.

Further work will focus on more rigid data–types consistency check. That requires formal development of abstract data structures. In most cases, such data structures are defined over infinite domains so further refinements should avoid infinite data domains, or introduce data abstractions.

Besides Spin, the comparison with other model checkers, like Petri net tools, could improve verification. Model checkers search for solutions within finite space, the improvement of model checking with unbounded parameters (`days` in our example) yields: $\mathcal{M}(days) \models \varphi$.

Bounded model checking [12] and the usage satisfiability modulo theory (SAT [13] and SMT [14]) solvers are the promising research direction.

Automated code generation from executable specification is another possible direction for research. The most promising is TDD "Test Driven Development" because code skeleton is populated with test case commands.

References

[1] D. Harel, "Statecharts in the making: a personal account," in *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*. San Diego, California: ACM, 9-10 June 2007, pp. 1–43.

[2] R. Eshuis, D. N. Jansen, and R. Wieringa, "Requirements-level semantics and model checking of object-oriented statecharts." *Requirements Engineering*, vol. 7, no. 4, pp. 243–263, 2002.

[3] R. Bharadwaj and C. L. Heitmeyer, "Model Checking Complete Requirements Specifications Using Abstraction," *Automated Softwware Engineering*, vol. 6, no. 1, pp. 37–68, 1999.

[4] K. Jiang, "Model Checking C Programs by Translating C to Promela," Master's thesis, Uppsala Universitet, Department of Information Technology, 2009.

[5] G. J. Holzmann, "Logic Verification of ANSI-C Code with SPIN," in *SPIN Model Checking and Software Verification*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885, 7^{th} International SPIN Workshop. Stanford CA USA: Springer, August 2000, pp. 131–147.

[6] "CBMC is a Bounded Model Checker for ANSI-C," (last time visited July, 5^{th}2012). [Online]. Available: http://www.cprover.org/cbmc

[7] E. Clarke, D. Kroening, and F. Lerda, " A Tool for Checking ANSI-C Programs ," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* , ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, January 1999.

[9] G. Holzmann, *Spin model checker, the: primer and reference manual*, 1st ed. Addison-Wesley Professional, 2004.

[10] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[11] A. Metzger, "A systematic look at model transformations," in *Model-Driven Software Development*, S. Beydeda, M. Book, and V. Gruhn, Eds. Springer Berlin Heidelberg, 2005, pp. 19–33.

[12] Armin Biere and Alessandro Cimatti and Edmund M. Clarke and Ofer Strichman and Yunshan Zhu, "Bounded Model Checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[13] Biere, Armin and Heule, Marijn J. H. and van Maaren, Hans and Walsh, Toby, Ed., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, February 2009, vol. 185.

[14] Armando, Alessandro and Mantovani, Jacopo and Platania, Lorenzo, "Bounded model checking of software using SMT solvers instead of SAT solvers," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 69–83, Jan. 2009.