LLVM-based C to DEPM transformation tool: New functionality and performance improvements

Viacheslav Vidineev
Faculty of Computer Science and Robotics
Ufa State Aviation Technical University
Ufa, Russia
vidineev.vyachesla01@ugatu.ac.ru

Tagir Fabarisov
Institute of Automation, Faculty of Electrical and Computer Engineering
Technische Universität Dresden
Dresden, Germany
tagir.fabarisov@tu-dresden.de

Nafisa Yusupova
Faculty of Computer Science and Robotics
Ufa State Aviation Technical University
Ufa, Russia
yussupova@ugatu.ac.ru

Klaus Janschek
Institute of Automation, Faculty of Electrical and Computer Engineering
Technische Universität Dresden
Dresden, Germany
klaus.janschek@tu-dresden.de

Andrey Morozov
Institute of Automation, Faculty of Electrical and Computer Engineering
Technische Universität Dresden
Dresden, Germany
andrey.morozov@tu-dresden.de

***Abstract*— Recently we have presented a tool for the Error Propagation Analysis (EPA) of the safety-critical software using the developed method for the transformation of the source code to the Dual-graph Error Propagation Model (DEPM) based on the Low-Level Virtual Machine (LLVM) compiler framework. This tool enables the automatic analysis of the LLVM supported front-ends such as C-code. In order to analyze functions, basic blocks, control and data flow structures, the source code is being transformed into LLVM Intermediate Representation (IR) which contains information required for the generation of a corresponding DEPM for further analysis.**

The DEPM is a stochastic framework developed by our research team. The DEPM captures system properties relevant to the error propagation analysis such as control and data flow structures, transition probabilities and reliability characteristics of single components, in this case, LLVM instructions. The DEPM helps to estimate the impact of a fault in a particular instruction on the overall system reliability, e.g. to compute the mean number of erroneous values in a critical system output during given operation time.

This paper is devoted to the improvements of the transformation tool that have been successfully implemented and tested. The three key extensions of the tool are (i) the support of the new version of DEPM, (ii) the generation of the control flow using the LLVM IR labels instead of the elements execution sequence, and (iii) the generation of the error propagation commands for instruction elements using probabilistic parametric methods. The paper describes all the steps of development of the improvements from design to implementation. In addition, the results of the performance evaluation are presented.

Keywords—LLVM, transformation methods, software reliability, error propagation analysis.

Proceedings of the 7th All-Russian Scientific Conference "Information Technologies for Intelligent Decision Making Support", May 28-30, Ufa – Stavropol- Khanty-Mansiysk, Russia, 2019

I. INTRODUCTION

Software reliability analysis is an important part of the system-level dependability evaluation for any safety critical industrial domain. Nowadays, one of the main critical part of any industrial system is a software. Due to the high complexity of the software structures, any data error, e.g. caused by a random bit flit in CPU or RAM, could result in a data error and propagate through the entire system and eventually lead to system failures. Therefore, our main goal is to evaluate whether a data error will reach a critical system output with certain probability during the system operation. For that Error Propagation Analysis (EPA) we have used a mathematical abstraction Dual-graph Error Propagation Model (DEPM). This paper presents a new version of the tools for the automatic generation of the DEPM models from the source code.

The reminder of this paper is structured as follows. Section 2 provides the overview of the background DEPM and LLVM technologies. Section 3 presents an overview and technical details of the proposed transformation method. Section 4 describes implemented improvements of the tool. One of the most necessary improvement is the generation of DEPM models in the new format in order to work with newest versions of OpenErrorPro. This improvement is described in Section 4-A. A new method for building the control flow between basic blocks is presented in the Section 4-B. The evaluation of the fault probabilities was left out of the scope of the research focus in the previous work. In this work we propose a new parametric method for the fault probability evaluation. The description of the method is provided in the Section 4-C. Finally, Section 5 provides the results of the functional and performance evaluation of a new version of the tool and the conclusion

II. STATE OF THE ART

A. DEPM

Fault activation and the error propagation are specified using probabilistic conditions of the elements, see the conditions of A, B, and C in the right part of Fig. 1. During the execution of an element, faults can be activated and occurred errors propagate to its output data. For instance, in the element A, faults can be activated with probability 0.1, defined in the conditions of A (see Fig. 1), and occurred

errors propagate to its output data d1 and d3. The error propagation probabilities for each element are defined also using probabilistic conditions. The errors can propagate from the inputs to the outputs. For instance, the conditions of the element B specify that the element B does not activate faults, but the errors can propagate from d1 to d2 with the probability 0.9.

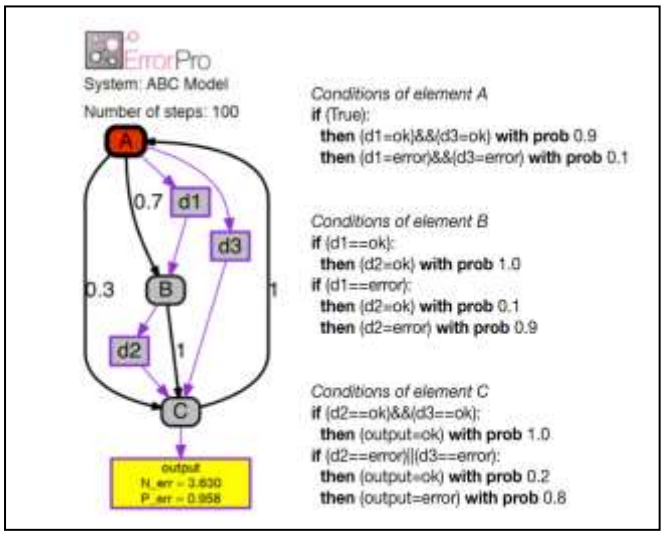

Fig. 1. A simple DEPM example and the conditions of the elements [4].

The DEPM allows the computation of several reliability metrics, such as the mean number of errors (Nerr) and probability of errors (Perr) in selected data storages. Nerr stands for the average number of erroneous values in a data storage, and Perr is the probability of an error in a data storage during the system execution. For instance, the evaluated Nerr in the data storage output during 100 steps (execution of one element is one step) is equal to 3.630, and the Perr is 0.958, as shown in Fig. 1. The computed reliability metrics are important measures for the system analysis, particularly for the reliability assessment and should comply with system requirements.

The OpenErrorPro [3, 4] is an analytical software developed in our lab that supports the system analysis with the DEPM. On the input there are baseline models that describe the target system. Several parsers transform the baseline models into the DEPM models [5].

B. LLVM

LLVM (Low Level Virtual Machine) is a collection of modular and reusable compiler and toolchain technologies. It provides a source and target-independent optimizer, as well as a code generation support for a number of CPUs [6].

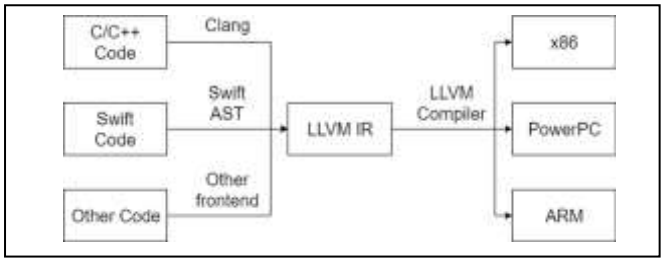

Fig. 2. LLVM use case.

These libraries are built around an assembly-like low-level code representation known as the LLVM Intermediate Representation (LLVM IR). The LLVM IR is a representation in-between a high-level language and a low-level machine code. The LLVM Pass Framework is an important part of the LLVM system. It performs the transformations and optimizations which compose the compiler, as well as building of the analysis results that are used for the transformations, moreover, passes are structuring technique for compiler code. According to the task we have developed a pass to extract required data for a DEPM.

III. TRANSFORMATION

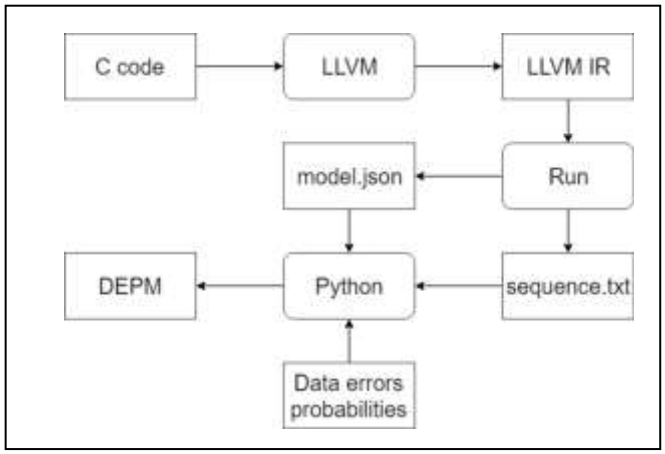

Fig. 3. An overview of the transformation process.

Figure 3 shows an overview of the transformation process. Rectangles represent data and rounded rectangles represent activities. The process is automated and performs by a single script that calls LLVM tools as well as the DEPM pass and the python script to generate the DEPM. The process consists the following three steps: (1) Compilation of the C code into the LLVM IR (see LLVM in Fig 3), (2) Execution of the generated LLVM IR code with a developed DEPM pass (see Run in Fig 3), and (3) generation of DEPM xml file using a developed python script (see Python in Fig 3).

A. Compilation of the given C code into the LLVM IR using Clang

Clang is a language front-end and the LLVM compiler infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) [6]. In this transformation tool Clang is used for the LLVM IR code generation for the DEPM pass. Listings 1 and 2 show an example of a simple function that compares two numbers and returns the biggest one in C code and LLVM IR code respectively.

```
int max(int a, int b) {
   if (a > b) {
      return a;
   } else {
      return b;
   }
}
```

Listing 1. C code.

The generated LLVM IR code will be used for the parsing of data and generation of the basic blocks execution sequence of.

```
define dso_local i32 @max(i32 %arg, i32 %arg1) {
bb:
  %tmp = alloca i32, align 4
  %tmp2 = alloca i32, align 4
  %tmp3 = alloca i32, align 4
  store i32 %arg, i32* %tmp2, align 4
  store i32 %arg1, i32* %tmp3, align 4
  %tmp4 = load i32, i32* %tmp2, align 4
  %tmp5 = load i32, i32* %tmp3, align 4
  %tmp6 = icmp sgt i32 %tmp4, %tmp5
  br i1 %tmp6, label %bb7, label %bb9

bb7:                                              ; preds = %bb
  %tmp8 = load i32, i32* %tmp2, align 4
  store i32 %tmp8, i32* %tmp, align 4
  br label %bb11

bb9:                                              ; preds = %bb
  %tmp10 = load i32, i32* %tmp3, align 4
  store i32 %tmp10, i32* %tmp, align 4
  br label %bb11

bb11:                                             ; preds = %bb9, %bb7
  %tmp12 = load i32, i32* %tmp, align 4
  ret i32 %tmp12
}
```

Listing 2. LLVM IR code.

B. Running the generated LLVM IR code with the DEPM pass.

model.json

```
{
   "max": {
      "F0_max_BB0": {
         "F0_max_BB0_INS0_alloca": {
            "call": "",
            "cf_outputs": [
               "F0_max_BB0_INS1_alloca"
            ],
            "df_inputs": [
               "max_Constant0"
            ],
            "df_outputs": [
               "max_%tmp"
            ]
         },
         …
      },
      …
   }
}
```

Listing 3. An example of the «model.json» file.

The pass iterates through the generated LLVM IR code and parses the required information for the DEPM. It generates «model.json» file which contains data in JSON format in order to keep the structure of the DEPM, e.g. function contains basic blocks, basic block contains instructions, etc. Listing 3 provides an example of the file generated from the LLVM IR code presented at the Listing 2.

Due to the restriction of the OpenErrorPro toolset, the pass generates a unique name for all the elements, e.g. “bb” becomes “F0_max_BB0”, “alloca” – “F0_max_BB0_INS0_alloca”, etc. For this instruction, the pass takes “%tmp” as an output data, “Constant0” as an input data (align 4 gives this allocation 4-byte alignment, i.e. the stack pointer will be on a 4 byte aligned address), and control flow transition to the next instruction. Control flow transitions between basic blocks are located in “br” instruction in the end of the basic block.

Additionally, in case when it is necessary to calculate a transition probability for some elements, the pass generates file “sequence.txt”, which contains basic block’s identifiers in their execution order.

C. Creation of the DEPM

At the final step of the transformation files “model.json” and “sequence.txt” are being used as inputs in order to create the DEPM using OpenErrorPro’s API. This process consists of the following steps:

1) Create a model for each function.

2) Add basic blocks to corresponding models and create a sub model for every basic block;

3) Add instructions and their control flow to corresponding basic blocks.

4) Add instruction’s and basic block’s data and data flow.

5) Add basic block’s control flow.

6) Set the control flow and commands for elements that have two or more control flow outputs using the method introduced in [8].

7) Set the error propagation commands for instructions.

8) Place all the models to their call instruction’s sub model.

The Python script parses the input, performs actions 1-8, and saves the result in XML format. The output file with DEPM model is ready for further analysis with OpenErrorPro.

IV. Improvements

Since the introduction of the method in [1], there were significance changes in the DEPM storage XML format [2]. Also, the new version of LLVM has been released. Therefore, the following improvements of the C to DEPM transformation tool have been implemented.

A. Generation of the DEPM in the new format

In order to analyze a source code with the latest version of OpenErrorPro, a DEPM xml file must be generated in the new format. Previous format (Listing 4) stored all elements, data, control flow and data flow arcs in the same main model regardless of whether they are members of a sub-model or not. For that reason, sub-level elements contained an attribute «host» with the value being a name of the host model. Elements of the DEPM saved in the new format (Listing 5) could contain an attribute «compound» designating that an element contains a sub-model.

Respectively, sub-models are represented as independent XML nodes.

```
<?xml version="1.0" encoding="utf-8"?>
<model>
   <element initial="true" name="A"/>
   <element name="B"/>
   <element name="C"/>
   <control_flow from="A" prob="1.0" to="B"/>
   <control_flow from="B" prob="1.0" to="C"/>
   <data name="d0"/>
   <data_flow from="d0" to="A"/>
   …
   <element host="A" initial="true" name="A0"/>
   <element host="A" name="A1"/>
   …
</model>
```

Listing 4. An example of DEPM model saved in the old format

```
<?xml version="1.0" encoding="utf-8"?>
<epl>
   <model initial_element="A">
      <element compound="True" name="A">
         <cfc>(cf=A) -> 1:(cf'=B);</cfc>
      </element>
      <element name="B">
         <cfc>(cf=B) -> 1:(cf'=C);</cfc>
      </element>
      <element name="C"/>
      <control_flow from="A" to="B"/>
      <control_flow from="B" to="C"/>
      <data name="d0"/>
      <data_flow from="d0" to="A"/>
      …
   </model>
   …
   <model host="A" initial_element="A0">
      <element name="A0"/>
      <element name="A1"/>
      …
   </model>
```

Listing 5. An example of DEPM model saved in the new format.

For this reason, we have modified the LLVM pass that generates DEPM data containers in accordance with the new DEPM format. Previously, C to DEPM transformation tool ran two LLVM passes, first being the element generator (Listing 6) and the second being the data list generator (Listing 7).

```
elements.txt
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
…
```

Listing 6. An example of generated «elements.txt» file.

```
elements.txt
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
Seq F_ID BB_ID INS_ID Opcode_ID BB_name INS_name
…
```

Listing 7. An example of generated «datas.txt» file.

The new version of the C to DEPM transformation tool runs only one LLVM pass that generate one file which contains information about both elements and data in JSON format (Listing 3). The structure of this file is follows the new version of DEPM and represented in more convenient for further processing way, which affects the performance significantly.

Another noteworthy improvement is a support of the new OpenErrorPro API for generation of the output XML file containing DEPM model. It uses Python's standard library to work with an XML, which makes the tool more independent of the possible format changes.

B. Generation of the control flow using the LLVM labels

The previous version of the tool was building the control flow based on the element execution sequence, and didn't consider a situation when some of the top-level elements could not be executed. In which case their control flow would not appear in the DEPM (Fig. 4). This problem has been solved in the new version of the tool, by building the control flow based on the information obtained by the DEPM pass.

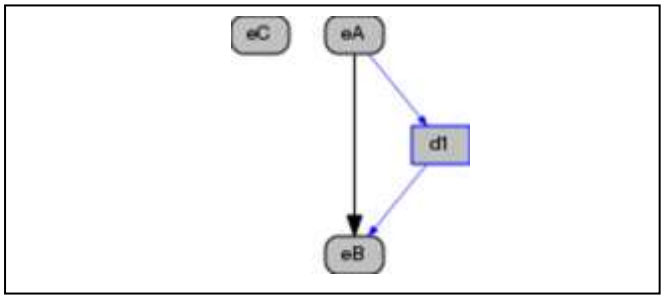

Fig. 4. An example of the DEPM with an element that has been not executed.

Another issue that has been solved is associated with the length of the execution sequence. In case when the length of the sequence is exceeding the number of the elements, tool was intending to iterate through the entire sequence for the control flow building, which significantly influenced the performance.

C. Generation of the error propagation commands for the instruction elements

Another implemented improvement is the error propagation commands generation for the instruction elements. This task was proposed for further development of the tool in [1]. Error propagation commands contain a probability of the likelihood of the data error occurrence during the execution of an instruction. It allows the computation of the reliability properties for the given software. As the result, we have developed a probabilistic

parametric method for the error propagation commands generation, that would contain failures probabilities for the instruction elements, with the probabilities being an input data. Using this method, an estimation can be made that the data errors could occur with the given probabilities.

TABLE I. An example of the method's input.

Instruction	Probability
Alloca	0.00008
Store	0.00010
Load	0.00012
Icmp	0.00013
Add	0.00015
Fmul	0.00017
Fsub	0.00017

Error propagation commands allow the computation of the reliability metrics. They are important measures that being used for the reliability assessment and should comply with system requirements [7].

V. TESTING AND RESULTS

In order to evaluate the developed functionality and improvements, a benchmarking has been performed. The tool was tested using over 20 C programs including the case study from the previous work [1]. The focus of assessment was to test the software on specific and extreme cases, such as: several nested functions, several returns, function arguments, several calls of one function and recursion. The tool has demonstrated correct and reasonable behavior for all cases. Several limitations of the tool are still in place and shall be addressed in the next versions, though:

- A function can be called only in one place.
- A function cannot call itself, i.e. no recursion.
- Pointers and arrays are not supported.
- All functions have to be defined in a single module.

In addition to the functional testing, there was also carried out the performance evaluation. Every generated model has been transformed into the DEPM model using the new version of the tool. Mean time of transformation per every model has been calculated. The snippet of results and a total result of the comparison is presented in the Table 2.

TABLE I. An example of the method's input.

Model name	Time to transform into DEPM, sec.
Access-array-pointer	0.476
Array-largerst-element	0.737
Average-arrays	0.543
Check-armstrong-number	0.342
Digits-count	0.354
Even-odd	0.401
Factorial	0.542
Fibonacci	0.631
Flugstuerung	6.517
Frequency-character	0.417
GCD	0.195
LCM	0.319
Leap-year	0.614
Natural-number-sum	0.590
Palindrome-number	0.456
Prime-number	0.675
Product-number	0.145
Standard-deviation	0.534
String-lingth	0.565
Swapping	0.399
V-for-p	0.789
Min	0.145
Max	6.517
Mean	0.773

The comparison has shown that new version takes 0,773 seconds to transform source code into the DEPM model.

CONCLUSION

The data error propagation analysis is an important part of the reliability evaluation of safety critical software. The transformation tool based on the LLVM technology has been proposed in [1]. In this paper we introduced a new version of the tool for the automatic generation of the DEPM. The key improvements of the tool are usage the new format of DEPM, the generation of the control flow using the LLVM IR labels instead of the elements execution sequence, and the generation of the error propagation commands for instruction elements using probabilistic parametric methods. The conducted functional and performance evaluation has shown that the tool's performance has been significantly increased as well as functionality has been extended.

REFERENCES

[1] A. Morozov, K. Janschek, and Y. Zhou, "Llvm-based stochastic error propagation analysis of manually developed software components," in ESREL 2018, 2018.

[2] T. Fabarisov, N. Yusupova, K. Ding, A. Morozov, K. Janschek, "Analytical Toolset for Model-based Stochastic Error Propagation Analysis: Extension and Optimization Towards Industrial Requirements", CSIT 2017, 2017.

[3] A. Morozov, Dual-graph Model for Error Propagation Analysis of Mechatronic Systems. Dresden: Jörg Vogt Verlag, 2012.

[4] A. Morozov and K. Janschek, "Probabilistic error propagation model for mechatronic systems," Mechatronics, vol. 24, no. 8, pp. 1189 – 1202, 2014.

[5] K. Ding, T. Mutzke, A. Morozov, and K. Janschek, “Automatic transformation of uml system models for model-based error propagation analysis of mechatronic systems,” IFAC-PapersOnLine, vol. 49, no. 21, pp. 439–446, 2016.

[6] The LLVM Compiler Infrastructure Project. https://llvm.org/

[7] K. Ding, A. Morozov, K. Janschek, “Reliability Evaluation of Functionally Equivalent Simulink Implementations of a PID Controller under Silent Data Corruption” in ISSRE 2018, 2018.

[8] V. Vidineev, K. Ding, A. Morozov, K. Janschek, N. Yusupova “Improved Stochastic Control Flow Model for LLVM-based Software Reliability Analysis”, CSIT 2018, 2018.

[9] OpenErrorPro on the github. https://mbsatud.github.io/OpenErrorPro/, 2019