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 Linear Time Logic (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 $\psi$–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 Promela model checker 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 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 unreach code or infinite loop. Similarly 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 $M$ 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 $M$ for reactive component or generic FSA or process from Promela are syntactically different but semantically equivalent basic building block for component specification and definition. Transition $t_M$ 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).

$$\begin{align*}
\text{input_event} & \\
\text{output_action} & \\
\end{align*}$$

**Input_events** are:
- guards, control-flow instructions, i.e., 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. $M_t$, model transformation as framework for model checking executable specification (Figure 1).

5. $\psi$ algorithms for $M_t$ 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 $\varphi$. Our approach follows usual approach [8] for model checking as described in Equation 2:

$$\mathcal{M}_{\text{FSA}} \models \varphi_{\text{LTL}}$$  \hspace{1cm} (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 (eFSA),
- hierarchical network of communicating FSA (hFSA).

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 (hcFSA).

From initial state $s_0$ (Figure 3) transition $t_0$ 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 $t_M$. As previously said $t_M$ can abstract the behavior of cFSA or hcFSA). Illegal behavior is executed within $t_{\sim M}$ transition. In regular cases FSA returns to initial state $s_0$ with $t_OK$ 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 $\varphi$ for safety and liveness:

$$\mathcal{M} \equiv \langle \{\text{INV pre} \} \text{ code } \{\text{INV post} \} \rangle \mapsto \langle \varphi \equiv \Diamond \square \text{np} \rangle$$  \hspace{1cm} (3)

Introduce linear time logification is checked with the pan analyzer of model checker Spin [9]. Executable specification is derived from top level semi–formal specification as described on Figure 2. ( specification labeled circle). Top level specification ($M_{\text{spec}}$ on Figure 1.) is transformed to executable specification with $\psi_1$ algorithm. In this paper 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 $\varphi$ 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. $\psi$-ALGORITHMS

First, we introduce the definition of cfg–control flow graph. We shall refer cfg simply as control flow graph cfg. A cfg is triple $\langle S, T, L \rangle$ where:

- $S$ set of states $s_i, s_i \in S$;
- $T$ (or $\rightarrow$) is the set of transitions such that $T \subseteq S \times L \times S$;
- $L$ is labeling functions (assign C instruction to the label $l_j, l_j \in L$)
Control flow graph $\text{cfg}$ follows the previously mentioned BNF syntax for eFSA, $\text{cfg}$ 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:

$$(s_i, l_j)$$

$$(s_i \rightarrow s_{i+1})$$

A. $\psi_2$–c to $\text{cfg}$

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

B. $\psi_2$–$\text{cfg}$ to prml

Control flow graph translation to Promela model ($\text{cfg2prml}$) algorithm consists of the following steps:

1. substitute arrow $\rightarrow$ with label:

   $$(s_i \rightarrow s_{i+1})$$

2. abstract label $l_j$: $l_j \rightarrow <l_j>$: 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

$\psi_2$–$\text{cfg}$ to prml translation is realized as Perl script.

C. Example

As an example we present model $M$ of Zune30 bug. C program has been taken from [10] and translated to $M_{\text{prml}}$. Analysis of Promela model $M_{\text{prml}}$ gives the sequence of instructions that raise undesired behavior.

```plaintext
/* BUG: issue ./zune30 366, 10593 */
/* and have endless loop */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int zune30(int days) {
    int year = 1980;
    while (days > 365) {
        if ((year % 4) == 0) {
            if (days > 366) {
                assert(1);
                days = days - 366;
                year = year + 1;
            } else {
                days = days - 365;
                year = year + 1;
            }
        } else {
            days = days - 365;
            year = year + 1;
        }
    }
    printf("%d\n", year);
    return 1;
}
```

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-233-2
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. ($\mathcal{M}_c^\omega \rightarrow \mathcal{M}_c^{cfg}$). Transformation is implemented in $\psi$ 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}_c^{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",year")")
(n_2_3 "!(year % 4 == 0)?")
(n_2_16 "return 1;"
(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;")
(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;")
(n_2_17 "End of Function")
```

For example, vertexes n_2_1 is assignment for C statement year=1980 and transitions between vertexes are triples (n_2_1 -> n_2_2), respectively.
After that, algorithm \( \psi_2:cfg2prml \) is applied, resulting in Promela model \( M_{prml}^{\pi} \) as presented in Section V-C. The algorithm translates control flow graph to Promela code \( M_{cfg}^{\pi} \rightarrow M_{prml}^{\pi} \). The Promela model is another invariant form of \( M_{\pi} \), model, Figure 6 visualize it as an finite state automaton. Spin’s verifier \( \text{pan} \) has options that enable visualization of Promela models as automaton.

There are significant difference from \( \text{cfa} \) from figure 5, instructions \( l_j \) are placed on transition labels and many instructions have abstracted form \( < l_j > \), for example, \( \text{assert} \) is replaced with \( \text{assertif}=4 \) 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 unreachend 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: \( \Diamond \Box \text{np}_\_ \) where \( \text{np}_\_ \) is Promela built–in variable for marking the progress of global system state status. In our example, formula is \( \text{false} \) producing counterexample with non–progress cycle.

- unreachened instruction from Promela model (“dead–code”) is standard built–in function into the \( \text{pan} \) verifier.

The output from the \( \text{pan} \) 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 ---

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

--- unreachend end--state ---

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, unreachend code, assertion violations, invalid end states, and analyze linear time log-icformula. 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 logic formula 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: $M(\text{days}) \models \varphi$. 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


