CHALMERS

Debugging the Networked Car

Master of Science Thesis in Computer Systems and Networks

TAO LI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2013

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Debugging networked car

Tao Li

© TAO LI, July 2013.

Examiner: Olaf Landsiedel

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 17 06

[Cover: a networked car.
Source: http://www.inf.mit.bme.hu/en/research/projects/dependable-embedded-components-and-systems-decos]

Department of Computer Science and Engineering
Göteborg, Sweden August 2013

Abstract

In the automotive industry, modern cars are equipped with numerous sensors and actuators to control safety critical aspects. These sensors collect large amounts of information, which are processed and used by vehicles to control brakes, steering, engine, etc. The AUTOSAR standard specifies the implementation details of these software units. This approach provides more flexibility for the management of vehicular Electrical/Electronic(E/E) systems related to complex driver assistant functions. However, due to the distributed features of such systems, software programs built on the AUTOSAR platform are difficult to achieve high quality and correctness. AUTOSAR programs are running on different machines and exchange messages via the underlying in-vehicle network, which makes the system non-deterministic and non-predictable. In this thesis work, these problems are addressed and a new debugging tool based on a symbolic execution engine, named KLEE, is developed. This new tool enables effective testing of AUTOSAR programs before deployment on Electronic control units(ECUs). It has the capability to detect pointer-related problems and inject non-deterministic failures to programs while symbolically executing them. We explore the design and the implementation of this tool and also come up with several possible scenarios as use cases of this debugging tool.

Keywords: AUTOSAR, Networked Car, Symbolic Execution, KLEE, Debugging

Acknowledgments

I would like to thank my supervisor and examiner Dr. Olaf Landsiedel. It is his patience, encouragement, vision, and immense knowledge that helped me through the difficulties encountered during my thesis work.

I also thank to my parents for their continuous support, in both financial and spiritual way, during the past years. Without them, it is impossible for me to pursue my dreams.

Last but not least, I want to thank my friends, XingGe Xu, Ivan Walulya, and Salvo Tomaselli, for sharing their ideas, endless passion, interesting experiences and knowledge with me.

Gothenburg, July 2013

Tao Li

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Aim . . . 10
1.3 Approach . . . 11
1.4 Scientific contribution . . . 11
1.5 Outline . . . 12

2 Background **13**
2.1 Symbolic Execution . . . 13
2.1.1 Test methodology overview . . . 13
2.1.2 Concept . . . 14
2.1.3 Example . . . 14
2.2 KLEE . . . 16
2.3 AUTOSAR concept . . . 18
2.3.1 AUTOSAR background . . . 18
2.3.2 AUTOSAR architecture . . . 20

3 Related work **25**
3.1 Code execution tool . . . 25
3.2 Model based tool . . . 25
3.3 KleeNet . . . 26

4 Design **27**
4.1 System overview . . . 27
4.2 OS model . . . 28
4.2.1 AUTOSAR requirement . . . 28
4.2.2 Existing model and modification . . . 29
4.3 Network model . . . 31
4.3.1 Network topology . . . 31
4.3.2 Simulation machine . . . 32
4.3.3 Logic node . . . 33
4.3.4 Dedicated branching . . . 34
4.3.5 AUTOSAR communication . . . 34
4.3.6 Synchronization . . . 35
4.4 Failure model . . . 36
4.4.1 Packet loss . . . 37
4.4.2 Remote invalid path invocation . . . 37
4.4.3 Assertion . . . 37

5 Improved Design **38**
5.1 Overview . . . 38
5.2 OS model improvement . . . 39
5.3 Network model improvement . . . 40
5.3.1 Multiple nodes hosting . . . 40
5.3.2 Communication channel . . . 40
5.3.3 Distributed view . . . 41
5.3.4 On-demand branching . . . 42

6 Implementation **43**
6.1 Implementing simple design . . . 43
6.1.1 Existing model . . . 43
6.1.2 System . . . 43
6.1.3 Data transmission . . . 44
6.1.4 Special functions . . . 45
6.1.5 Runtime . . . 46
6.2 Implementing improved design . . . 47
6.2.1 Existing model . . . 47
6.2.2 System . . . 47
6.2.3 Dynamic switching . . . 48
6.2.4 Special function . . . 49
6.2.5 Communication and On-demand branching . . . 49
6.2.6 Scheduling and synchronization . . . 50

7 Evaluation **52**
7.1 Packet loss . . . 52
7.1.1 Experiment assumption . . . 53
7.1.2 Experiment setting . . . 54
7.1.3 Experiment result . . . 54
7.2 Invalid sensor data . . . 56
7.2.1 Experiment setting . . . 56
7.2.2 Experiment result . . . 57
7.2.3 Further test and result . . . 58

8 Conclusion **61**
8.1 Discussion and future work . . . 61
8.2 Conclusion . . . 61

List of Figures

1 Symbolic execution tree illustrating program example 16
2 Layers of AUTOSAR architecture 21
3 Layers of AUTOSAR basic software component 22
4 AUTOSAR CAN communication stack 24
5 Components of a simple debugging tool design 28
6 Architecture of the OS model 30
7 The discrete event queue containing the periodic event and the one-shot event 31
8 Multiple nodes reside in one KLEE instance on one machine . 32
9 Multiple KLEEs locate on independent machines 33
10 A symbolic execution tree showing the dedicated branching . 35
11 Synchronization method: the loose synchronization and the tight synchronization 36
12 Components of the improved debugging tool design 39
13 Multiple instances of KLEE reside in an independent native process on one simulation machine 41
14 A symbolic execution tree showing on-demand branching . . 42
15 Showing the class diagram of ExecutionState and related classes 44
16 Showing the class diagram of ExecutionState after modification 44
17 Showing the class diagram of Executor and related classes . . 47
18 Components of the improved debugging tool design 48
19 An example showing the communication procedure 50
20 The AUTOSAR communication stack integrated with the CAN transport layer protocol 53
21 Two scenarios where packet loss triggers assertion failures . . 55
22 Node A(sensor) send a periodic signal to Node B(actuator) . 56
23 The distributed view containing redundant execution states . 58
24 Execution time of the debugging tool changes with the number of execution states of Node A 59
25 Memory usage of the debugging tool changes with the number of execution states of Node A 59

List of Tables

1 Performance of two debugging tool designs in packet loss user case 55
2 Number of the test cases generated by the simple tool design and the improved tool design 57
3 Distributed view's references to ktest files 57
4 Performance of the two debugging tool designs in the packet loss use case 58

1 Introduction

In this section, we describe the motivations of this thesis work, the aims and approaches we use in the tool design and implementation. We also present the key contributions in the work and give an overview on the structure of this report.

1.1 Motivation

Since the software industry expanded from the early of 1960's, different kinds of computer programs have dramatically changed ways of people's life. The people are familiar with the general software used in daily office works for processing documents, data analysis and sharing information. Scientists develop and introduce hundreds of scientific software tools to assist laboratories, to verify their guesses and to boost new findings. Other industries combine the computer hardware and the embedded software with physical and mechanical parts to create more advanced commercial products to improve the living quality.

The automotive industry is one of the traditional industries enthusiastically embracing complex and innovative IT systems, particularly in-vehicle embedded systems, in the recent time. These automotive manufacturers wish to provide not only vehicular functions but also car-human interactions to consumers. It is appropriate to declare that the innovations in the automotive industry are driven by integrating digital hardware, programmable processors and software into vehicles. Given these facts, there is no reason to doubt that software plays a vital role in the quality and performance of vehicular products.

Nowadays' Software systems contain thousands of lines of code. They are usually created by a team of programmers and the team could consist of from several engineers to hundreds of them. The process of developing such systems becomes more and more difficult to be managed. It is extremely hard to verify their correctness for both general-purpose programs and embedded programs with critical missions. If the software is faulty, users could have uncomfortable experience such as a frozen system, files missing or privacy leakage. In the worst case, the bugs in a safety critical software system could lead to disasters, damages, and even loss of life. In 2005, the Toyota Motor Corp recalled about 75,000 Prius gasoline-electric hybrid cars. The bugs in the software caused the dashboard warning light to come on and the gasoline engine to shutdown occasionally.

Researchers and developers have paid attentions on the quality of software since the widespread use of digital components in the automotive industry. However, several special technical obstacles and challenges make it difficult to assure the correctness of automotive software:

- **Mutiple software resources**

In the modern development process of advanced generations of vehicles, a large amount of components, including mechanical parts and digital parts, are outsourced to third-party companies [8]. Automotive manufacturers intend to optimize their supply chains, achieve rapid prototyping, and reduce production cost. Meanwhile, vehicle manufacturers usually do not have access to the source code of software components due to the confidential reasons. It is difficult for them to know if the outsourced code satisfies the requirements or has potential bugs. Thorough testings, in aspects of both functionality and reliability, should be performed on the software components before integrating them into existing systems.

- **Distributed character**
  Modern vehicles heavily rely on embedded networks, compared with those cars equipped with isolated hardware and software years ago. These ECUs locate beside sensors and actuators, execute their dedicated tasks and exchange information with other controllers. This distributed design of digital components modularizes the production of cars while introduces non-deterministic behaviors such as packet loss into the entire system.

- **Realtime constraint**
  The software system in a vehicle usually is event-based. The execution order of its tasks follows a predefined scheduling policy. The time domain is one of the primary considerations regarding to system reliability. The system designers should define timing properties of tasks to avoid deadline missing and fully utilize the processor's resources.

The points mentioned above do not include all the sources of flaws in vehicular digital system designs and implementation code. This fact makes it challenging to find potential bugs in both automatic and manual ways. There are only a few available tools supporting automatic testing on such complex embedded systems [19]. As a result, it is necessary to develop a debugging tool differing with existing ones.

1.2 Aim

In this thesis work, we address automatic testing problems in vehicle embedded software. More specifically, we investigate the feasibility of integrating the symbolic execution method into debugging tool designs for an industrial in-vehicle software platform standard named AUTOSAR. This tool shall not only detect common program bugs, e.g. wrong pointer, on one single node but also warn possible software flaws caused by network communication among ECUs. From the view of this tool's users, one of the requirements

is to provide an easy-to-use debugging approach without significantly modifying the source code of AUTOSAR applications or changing AUTOSAR behaviors.

1.3 Approach

We choose the symbolic execution approach to achieve the relatively high code coverage during software testing. An existing symbolic execution engine named KLEE is the base of this debugging tool for AUTOSAR platforms. Academic researchers have accepted it in the system reliability field during the recent years. We motivate the choice of KLEE as the basic block of the debugging tool as following:

- **Universality**
  KLEE is built upon the LLVM (formerly *Low Level Virtual Machine*) infrastructure. Theoretically speaking, KLEE is able to test programs written in any language supported by this compiler infrastructure. Because it works as an interpreter of LLVM bit code. AUTOSAR is mainly written in C programming language while it contains a small fraction of hardware specified assembly code. With slight modifications on AUTOSAR code, KLEE gains the capability to perform automatic testing on this embedded software platform.

- **Automation**
  One of the goals of this thesis work is to simplify the process of debugging AUTOSAR systems and applications. One of the important features of KLEE is that it can use the program code under testing to generate complex test cases. With this feature, testers can focus on designing testing strategy instead of being stuck in tedious manual testings. KLEE provides various commands to configure the debugging environment conveniently.

We also choose the Arctic Core embedded software platform as the targeted system. The most important reason for this choice is that the Arctic Core is open-sourced under the GPLv2 license; thus, we have the freedom to modify and test AUTOSAR systems depending on the demands. For the convenience of understanding the report, we call the targeted system AUTOSAR instead of Arctic Core.

1.4 Scientific contribution

To the best of our knowledge, applying the symbolic execution engine KLEE to debugging AUTOSAR applications has not been done before. This thesis work explores the internal structures of both KLEE and the AUTOSAR platform, and bridges the gaps between them. This debugging tool supports

several failure models that are used to find hidden bugs in the AUTOSAR CAN communication stack.

1.5 Outline

We have given a short introduction to the testing methodology and the testing tool foundation in section 1. In section 2, we introduce essential concepts and more details on the symbolic execution, KLEE and AUTOSAR standard.

Section 3 gives an overview of general testing methods existing in the automobile industry and previous related work.

We first propose a simple debugging tool design in Section 4 to demonstrate the fundamental problems in this thesis work. The solutions to these problems are motivated. Then, we further modify the debugging tool based on the observations in the previous section. This improved debugging tool is presented in Section 5.

Section 6 provides detailed information about the implementations of the system designs. Both of the debugging tool designs are evaluated in Section 7. We also compare their performance under the different situations.

2 Background

This section gives more details on the background of this thesis from two knowledge domains, the concept of symbolic execution and the concept of the AUTOSAR standard.

2.1 Symbolic Execution

2.1.1 Test methodology overview

The development of testing methods has great impacts on the software industry. Testing software functionality and reliability used to rely on tedious manual efforts of the Q&A team before the software is released as a product. In such a manual approach, testers need to follow a testing plan containing a set of predefined test cases. Then, they collect and document the results, report to the developers to correct the problems. However, this approach is not suitable for complex software that takes arbitrary inputs from peripheral modules or communicates through interactive network, considering the extremely huge number of possible test cases. During the past decades, the research of the testing method on the automatic software has made the great progress to solve this problem. Innovative techniques and tools are created to increase the productivity. In general, these methodologies consist of two categories, static testing and dynamic testing, both with strengths and weaknesses:

- **Static testing**
  The static testing approach focuses on the structure of code instead of the run-time state of a program. It can reveal a variety of errors and bad programming practices in code, including but not limited to uninitialized variables, misuse of local variables and a piece of code that cannot be executed with any input data. On the other hand, this approach suffers from intrinsic limitations. For example, the static testing method cannot detect the faulty access into an array since the array indexing is a run-time behavior.

- **Dynamic testing**
  The dynamic testing approach monitors the execution state of a program. The dynamic testing framework collects information about the executed statement count, the validity of variables and the control-flow history during the running of the program. The correctness of the program under testing is either verified during its execution or analyzed in the post-stage. This approach allows the testers have clearer visions of the run-time performance of a program. The drawback of dynamic testing is that it expects testers to feed the programs with proper input data to trigger a possible erroneous execution path.

The automatic testing method, like the random input string generation(Fuzz) [20], could simplify this task but has the risk missing the critical cases leading to the crash of a program.

2.1.2 Concept

Symbolic execution is introduced to the automatic software testing three decades ago. With the progress in the algorithmic research, the availability of powerful computing machines, this testing technique has gained popularity in the real-world software analysis. NASA's Symbolic Java Pathfinder and Stanford's KLEE are the debugging tools complying to this method.

Symbolic execution is a branch of dynamic testing, but it distinguishes with other approaches in this category. A symbolic execution engine executes the program on symbolic input data values instead of concrete ones. The variables associated with symbolic input data and computed outputs of a program are presented as symbolic expressions. In symbolic execution engine, *symbolic state* has the knowledge of the mappings between variables and corresponding symbolic inputs. At the same time, the symbolic execution engine maintains a *path condition*(PC) [22] to keep track of constraints that turn the program control flow to a particular path. PC is updated at each conditional statement containing symbolic expressions, by evaluating the alternative symbolic states of this conditional statement to $true$ and $false$. If the updated path condition becomes unsatisfiable, the symbolic execution engine stops along this path while continues on the other reachable path. A test case is generated by solving the associated PC when an execution state of the program terminates. Each test case contains concrete values assigned to the symbolic input data. The program can traverse along the same execution path as the symbolic execution procedure if it executes on these concrete input values.

Compared with other software reliability validation approaches, the symbolic execution method tests the program on a set of values as input data and explore execution paths simultaneously in an efficient way. Besides, the logic structure of the program is explored during the execution.

2.1.3 Example

In order to have a better understanding of how the symbolic execution engine works, a piece of example code are showed in list 1. It is needed to point out that line 16 and line 17 are not actual initialization methods to make variables symbolic. Different symbolic execution engines symbolize variables in their own way.

The symbolic execution engine steps through program instructions after marking the variable x and y as symbolic. As explained in the symbolic execution concept, program execution states are branched at each conditional

Listing 1: Symbolic execution example

```
1  int compare(int x, int y){
2          if (x > y){
3                  z=2*x;
4                  if (y > z){
5                          return 0;
6                  }else{
7                          return 1;
8                  }
9          } else{
10                 assert(false);
11         }
12 }
13
14 int main(){
15         x = symbolic_value();
16         y = symbolic_value();
17         compare(x,y);
18         return 1;
19 }
```

statement:

1. `initialization`. Assuming the variable x and y are initialized to the symbolic value X and Y, also the path condition, PC, is initialized to $true$.

2. `x > y`. The execution engine updates PC of the `then` statement to $true \wedge X > Y$ when it sees the comparison between symbolic values; it also generates a new path condition, PC', for another branch, `else` statement ,and initializes it to $true \wedge X <= Y$. The constraint solver checks their satisfiability. These two execution states are explored since both PC and PC' are true. z is assigned a symbolic value $2X$.

3. `assert(false)`. This statement indicates a failed assertion encountered by the symbolic execution engine. In this case, the current execution state terminates in an early stage and a test case is generated.

4. `y > z`. Similar to step 2, the symbolic execution engine also updates the path condition on this conditional statement. For the `then` statement, PC is updated to $true \wedge X > Y \wedge Y > 2X$, and PC' is initialized to $true \wedge X > Y \wedge Y <= 2X$ for the `else` statement. In this case, PC is unsatisfiable while PC' is satisfiable. The symbolic execution engine terminates the execution branch of the `then` statement and generates a test case to warn the unreachable code fragment.

A *symbolic execution tree*, in Figure 1, visually describes the execution paths explored by a typical symbolic execution engine. The leaf nodes represent the execution states being able to trigger test case generation. The inner nodes are transient program states along the execution paths.

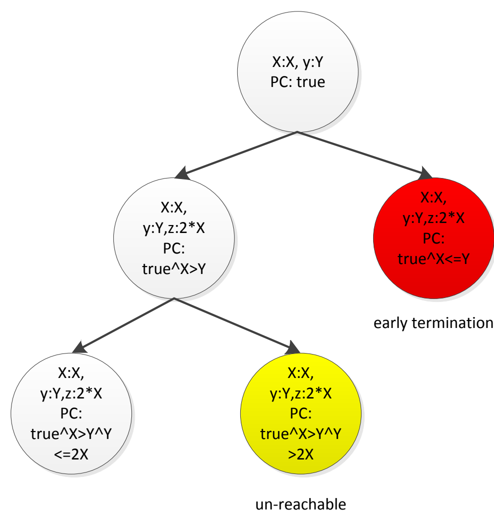


Figure 1: Symbolic execution tree illustrating program example

2.2 KLEE

KLEE is a symbolic execution engine created by Cadar Cristian, Engler Dawson [10]. It is a redesign of EXE [11], a symbolic execution tool with an emphasis on debugging system programs. KLEE has become an open-sourced program downloaded by numerous users from the industry and the academia since June 2009 [1]. KLEE intends to detect potential faulty operations while achieve the high code coverage on the varieties of programs. Experiments show that KLEE succeeds in finding bugs in GNU COREUTILS utility suites, embedded system distributions and the HISTAR operating system kernel.

KLEE is built upon the LLVM infrastructure, and it functions as an interpreter of LLVM assembly code. Each execution state of a program has its register presentation, stack, heap, program counter and path condition. KLEE behaves like an operating system similar to UNIX except that it supports the symbolic execution. The mechanism of KLEE allows the concurrent exploration of the execution paths based on the internal logical structures of programs. In order to achieve good performance of testing, KLEE employs a lot of tricks and design choices on the optimization.

- **Compact state representation**

One of the inefficiencies of the symbolic execution is the quickly growing number of cloned execution states, which results in great consumption of the memory. Instead of mapping each execution state into a native OS process, KLEE employs the copy-on-write strategy at the memory object level. These states share some parts of a heap until the content of the memory object changes.

- **Query optimization**
  An NP-complete logic always leads to complex queries. It costs the majority of the execution time to solve these constraints. There is no exception for KLEE. The designers of KLEE believe that the fastest query is an empty query; hence, they simplify the generated queries before sending them to the constraint solver.

  `Expression rewriting` is a common technique in a compiler design. The expressions are usually reduced to a simpler form by algorithmic operations: e.g. The expression $2x - x + 5 > 7$ could be transformed into $x > 2$.

  `Constraint set simplification` takes advantage of the fact that the constraints on the path condition become more specific as more constraints added at each branching instruction. KLEE sets a wider constraint to a more concrete one. For example, assuming the constraint $x > 10$ already exists in the constraint set; and another constraint $x > 20$ is added later on. If KLEE notices this situation, it substitutes the first constraint with the logic value *true* since any value greater than 20 must be larger than 10.

  `Implied value concretization` enables KLEE to consume less time by processing a concrete value instead of handling a symbolic input. KLEE assigns an actual value back to the memory cell of the symbolic input when a constraint implies so. For example, considering a constraint like $2 * x == 10$, KLEE infers that x is 5 and treats the value of x along the following path as a concrete expression.

  `Constraint independence` sets a bundle between the constraints associated with the different symbolic variables. The constraint solver may only need a subset of constraints to determine a simple query. KLEE identifies those constraints, which refer to the symbolic variables in the query sent to the constraint solver. For example, assuming a query $x == 15$ and two symbolic variables x and y with the constraint $x > 10, x < 20, y > 30$, KLEE only considers the first two constraints and ignores the third one $y > 30$ since it is irrelevant to this query.

  `Counter-example cache` explores the satisfiable relations between the subsets of constraints and the supersets of constraints. Firstly, an unsatisfiable subset of constraints implies that the original set is not

satisfiable either. Secondly, the subsets of constraints are also satisfiable if the original set of constraints is satisfiable. Finally, the solution for a satisfiable subset can be easily verified if it also makes the original constraint set satisfiable. It is because that solving a query is an NP-complete problem.

- **State scheduling**
  KLEE explores multiple execution states of a program with different inputs, in a virtually concurrent way. KLEE chooses one candidate from the current execution states, loads run-time information and interprets the next instruction. To cover more code and avoid getting stuck in a particular execution state, KLEE chooses state by switching between two heuristic selection algorithms:

  `Random path selection` always starts from the root of the symbolic execution tree, traverses it to reach one leaf representing a candidate state. KLEE randomly chooses from the two descendants of an internal node and follows this path until it hits the leaf. This algorithm completely outperforms the random execution state selection algorithm in two ways. First of all, it favors the execution states with less added constraints since these states locate at the high level of a symbolic execution tree. At the same time, the randomness of traverse path selection assists this algorithm to avoid the starvation of a rapidly branching state.

  `Coverage-optimization search` focuses on improving code coverage. The metric weighted in this heuristic method is the minimum distance to the uncovered instruction.

Besides, KLEE has good supports for interacting with the environment. It simulates function calls by *models* implemented in C and generates function call failures to exercise program in such scenarios.

2.3 AUTOSAR concept

This subsection provides a background overview of the AUTOSAR standard and its different layers containing critical software components.

2.3.1 AUTOSAR background

Modern luxury vehicles heavily rely on internal E/E comprising of up-to 70 Electronic Control Units(ECUs). Many critical mechanical and hydraulic components are completely replaced by advanced realtime embedded E/E systems. They are usually the selling points in the market and the key elements differentiating with other vehicle products. Around 90% of the automotive innovation happens in this area. And the cost of the development

of IT systems could take up to 30% of the overall cost of a commercial car [8]. This evolution in the automobile industry is not only driven by the severe competitions among vehicle manufacturers but also the demands of consumers. E/E systems are able to improve the overall quality of vehicles in these aspects:

- **Vehicle dynamics**
  Driving dynamics refer to the behaviors and states of moving vehicles. Together with necessary mechanical actuators, E/E systems have the capability to do the engine management, body and brake control, etc. Besides, E/E systems also handle the situations where the outside environment suddenly changes or the emergency occurs. Modern vehicles react quickly and precisely to minimize the damage to vehicles. Some modules such as the air bag control module can protect drivers from hurts.

- **Driving assistance**
  A driver assistance system consists of sensors, computation equipment and actuators. The sensors, like the GPS and the distance radar, continually collect data from the road path and send them to more powerful computers via networks. Centralized machines compute these data to control executors, making the driving experience smooth.

- **Non-functional legal requirements**
  Most governments have restrictions on vehicle products, such as the restriction on the carbon emission of vehicles. To solve this environmental problem, the hybrid engine propulsion system is built on advanced energy and electronic technologies. Some types of vehicles even achieve the zero emission during the operation [18].

- **Comfort and convenience support**
  Apart from classical vehicular functionalities, modern automobile vehicle manufactures also make a large investment in improving vehicular comfort. For example, the E/E system inside the air conditioning system adjusts the temperature in an energy efficient way.

The rapid progress of in-vehicle E/E systems makes modern vehicles smarter and eco-friendly ,but it also brings challenges to the automobile industry. Today's vehicle manufacturers reduce operating expenses via global supply chains. However, this business model makes the requirement specification and the verification process extremely difficult. Independent suppliers have their unique solutions built on different hardware, design methodologies, interfaces and development tool chains. Hence, if a vehicle manufacture attempts to switch the supplier of one component, the integration process may fail due to the possible incompatibility. To address this problem, vehicle

manufacturers, OEM suppliers and automotive development tool providers propose an open and standard software architecture framework, AUTomotive Open System Architecutre(AUTOSAR). The philosophy of cooperation on AUTOSAR standard is based on a common consideration: cooperate on standards but compete on implementations [23]. This automotive standard emphasizes on the cooperation among different stakeholders in partnership while promotes competitions on innovative vehicular applications.

The AUTOSAR software infrastructure aims to standardize the interfaces between applications and underlying basic software modules running on different hardware platforms. It supports unified application function libraries. Introducing the AUTOSAR standard into the automobile industry brings changes and benefits to software development in this domain:

1. Scalability and exchangeability of software modules are provided across the different suppliers and the hardware platforms.
2. The development, modification and integration throughout the life cycle of ECU application products are highly flexible and manageable.
3. The workload of building the distributed in-vehicle software system is comparably low since the details about hardware across vehicles are hidden.
4. Early detection of errors in the functional scope is facilitated. Vehicle manufacturers can deliver products with high quality and reliability.

2.3.2 AUTOSAR architecture

In the AUTOSAR standard, a layered, modularized and hardware-independent architecture is proposed to reduce the structural complexity existing in E/E systems. OEM suppliers and vehicle manufacturers build innovative software components based on the AUTOSAR infrastructure for both functional and safety purposes. This architecture design enables the Model-Based Design that is widely accepted and used by engineers in the software industry.

An AUTOSAR system is comprised of a set of software components(SWCs). In the early stage of development, detailed specifications on the automotive functionality allows the engineers abstract from implementation details of entities. The development tools turn such abstraction of models into concrete software code by choosing AUTOSAR basic software(BSW) components. Basic software components encapsulate hardware resources and provide them to software components through a transparent middle-ware layer named Runtime Environment(RTE). Combined with the ECU's description files and the system constraint description files, these tools can map RTE and BSWs to ECUs. Figure 2 shows a variety of AUTOSAR layers and modules in a high level.

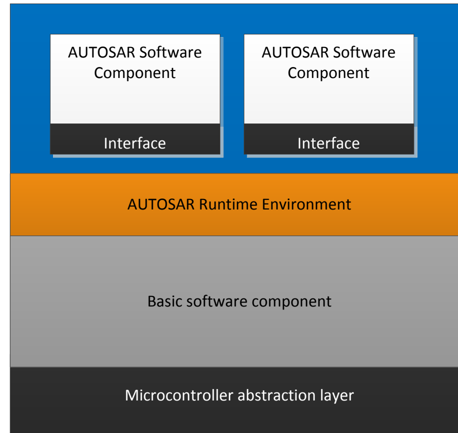


Figure 2: Layers of AUTOSAR architecture

2.3.2.1 Software component

Software components(SWCs) serve as the basic blocks of an AUTOSAR *application* distinguishing from another concept, *infrastructure*. A software component has to be 'atomic' and resides on one ECU although it may contain a large part of one automobile functionality. Another important feature is that one software component shall not know the locations of other components. The cluster of software components is interconnected by explicit 'connectors' in implementation code. The communication can happens through different types of embedded networks.

2.3.2.2 Virtual function bus and Runtime environment

Virtual Function Bus(VFB) is the abstract concept of the RTE in Figure 2. It provides software components with a view of virtual hardware and abstract communication methods. SWCs exchange data via standardized interfaces, without considering neither ECU types or in-vehicle communication networks. In the description file of VFB, each involved software component binds with static ports. These ports can be configured with different distributed communication patterns, including the Client-Server model and the Sender-Receiver model.

The Runtime environment(RTE) of the AUTOSAR system implements the abstract functionality specified in the VFB description. Automatic development tools generate RTEs for each AUTOSAR system. They share

most of code for a desired communication service. Meanwhile, they are customized for hardware platforms and device drivers.

2.3.2.3 Basic software component

The Basic Software Component(BSWs) is a set of AUTOSAR moduels below the runtime environment. It provides the infrastructural functionality to SWCs of an application. BSWs are designed to be an adapter gluing the standardized upper interfaces with the ECU-dependent services. The basic software component layer is divided into five sublayers, as in Figure 3.

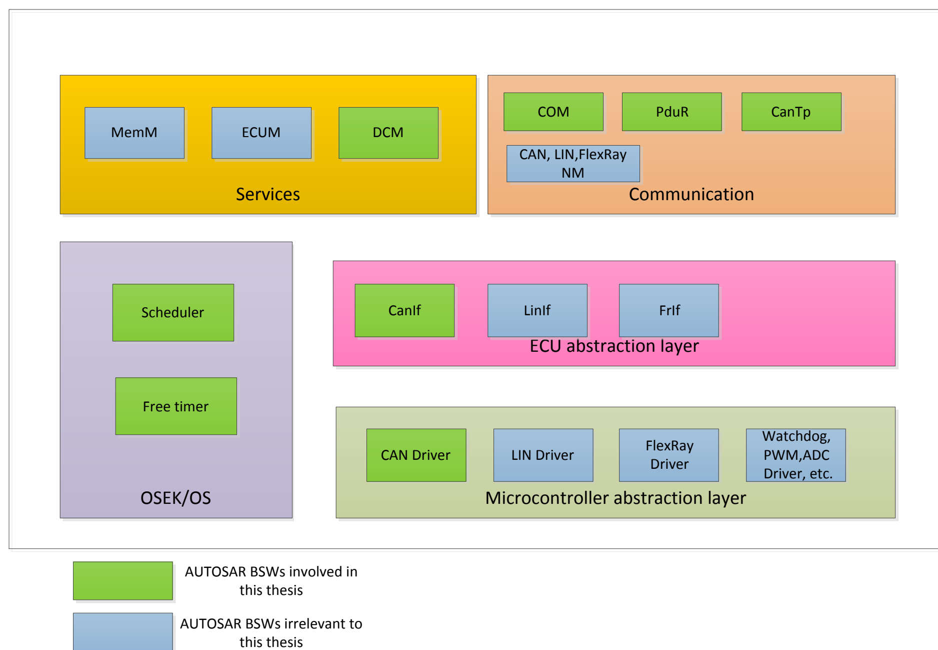


Figure 3: Layers of AUTOSAR basic software component

- **Services**
  This sublayer of the BSW consists of several system services, e.g. the memory management, the diagnostic mechanism.
- **Communication**
  Communication sublayer is responsible of managing the vehicular communication networks, e.g CAN, LIN, FlexRay.
- **Operating System**
  The operating system of the AUTOSAR standard is compatible with

the industrial OSEK OS [16] for the maintainability of legacy automotive applications. The extended OSEK OS part supports advanced safety techniques at runtime. The AUTOSAR OS has several features:

1. It only supports the static configuration. With this feature, the memory is allocated when ECUs load binary code while the dynamic allocation is not allowed.
2. Real-time performance is one of the primary considerations in the AUTOSAR OS design.
3. Runtime protective functions regarding the memory and time are provided.
4. Low-end ECUs are fully supported without using external resources.

- **ECU abstraction layer**
  The ECU abstraction layer is an intermediate layer above the drives in the microcontroller abstraction layer. It provides APIs of the internal/external peripheral devices and maps them to microcontroller's pins.

- **Microcontroller abstraction layer**
  The AUTOSAR applications and the basic software components have no direct access to ECU's registers. The Microcontroller Abstraction Layer (MCAL) distributes all accessing requests to hardware interfaces, including I/O, EEPROM, the analog/digital converter (ADC), etc. Basic software components send commands, together with ECU independent values, through the standardized interfaces of MCAL. The MCAL uses a notification mechanism to inform different AUTOSAR components about the states of their requests.

2.3.2.4 Communication stack

The communication stack is one of the most important parts of basic software components. It supplies AUTOSAR applications with unique interfaces regardless of the types of underlying networks. In this thesis, we focus on the CAN communication network. Figure 4 depicts the modules associated with the CAN communication stack. The main functions and features of these modules are summarized as following:

- **COM**
  COM [6] provides signal oriented data interfaces to the RTE to send and receive data. It packs signals to I-PDUs(Interaction Layer Protocol Data Units) to transmit to lower modules and unpacks received I-PDUs to extract signals.

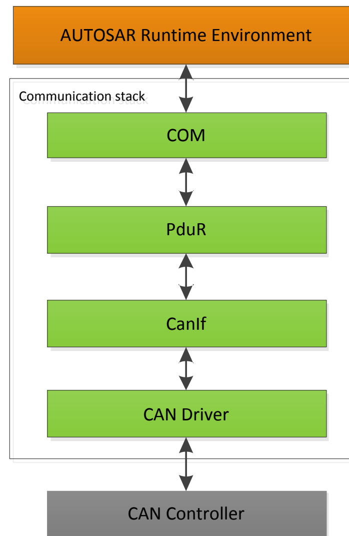


Figure 4: AUTOSAR CAN communication stack

- **PduR**
  PduR [7] is responsible of routing the I-PDUs of upper layer modules(e.g. COM) to the statically configured destinations. It forwards I-PDUs to either another upper layer module or communication hardware abstraction modules(e.g. CanIf).

- **CanIf**
  CanIf [4] is the abstraction layer of CAN communication device driver. It provides the APIs of CAN hardware functionalites to the PduR module or directly to the Com module. CAN L-PDU(Link Layer Protocol Data Unit) is the basis of the data processing and the notification mechanism of the CanIf module.

- **CAN Driver**
  CAN Driver [3] is the lowest layer of the CAN communication stack. It handles the accesses to several CAN controllers belonging to the same CAN Hardware Unit. It has the ability of forming the CAN frames transmitted on the CAN bus, extracting the payload from CAN frames and notifying events by calling callback functions.

3 Related work

Verifying and validating AUTOSAR software components are important issues through the life cycle of products. There exist numerous specifications of the testing process in the industry and research works in academia. In general, these works comply to the basic principles of software testing but are also tailed for automobile embedded systems. This section gives an overview of current testing tools developed for automobile applications. These tools can be categorized into two classes, model simulation tools and code execution tools with respect to system output behavior [9]. Besides, in order to show the success of KLEE in other domains, one KLEE-based tool is also introduced.

3.1 Code execution tool

Detecting software faults while executing programs on a platform is a common testing method in the development of embedded programs. Such testing tools compare the behaviors of vehicular systems, based on predefined inputs, with expected ones. If the result of comparison violates the mapping relation between input and pre-computed output, testers are alerted about potential software issue. Code execution can happen on a hardware platform or in a microcontroller simulator.

AutoPlug [13] is a test-bed for debugging automotive electronic system. This system consists of multiple subsystems including vehicle dynamic simulator, a CAN-bus based ECU network and runtime diagnosing middleware. This system runs on an ECU as a background task to perform result verification, controller implementation verification and sensor value validation.

3.2 Model based tool

AUTOSAR is a framework fully supporting the model based design method, which assists managing complex data relations inside vehicle digital systems and resolving version conflicts. AUTOSAR software component model is developed under the MATLAB Simulink/Stateflow framework in the design phase, as in [21]. The generated models can be verified and validated in a higher abstraction level by model simulation. The model languages and checker are used to verify the design choices and automatically generate test cases needed in such high level simulation.

AutoMOTGen [15] is a model based automatic test case generation tool. It uses an intermediate representation, SAL, and its associated tools to process and verify the Simulink/Stateflow model. The goal of AutoMOTGen is to generate a test suite consisting of input-output mapping sequences while achieving user specified coverage goal.

In [17], the widely used Simulink Design Verifier in the automotive industry and a general purposed SPIN model checking tool are evaluated in terms of their usability in verification of the vehicular software model. Particularly, the correctness of an AUTOSAR memory management module, NVRAM, is verified with these two different approaches.

3.3 KleeNet

KleeNet [25] is a KLEE-based debugging tool for Wireless Sensor Network(WSN), which is an infrastructureless and self-organized network comprising hundreds of resource-constraint devices. It aims at debugging WSN applications before deployment. KleeNet detects potential corner-cases leading to pre-termination of programs. The research work of KleeNet makes several contributions to WSN debugging domain:

- **Coverage**
  KleeNet supports debugging WSN application on unmodified code with symbolized environment input values, enabling a high code coverage.

- **Non-determinism**
  Non-deterministic events, such as packet loss, duplication and corruption, are common in distributed network. KleeNet guides the program execution flow to corner cases, and find potential bugs there.

- **Distributed assertion**
  The distributed state of wireless sensor network is checked by a technique named distributed assertions. KleeNet generates test cases for each execution path where distributed assertion is violated. Similar to the requirement of white-box testing, using this KleeNet feature needs inside knowledge of the program.

- **Repeatability**
  KleeNet can reproduce the execution by using automatically generated test cases.

KleeNet has good performance in finding four bugs in one of the popular WSN applications, μIP TCP/IP stack. On the other side, KleeNet also suffers from issues, such as great run-time and memory complexity, demand for manual effort and limited application domain.

4 Design

In this section, we present the architecture of this KLEE-based debugging tool for the AUTOSAR software platform in a high level view and leave discussions on its implementation details to section 6. In the thesis work, we first propose a naive system design revealing and solving the difficulties existing in applying KLEE to debugging the AUTOSAR system. Then, we optimize the tool design based on the observations which are demonstrated in section 5. This report documents both of the system architectures to provide more information and hints for further development of such a debugging tool.

4.1 System overview

The system design of this KLEE-based tool follows a direct and simple rule: the structure of the AUTOSAR system shall be retained as much as possible. We follow this design rule for two reasons:

1. A complete AUTOSAR stack supports a variety of features without compromise, improving the chances to trigger a faulty execution path.
2. The AUTOSAR system is a complicated system where different layers are connected via complex interfaces, making decoupling it into several independent layers difficult. This design rule treats the AUTOSAR stack as a black box requiring only a limited number of basic external supports.

The basic components of the debugging tool are shown in Figure 5. The *kernel* represents the symbolic execution engine interpreting the LLVM bitcode compiled from an AUTOSAR application. It is the extension of KLEE with supports for the AUTOSAR operating system and network functionality. The *AUTOSAR runtime* is an intermediate layer between the symbolic execution engine and the AUTOSAR platform. It serves as a wrapper of a set of KLEE special functions and provides necessary interfaces customized for the AUTOSAR platform.

The *Operating system model* and the *network model* are the basic components adapting the KLEE kernel to the AUTOSAR standard. Generally speaking, the OS model in this debugging tool kernel simulates the behaviors of the OSEK/OS basic software components, which enables debugging standalone programs. The network model allows the AUTOSAR programs on multiple ECUs exchange messages, especially CAN bus messages. A set of *failure models* are designed to inject symbolic events during the execution of a program and drive it into a rarely explored path.

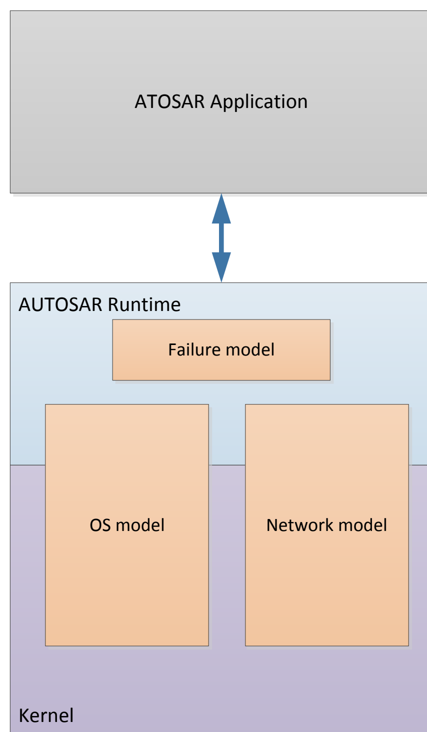


Figure 5: Components of a simple debugging tool design

4.2 OS model

The OS model in the design overcomes the incompatibility between the architectures of the KLEE kernel and the AUTOSAR operating system. The basic entities of the AUTOSAR OS are implemented in this model by modifying KLEE as well as adding an adapted AUTOSAR runtime layer.

4.2.1 AUTOSAR requirement

The operating system in the AUTOSAR standard is an embedded OS with basic hardware management functionalities and extended real-time features. It is derived from an industrial legacy operating system specification, OSEK, for microcontrollers used in the automotive domain. As mentioned in section 2.3.2, every application software component built on the AUTOSAR platform is independent of each other. They compete for the ECU resources with other program routines. These AUTOSAR OS entities can be categorized into two types:

- **Task**

The AUTOSAR OS uses a task to present one software component with real-time parameters. A task can be either executed periodically or scheduled according to a predefined scheduling table. In a typical AUTOSAR application with complicated interactions of software components, there are two types of tasks, the *basic task* and the *extended task*. Both of them have *running*,*ready* and *suspended* states, and only the extended task can enter *waiting* state.

- **Interrupt service routine**
  The Interrupt Service Routine(ISR) plays an important role in an event-driven embedded system, allowing ECUs communicate with external devices. There are two types of interrupts, the hardware interrupt and the software interrupt. An AUTOSAR program handles multiple interrupts from different resources. In this thesis work, we focus on two important ones, the hardware timer interrupt and the CAN communication interrupt.

The task entity and the ISR entity both have the user specified priority. The AUTOSAR OS manages the runtime context and assigns ECU resource to tasks during the execution.

4.2.2 Existing model and modification

KLEE is a symbolic execution virtual machine with only one process hosting a basic program. Hence, it cannot symbolically execute an embedded program like an AUTOSAR application. A set of operations, such as task creation, task switching and task termination, are missing in KLEE. Besides, unmodified KLEE is only suitable for traditional sequential programs, failing to support the event-driven programming paradigm.

The kernel of KLEE and runtime environment are re-designed to implement the task entity and the ISR entity. In this naive approach, the interrupt service routine model locates in the AUTOSAR runtime layer while the task model resides inside the KLEE engine as depicted in Figure 6.

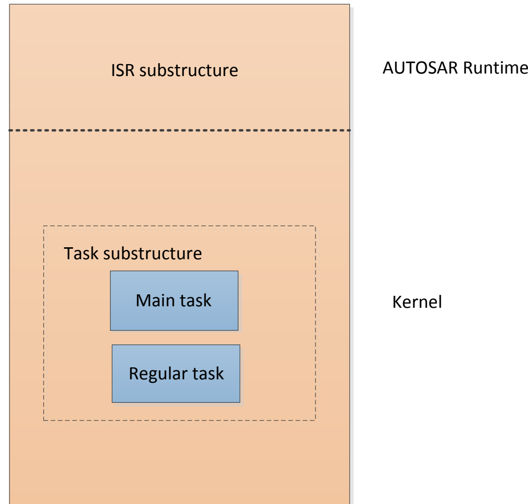


Figure 6: Architecture of the OS model

- **Task substructure**
  The software components of an AUTOSAR application are mapped into the *regular tasks*. The other parts of the AUTOSAR stack, including the alarm and the scheduler, reside in a *main task*. It has the same structure as a regular task but serves as the default one in the system. The basic APIs are provided to the AUTOSAR runtime to create, switch and terminate internal tasks inside the KLEE kernel.

- **ISR substructure**
  The interrupt service routine is simulated by substituting hardware interrupts with software interrupts. They are viewed as discrete events driving the whole system. A *discrete event queue* manages these simulated interrupts, as in Figure 7.

  This event queue handles two types of interrupt events, the *periodic event* and the *one-shot event*. The periodic event has the capability to insert itself back to the tail of the event queue. The one-shot event removes itself after termination. In the AUTOSAR system, the timer interrupt behaves as a periodic event since it advances the AUTOSAR OS *counter* repeatedly. On the contrary, the communication interrupt is a one-shot event inserted by the other program routine. Additionally, the APIs of this discrete event queue are outside the KLEE kernel and linked with AUTOSAR applications.

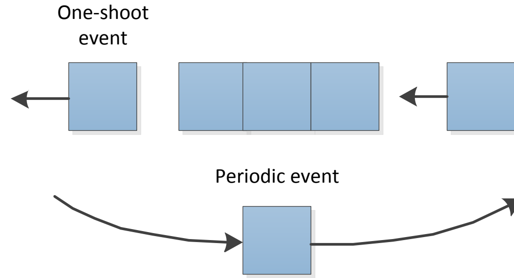


Figure 7: The discrete event queue containing the periodic event and the one-shot event

4.3 Network model

Communication is one of the most important features of AUTOSAR platforms, allowing software components on multiple ECUs cooperate on the same automotive functionality. The network model aims at debugging AUTOSAR applications communicating with each other. The non-deterministic issues existing in the ECU network are further explored with it.

4.3.1 Network topology

The network topology is essential to every debugging tool supporting networked systems. Compared with the scenarios in the other research work such as KleeNet, we simplify the network topology in the AUTOSAR system for three reasons:

- An automotive functionality usually needs a few number of ECUs although there could be up to 70 nodes in a vehicle. For example, displaying the current speed of a vehicle needs the ECUs of the speed detecting module and the driver panel module.
- A less complicated network topology reduces the difficulty to design a high-performance debugging tool.

The network scalability issue is taken into consideration, but we pay more attntion to small-scaled AUTOSAR systems. In addition, the complex network structure containing multiple communication methods is beyond this thesis work.

4.3.2 Simulation machine

To debug a networked system, the tool should have the capability to execute programs on several ECUs. Only one program under test can be symbolically executed on KLEE. There are two promising approaches to solve this problem:

- One KLEE instance on one machine. This approach models the behaviors of the networked system inside one KLEE instance running in a native OS(e.g. Linux) process.
- A distributed execution engine consisting of multiple KLEE instances running on several machines. These KLEE instances exchange execution states of programs through the network infrastructure such as TCP/IP. [12] uses a similar approach to share the testing workload among KLEE instances.

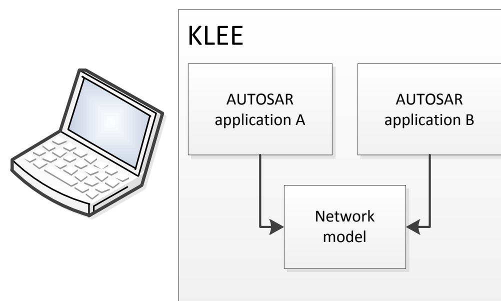


Figure 8: Multiple nodes reside in one KLEE instance on one machine

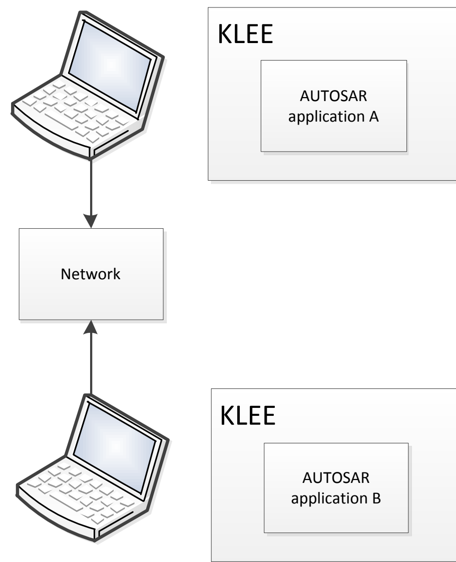


Figure 9: Multiple KLEEs locate on independent machines

Figure 8 and Figure 9 demonstrate the difference between these two approaches. In this debugging tool design, the first approach is preferred for two reasons. First of all, it would be annoying to deploy the testing environment and settings for all the machines. Secondly, it is unrealistic to use one machine for one ECU program since there could be several nodes in the network.

4.3.3 Logic node

A *Logic node* is introduced into this KLEE-based debugging tool to model multiple AUTOSAR nodes. This system design retains one of the KLEE's drawbacks. It cannot interpret several LLVM bit code files at the same time. To solve this problem, the source codes of different nodes are merged into one file but separated by the *logic scope* as in List 2. The network activity is simulated by the interaction between the different copies of this merged program.

Listing 2: Merge code of two nodes to one file

```
1 IF I am Node A THEN
2         Do job A
3
4 IF I am Node B THEN
5         Do job B
```

During the execution, every logic node has an individual OS model that provides the separation of their memory address space. Several logic nodes coexist in a single program, and a set of OS tasks form a logic node.

4.3.4 Dedicated branching

KLEE branches the execution paths of a program and concurrently explores every possible execution state. This symbolic debugging tool for the the AUTOSAR system focuses on two scenarios triggering such execution path branching:

1. The program on the single node branches due to the conditional statements associated with the symbolic value. It is a feature inherited from the original KLEE.

2. The execution path of a networked AUTOSAR system branches due to the non-deterministic events such as packet loss.

This naive system design uses a branching strategy called *dedicated branching*. It forks the runtime states of each node regardless of the branching type. Figure 10 gives an example of dedicated branching. In this example, a networked AUTOSAR system consists of two nodes and one of them transmits a packet. The symbolic execution engine branches their execution paths for twice. The first branching is due to a conditional statement while the second branching is caused by a symbolic packet loss event. All the execution states of the nodes are forked internally.

4.3.5 AUTOSAR communication

A variety of communication methods, including CAN, FlexRay and LIN, are used in the AUTOSAR standard. We choose the CAN stack as the underlying network. The bottom layer of the AUTOSAR communication stack is the CAN Driver interacting with the CAN controllers. The simulation of communication over the CAN bus contains two phases. First, the packet is copied from the TX buffer of the sender to the RX buffer of the receiver. Then, the sender triggers the RX interrupt on the side of the receiver.

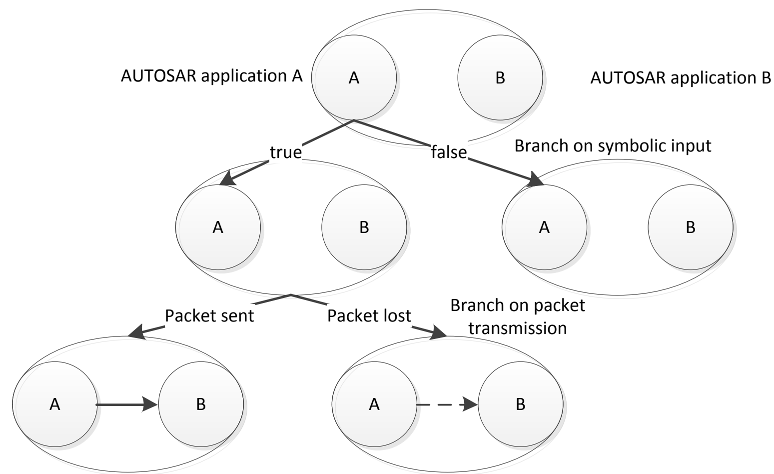


Figure 10: A symbolic execution tree showing the dedicated branching

4.3.6 Synchronization

Timing issue is critical in the AUTOSAR real-time system since its behaviors are driven by discrete timer interrupts. For example, the function calls to transmit and receive packets are wrapped in periodic tasks. The AUTOSAR OS calls these functions when the system clock reaches a certain value. For example, Node A and Node B boot up at the same time. If Node A issues packet transmission when the system clock is 10, Node B should receive and process this packet when the value of its system clock is equal or greater than 10. This feature makes it necessary to synchronize the system clock of each node in the network. It is completely different from debugging non-realtime programs such as a client-server application [14]. In general, there are two ways, as in Figure 11, to synchronize the nodes in the realtime network.

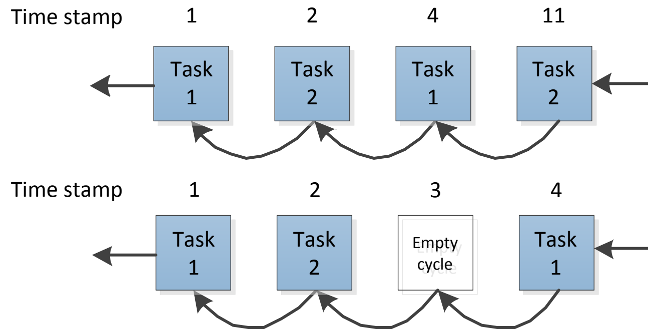


Figure 11: Synchronization method: the loose synchronization and the tight synchronization

- Loose synchronization
  In this approach, a heap sorts the execution order of nodes according to the time stamp to trigger their candidate periodic task. The node is re-scheduled after the task period if its current task terminates.

- Tight synchronization
  The system clocks of AUTOSAR nodes are well synchronized in a way of *tick-by-tick*. It means that every node has to increase its system clock by 1, then waits for the other nodes to do the same operation.

KleeNet uses the first approach to modify the Contiki system [24]. However, tight synchronization is more suitable for this AUTOSAR debugging tool. The AUTOSAR OS combines the periodic execution and a predefined scheduling table to manage tasks. It is difficult to handle these two mechanisms separately in the loose synchronization approach. This method also ignores some critical parts of the AUTOSAR OS stack. It is wise to leave the work of managing tasks to a complete AUTOSAR OS stack by driving it from the lowest layer.

4.4 Failure model

As a part of the debugging tool design, it is important to define the types of failures that can be detected. Although KLEE provides several built-in failure models such as the memory access failure, it does not address the failures encountered in distributed scenarios. Most of them are caused by non-deterministic events like packet loss or random input data. Three possible failure models in the AUTOSAR networked system are proposed and discussed in the following section.

4.4.1 Packet loss

Although underlying networks, such as CAN and FlexRay, are designed to be reliable in the physical layer, the exchanged packets can get lost in the presence of malicious ECUs. This non-deterministic event potentially leads to strange or undesired performance issues of vehicles. This debugging tool drives the system to two paths by using a packet loss model. One path is that the CAN frame is transmitted correctly while the other one is that this debugging tool discards the CAN frame. Test cases are generated for any buggy scenario if packet loss causes the system to crash.

4.4.2 Remote invalid path invocation

KLEE can trigger a faulty execution path in a standalone program by following the control flow of it. In a networked system, one common case is that the execution flow of one node is controlled by another node through the communication network. For example, the sensor module collects the environment data, then process it and send the computed results to the actuator. The actuator performs the actions such as acceleration on vehicle dynamics. The invalid input data harvested by sensors would lead to erroneous computation results and trigger a faulty execution path on the side of the actuator. This debugging tool implements this failure model and warns application developers to notice the invocation of an invalid path.

4.4.3 Assertion

This debugging tool uses assertions to verify the correctness of the behaviors of AUTOSAR nodes. It is not only designed for validating a standalone program but also for checking the consistency of a networked AUTOSAR system. In this thesis, we consider a consistency checking example, whose condition is a simple statement of 'if the received value equals to the sent value'. This assertion is mainly used to find the buggy part of a communication stack or protocol when packet loss happens.

5 Improved Design

In the previous section, we propose a naive debugging tool design to demonstrate the concept of applying the symbolic execution engine KLEE to debugging AUTOSAR systems. This KLEE-based symbolic execution engine simulates most of the behaviors of the AUTOSAR applications running on either single ECU or multiple ECUs. The OS model and the network model are designed to provide system infrastructures to hosted programs. The failure model contains a preliminary study of possible debugging functionalities that can be enriched in the future work.

This simple system design reveals several fundamental problems existing in developing a KLEE-based symbolic execution tool. It also employs the approximate solutions to solve these problems. These approximate methods have negative effects on either the performance or usability of this debugging tool although they reduce the workload of the system design in the early stage. It can be further modified to achieve:

- improvement of usability of this KLEE-based symbolic execution engine. This target assists the future integration of this debugging tool into the existing AUTOSAR application development environment.
- better performance during the symbolic execution of programs. It is important to save the computation resources on the simulation machine and reduce the computation time.

We present some improvements on the previous simple design as well as the motivations for these different design choices in the remaining part of this section.

5.1 Overview

This improved symbolic debugging tool is more complex, but its basic structure does not change. It still comprises the AUTOSAR runtime, an OS model, a network model and a failure model. They have been re-designed and modified to some extent. Figure 12 gives an overview of this system design.

At the first glance, the only difference between these two designs is that all the components of the OS model and the network model now reside in the kernel of KLEE. This choice leads to another design philosophy: the architecture of the debugging environment should be invisible to AUTOSAR applications. In other words, an AUTOSAR application interacts with the debugging tool via a few interfaces instead of liking with a runtime library such as a discrete event queue in section 6. This approach accelerates the execution speed because interpreting the code by KLEE is much slower than performing the same operation in the kernel of the symbolic execution engine.

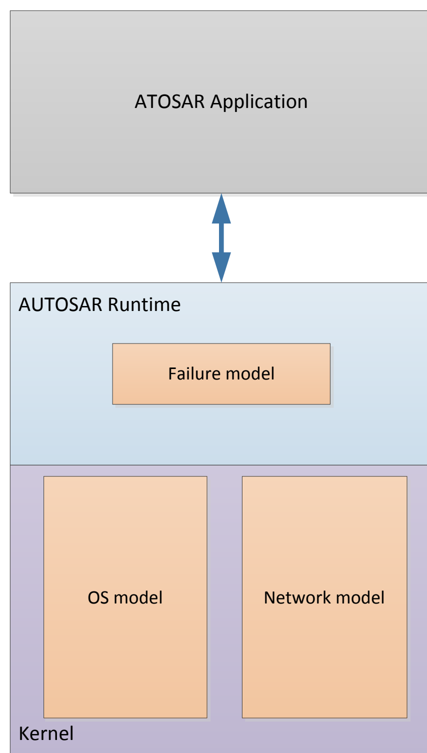


Figure 12: Components of the improved debugging tool design

5.2 OS model improvement

The OS model of this debugging tool design is not improved dramatically. The task substructure of the OS model remains to support concurrent tasks while the interrupt service routine(ISR) substructure changes. In the previous design, the ISR is simulated by discrete events inserted by the tested program, which results in large execution overhead outside the kernel of KLEE.

To address these problems, a *special task* is introduced here. This OS entity uses the task substructure of the OS model to host an interrupt routine. It has different features from either the main task or the regular task. Every special task has its *mirror* that is initialized with the interrupt routine entry. The kernel of KLEE provides an interface to switch the execution path to the interrupt routine by awaking a sleeping special task. The kernel of KLEE spawn a brand-new sleeping special task from its mirror *automatically* after termination. This approach precisely simulates the behaviors of interrupts and reduces the overall complexity of the OS model. The interrupts generated by packet transmission use this approach.

5.3 Network model improvement

5.3.1 Multiple nodes hosting

In section 4.3, the programs of multiple nodes are merged into one executable file and simulated in one instance of KLEE. This approach is easy to implement but has limitations in debugging the AUTOSAR networked system.

The ECUs in an automobile run completely different programs performing their own functionalities. It is unlike the wireless sensor network in which the sensor nodes usually execute the same program. The AUTOSAR application developers pick different basic software components from the library and connect them via extremely complicated static configuration files. These files contain variable declarations and definitions. The logic node approach handles the case where the programs share most of the basic software components and use the same configuration files. It is not suitable if ECU programs are more independent in the aspect of the system structure and the configuration. For example, two ECU programs may have different configuring values for the mode of packet transmission. This scenario requires the merged file to provide a variable array hosting multiple values. It brings troubles to extract the correct value for each ECU program.

The intuitive way solving this problem is to execute each program by a dedicated KLEE engine. Meanwhile, it is impossible to allocate one simulation machine to each ECU program. Thus, we propose a new way to host multiple nodes to fulfill these requirements, as in Figure 13.

The main idea is running multiple instances of KLEE on the same simulation machine, allowing the concurrent execution of completely different programs. It deserves to be specially noted that the instances of KLEE do not run in different Linux processes. Instead, they reside in the same native process. It is much easier to synchronize the execution of nodes. In addition, we avoid using POSIX system infrastructures to manage programs' branched execution states.

5.3.2 Communication channel

This multi-instance KLEE engine makes it difficult to read from or write to the memory object of the other program. It is Because each instance of KLEE has an independent address space and a different LLVM bit code module of the program. The *channel* is introduced into the design to assist the communication between the instances of KLEE. It is similar to the link between two communication ports except that the ends of a channel are memory objects. This approach provides a way to transfer value among KLEE instances and eliminates the risk of the faulty access to the memory object in other node's address space.

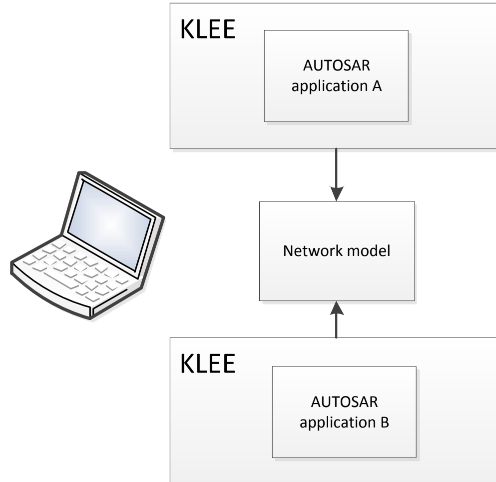


Figure 13: Multiple instances of KLEE reside in an independent native process on one simulation machine

5.3.3 Distributed view

A *distributed view* manages a set of execution states having the same *communication log* for all the nodes in the AUTOSAR networked system. In a distributed view, the execution states of the same node are branched only due to the conditional statement on symbolic value. The validity of the distributed view cannot hold if one execution state of a node transmits a packet. The execution engine creates a distributed to solve this invalidity. In the extreme case of no communication activity, there is only one distributed view, and every node executes and branches independently.

Distributed view test files log the references to the test case files generated by KLEE for the execution states in one distributed view. They are useful to trace the branching history of each node and investigate the communication history of the whole system.

5.3.4 On-demand branching

Dedicated branching is used in the previous system design, but it is not efficient enough to handle the state explosion caused by non-deterministic events such as packet loss. The *on-demand branching* strategy, based on distributed view concept, partially eliminates the redundant execution states existing in dedicated branching. Figure 14 gives an on-demand branching example.

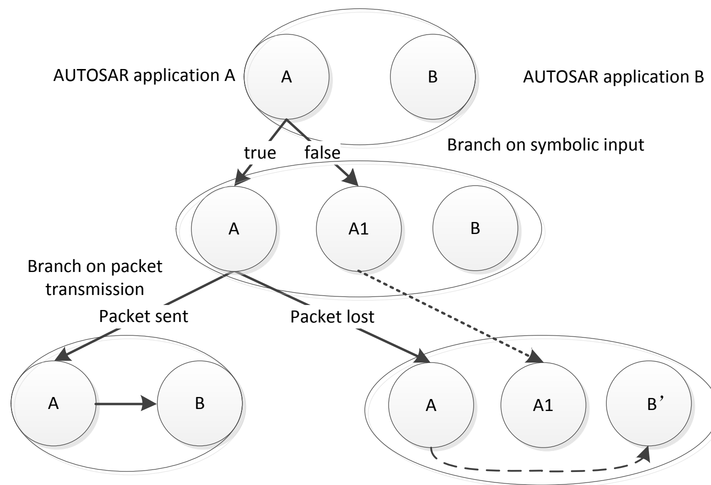


Figure 14: A symbolic execution tree showing on-demand branching

A conditional statement on a symbolic value triggers the first branching of Node A. The execution engine generates an execution state A_1. Then, it inserts it into the current distributed view without forking the execution state of Node B. The second branching is due to packet transmission issued by execution state A. The distributed view is not valid anymore since state A and A_1 have different communication logs. A new distributed view is created to solve this conflict. The execution engine forks the execution states of Node B, and inserts it into the new distributed view. The on-demand branching strategy reduces the total number of execution states in the system, compared with the dedicated branching strategy.

6 Implementation

In this section, we provide the implementation details of previously described designs. The descriptions include the structure of the debugging tool system, the interfaces provided by the runtime environment, and explanations about the modifications on both KLEE and AUTOSAR code. We start the description from the simple system design in section 6.1. Then, we explore details of the improved design in section 6.2. These two designs share subsystems such as the task model and underlying communication methods.

6.1 Implementing simple design

6.1.1 Existing model

The most important class of KLEE is `ExecutionState` representing the state of a program under testing. `ExecutionState` is initialized by loading the LLVM bitcode file of the program. During the execution, the current `ExecutionState` branches at every conditional statement that can be true or false. `ExecutionState` contains several essential classes:

- **Address Space** Address space keeps track of the state(e.g. value) of the 'memory object', which simulates the cells of the memory.
- **Program Counter** Program counter maintains a pointer pointing to an instruction in the LLVM code module. There exist two program counters, the current program counter and the previous program counter.
- **Constraint Manager** The constraints on the path condition computed from symbolic variables are maintained by a constraint manager. It has the capability to optimize the collected constraint expressions.
- **Stack** The Stack is a vector of stack frames, keeping track of caller instructions, the entry of the current function, local variables and arguments.

6.1.2 System

The modifications on `ExecutionState` implement the task substructure of the OS model, the logic node concept and the synchronization concept in the network model. The relations between these components are depicted in Figure 16.

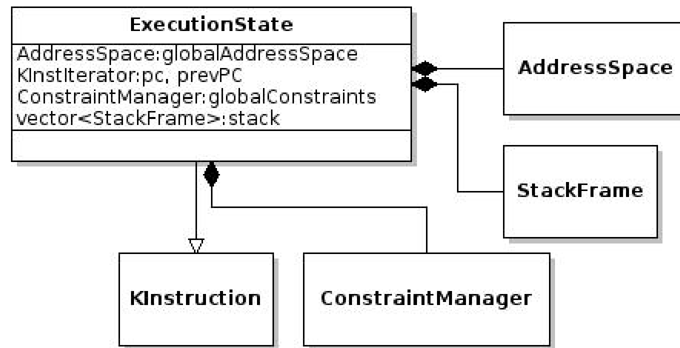


Figure 15: Showing the class diagram of ExecutionState and related classes

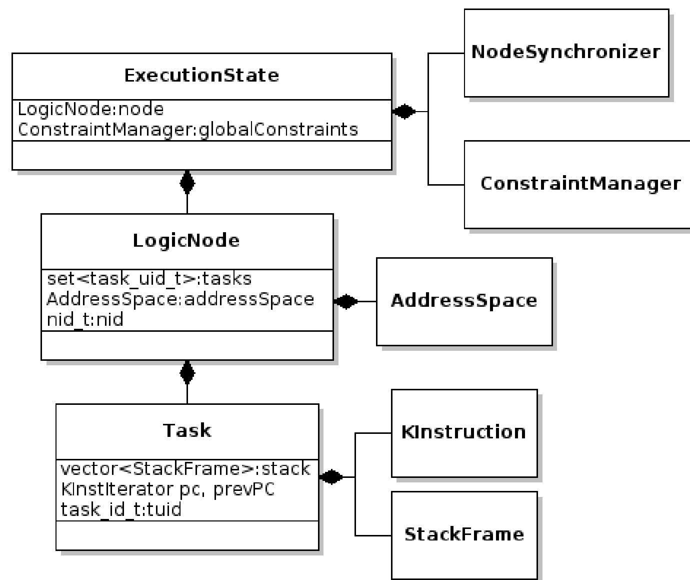


Figure 16: Showing the class diagram of ExecutionState after modification

6.1.3 Data transmission

Data transmission is simulated by transferring the value of one memory object to another memory object. Particularly in the AUTOSAR system built on the CAN network, the data are transferred between the sending buffer and the receiving buffer. The transfer process can be split into three phases:

1. Extract the address space of the sender node and resolve the object state of the memory object associated with the sending buffer. The pointers of `Expr` representing state value are copied into a vector-based temporary data storage.

2. Extract the address space of the receiver node and grasp the the

writable object state of the memory object associated with the receiving buffer. The pointers of the state value `Expr` from the data storage in the previous step are written into the writable object state.

3. Merge the constrains of the sender's execution state into the receiver's execution state.

One tricky problem about packet transmission is that data can be symbolic values. It is different from the case where the data are concrete values. In the memory object state model, an array of `uint8_t` values derived from `ConstantExpr` is used to store concrete values. On the contrary, an array of `Expr` pointers serves as the unique identifiers in memory for itself. To solve this problem elegantly, two smart functions `ObejectState::read8` and `ObjectState::write`, that can distinguish between concrete and symbolic value, are employed in the first and the second step of data transmission.

6.1.4 Special functions

The special function is an important feature provided by KLEE, allowing programs talk to the execution engine. We define a set of special functions:

- `klee_get_context` AUTOSAR nodes use it to obtain the node id allocated by the symbolic execution engine.
- `klee_task_create` This special function takes a task id, an entry point of a task routine and arguments as its parameters. It will create an initialized task object in the kernel.
- `klee_task_terminate` If a periodic task completes its execution, it has to call this special function to eliminate the corresponding task entity inside KLEE.
- `klee_task_switch` This special function takes the id of the expected task as its parameter and forces the execution flow of KLEE to switch to this task.
- `klee_handle_transmit` The purpose of this special function is to transfer the data from one memory object to the other memory object. Hence, it requires the address of both source and destination variables, the ids of both source and destination nodes and the length of transferred data.
- `klee_add_node` This special function is used to fork the execution state of a program containing logic nodes.
- `klee_node_sync` An AUTOSAR application uses it to synchronize its execution with other nodes.

6.1.5 Runtime

The interrupt service routine substructure of the OS model is implemented in the AUTOSAR runtime layer. The implementation has similar APIs as in the AUTOSAR system. List 3 shows the key elements in implementation.

Listing 3: Code fragment of the software interrupt

```
typedef enum {
        VIRTUAL_TIMER_INTERRUPT,
        VIRTUAL_IRQ_TYPE_CAN0_TX,
        VIRTUAL_IRQ_TYPE_CAN0_RX
}virtualInterruptType;

virtualInterruptType interrupt_type;
unsigned int incoming_interrupt_flag;

#define GEN_VIRTUAL_INTERRUPT \
        void * virtual_interrupt_table[MAX_NUMBER_VIRTUAL_INTERRUPT]=

void attach_virtual_interrupt(TaskType tid, virtualInterruptType vector);

void virtual_interrupt_entry(virtualInterruptType);

void virtual_set_interrupt(virtualInterruptType set_interrupt_type,
                uint8 receiver, uint8 sender);

void Os_VIsr(void *pcb);
```

In the AUTOSAR system, an interrupt routine is managed by a process control block(pcb) as a task. It has the highest priority over other OS entities such as periodic tasks and scheduled tasks. To 'install' an interrupt routine, the id of its pcb has to be mapped to an interrupt type. The function `attach_virtual_interrupt` helps to register an interrupt routine in a virtual interrupt table. At the same time, `virtual_interrupt_entry` extracts the interrupt routine associated with a type value and call `Os_VIsr` to execute it.

In section 4.2.2, it is pointed out that there are two types of interrupt events, the periodic interrupt event and the one-shot interrupt event. One tricky question is how a node issues an interrupt to other nodes. A *polling* strategy is employed here. The flag `incoming_interrupt_flag` is set from 0 to 1, and the variable `interrupt_type` is configured by the other node if it issues an interrupt. Each node checks the flag for each clock tick. If the flag is set, this node creates an interrupt event and inserts it into the private event queue.

6.2 Implementing improved design

6.2.1 Existing model

In this improved system design, more classes of KLEE are explored and modified. Apart from `ExecutionState`, another important class is `Executor`, which interprets the bit code file of a program, maintains the program state, sends constraint queries, and interacts with the environment. It consists of many classes, but only the essential ones are shown in Figure 17. In `Executor`, a process tree remembers the execution states associated with a program. The leaves of the `PTree` are current candidate execution states that can be scheduled. The other internal nodes present the branching history of a program. A `searcher` selects a state from the leaves of the process tree and hands it to the executor to step one instruction. KLEE firstly uses its own expression language [2] to express the constrains on a program. Then KLEE transforms them into equivalent STP expressions that can be solved by an STP solver being wrapped in `TimingSolver`.

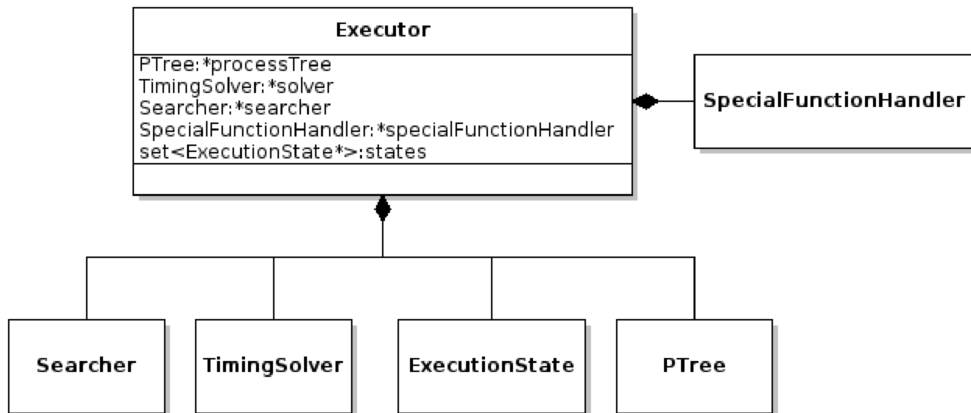


Figure 17: Showing the class diagram of Executor and related classes

6.2.2 System

The further modified structure of the debugging tool retains the essential models, including the task model implemented in `ExecutionState`. Recall that this design shall support multiple instances of KLEE in one native Linux process. To achieve this goal, we allow several `Executor` instances coexist in one KLEE program. These objects of `Executor` class share user configurations for KLEE, but work independently as fully functional symbolic execution engines. This thesis work aims at verifying the idea of applying KLEE to debugging the AUTOSAR program. Hence, the system design only considers two instances of `Executor`. Figure 18 gives an overview of the debugging tool structure after modifications.

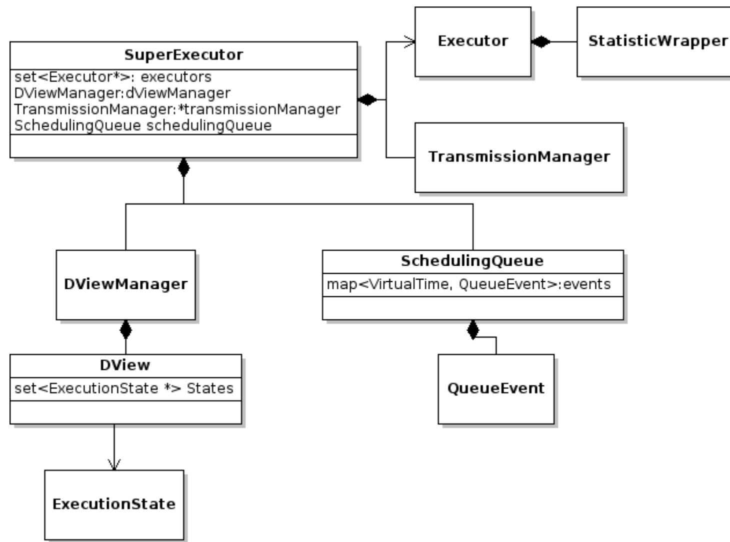


Figure 18: Components of the improved debugging tool design

This system comprises `SuperExecutor`, `Executor`, `DViewManager`, `TransmissionManager`, `StatisticWrapper` and `SchedulingQueue`.

- `SuperExecutor` is the center of the execution engine. It has global knowledge about each executor as well as the execution states of every program under testing.
- `DViewManager` is the implementation of the distributed view model.
- `TransmissionManager` is used to handle the operations related to packet transmission.
- `StatisticWrapper` records the runtime statistic information about the program module.
- `SchedulingQueue` stores the execution states of each node according to their time stamps.

6.2.3 Dynamic switching

This is an implementation trick that deals with global variables in both KLEE and interface code of the STP solver. Although the major parts of KLEE and STP solver are written in C++, there are variables keeping track of the global status. Adding more executor instances to KLEE demands the separation between their statuses. To address this problem, a trick called *dynamic switching* is used. It means that these global variables retains in the code but are forced to point to the corresponding local wrappers when the

executor is switched. The `globalStatisticWrapper`, `theStatisticManager` in KLEE and `BEEV::ParserBM`, `BEEV::GlobalSTP` in the STP solver employs this trick.

6.2.4 Special function

The special functions used to manipulate task entities of the OS model are as same as in the previous design. We define five additional special functions for AUTOSAR applications:

- `klee_schedule_state` This special function takes a time interval as its parameter. The node calling it is scheduled to execute after this time interval.
- `klee_port_bind` It defines a communication channel at the beginning of each AUTOSAR application. Several ports, which refers to variables, can be bound to the same channel id.
- `klee_transmit_via_channel` This function takes the source channel id, the destination channel id and the length of the transferred variable as its parameter. The broadcast mechanism of the CAN bus motivates that all the ports of other nodes receive the value.
- `klee_special_task_create` This special function creates a special task for each interrupt service routine.
- `klee_special_task_terminate` It is added to the end of each special task. The purpose of this special function is to terminate and re-load the interrupt routine automatically.

6.2.5 Communication and On-demand branching

The communication between different instances of KLEE is via a *channel* whose ends are memory objects. An option, `'debug-packet-loss'`, enables debugging the AUTOSAR application in different scenarios including normal packet transmission and packet loss. The communication procedure under the latter case is demonstrated by a simple example in Figure 19. In this example, node A has branched into state A and A' on a conditional statement. State A attempts to send a packet to Node B, making the current distributed view invalid. We use an algorithm to solve this conflict and handle the packet transmission.

1. **Alias query**
   `Alias` refer to the execution states of other nodes in the same distributed view as the sender. The alias are marked with green color in Figure 19.

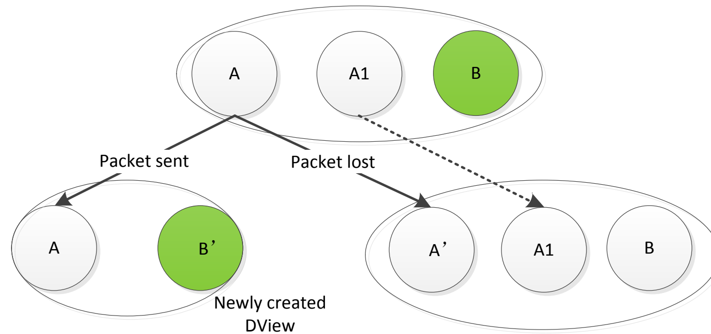


Figure 19: An example showing the communication procedure

2. **Alias forking**
   `SuperExecutor` forks the execution state of the alias found in Step 1 and notifies the executor. The newly forked execution states are inserted into the process tree in the executor.

3. **DView expanding**
   DView is expanded in this step to solve the communication log conflict between two execution states of Node A. In this step, an empty distributed view is created first. Then the forked execution state of alias, B', is inserted to this DView, together with current execution state of the sender, A.

4. **Transmission handling**
   `Targets` are the execution states that should receive the packet. They are either forked execution states of alias in the packet loss scenario or just alias in the normal packet transmission scenario. The data transmission framework in the simple system design is reused here. As the discussion in section 5.2, a `special task` is used to trigger the CAN Rx interrupt. Hence, `SuperExecutor` calls `invokeSpecialTask()` of each target at the end of packet transmission.

6.2.6 Scheduling and synchronization

`SchedulingQueue` schedules and synchronizes the execution states. This class provides two key interfaces to manipulate this map-based event queue:

- `reScheduleExecutionState`
  It is wrapped in a special function, `klee_schedule_state`. An AU-

TOSAR node uses this function to delay the execution of itself after n ticks to synchronize with other nodes.

- `updateSchedulingQueue`
  `SuperExecutor` collects the newly branched or terminated execution states in every `Executor`, then updates the scheduling queue by stepping through each element from collected execution states.

In the original KLEE, the `searcher` of an executor selects an execution state from the process tree using different searching strategies. In this system design, it is `SuperExecutor` that selects an execution state for every executor. List 4 shows the code to select an execution state. The execution state with minimum time stamp is popped from the `schedulingQueue` in Line 2. `SuperExecutor` finds the associated executor and feeds it with this execution state.

Listing 4: Code fragment of `SuperExecutor` to select execution state

```
1 while (!schedulingQueue.isEmpty()) {
2         ExecutionState* crtes = schedulingQueue.selectExecutionState();
3
4         NodeId node_id = (crtes->dInfo).getNodeId();
5         Executor* crtExecutor = executor_mapping[node_id];
6         if (!crtExecutor->haltExecution)
7                 crtExecutor->runExecutionState((*crtes));
8         }
```

7 Evaluation

In this section, we evaluate the debugging tool designs proposed in section 4 and section 5. The purpose of the evaluation is to demonstrate the applications of this KLEE-based tool in debugging AUTOSAR programs. We also try to compare the performance of the different system designs from the aspects of their execution speed and memory consumption. We focus on these two metrics because they are usually the major issues existing in the symbolic execution methodology. The extremely large number of execution states explored by a symbolic execution engine makes the execution procedure both time consuming and memory inefficient.

During the period of this thesis work, there is no available real-world AUTOSAR application for the testing purpose. Hence, we present two possible use cases of this KLEE-based debugging tool to show that it can find faults purposely injected into the programs. The applications are constructed on the basic functionalities of the AUTOSAR platform. We use some tricks to simulate the behaviors of the system in extreme cases to push the performance of this system to the limit. Besides, the other researchers working on the same topic can use and analyze these use cases to make further improvement and testing.

7.1 Packet loss

In the klee-related research, debugging the networked system usually means injecting non-deterministic events, such as packet loss, into the system. The debugging tools check the consistency of nodes. This method finds bugs existing in applications such as communication protocol stacks. This use case aims at exploring the performance of this debugging tool when it is used to debug the AUTOSAR networked system with CAN packet loss.

[5] points out that the CAN transport layer protocol is mostly used by vehicle diagnostic systems. However, applying the CAN transport protocol to regular message passing is an interesting attempt to change the AUTOSAR stack. With this modification, the AUTOSAR communication stack can transfers the signal whose message size is larger than 7 bytes. The modified AUTOSAR communication stack is shown as Figure 20.

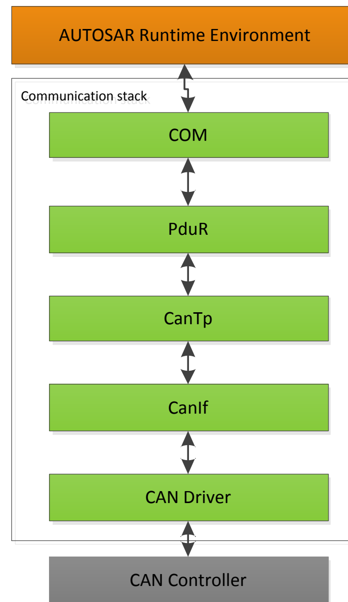


Figure 20: The AUTOSAR communication stack integrated with the CAN transport layer protocol

7.1.1 Experiment assumption

This experiment is designed to demonstrate the capability of this KLEE-based tool to find insidious bugs in the full AUTOSAR communication stack. In order to achieve this purpose, we deliberately make a piece of code in the AUTOSAR communication stack buggy. We apply this symbolic execution engine to the application built upon this faulty AUTOSAR platform. The symbolic debugging tool should be aware of such a buggy part in the implementation code.

In this experiment, we choose to make the interface function, PduR_CanTpRxIndication, between CanTp and PduR buggy, as in List 5.

Listing 5: Buggy code fragment in AUTOSAR communication stack

```
1 void PduR_CanTpRxIndication(PduIdType CanTpRxPduId, NotifResultType Result) {
2         PduR_ARC_RxIndication(CanTpRxPduId, &comRxBuffer.pduInfo, 0x04);
3         comRxBuffer.status = NOT_IN_USE;
4 }
```

PduR_CanTpRxIndication was used by the diagnostic software component, Det, to report the status of the CAN transport protocol, but it has been modified to deliver the data received from CanTp to the upper Com layer. The parameter, Result, indicates the different status of CanTp such as NTFRSLT_E_WRONG_SN, NTFRSLT_E_NO_BUFFER. The code in List 5 delivers

the data to the upper layer without checking the status of `CanTp`, which is assumed to cause severe problems.

To verify this assumption, an AUTOSAR application containing this buggy code is tested with the KLEE-based debugging tool.

7.1.2 Experiment setting

This use case of packet loss contains the scenario where two AUTOSAR nodes are communicating with each other. In this application, Node A sends an 8-byte value to Node B through their AUTOSAR communication stacks. Node B verifies that whether it receives this value correctly in an *assertion*. The debugging tool raises an execution error if Node B receives an incomplete or faulty value.

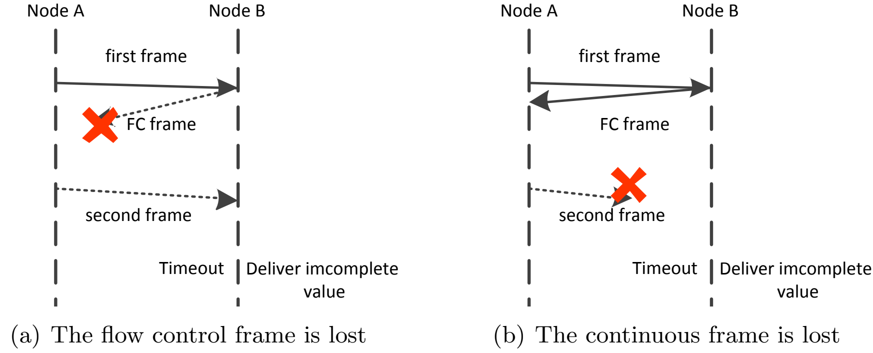
The maximum length of a carried message is set to 8 bytes. The `CanTp` layer splits this message into two CAN message frames. Timeout of `CanTp` is set to 2(= 2000/1000). The application program has to call the main function of `CanTp`, `CanTp_MainFunction`, twice before the timeout period expires.

We use 'klee -debug-packet-loss com_simple/main.bc com_simple2/main.bc' to invoke the improved version of the debugging tool.

7.1.3 Experiment result

We discuss the experiment results from two aspects, generated test cases and performance of the debugging tool.

The debugging tools implementing both design choices generate the test cases including .ktest files. These test cases shows that two scenarios where packet loss happens trigger the the assertion failures as in Figure 21.



(a) The flow control frame is lost (b) The continuous frame is lost

Figure 21: Two scenarios where packet loss triggers assertion failures

In both of the scenarios in Figure 21, Node B receives the first frame of the message but keeps waiting for the second frame. `PduR_CanTpRxIndication` is called because of the timeout of `CanTp`. The status of `CanTp` is `NTFRSLT_E_NOT_OK`. It delivers the incomplete value, without checking the status of `CanTp`, to the upper layer which triggers the assertion failure. To correct the buggy code, the status of `CanTp` should be checked before packet delivery, as in List 6.

Listing 6: Correct code fragment in AUTOSAR communication stack

```
1  void PduR_CanTpRxIndication(PduIdType CanTpRxPduId,
2                          NotifResultType Result) {
3   if (Result == NTFRSLT_OK) {
4           PduR_ARC_RxIndication(CanTpRxPduId, &comRxBuffer.pduInfo, 0x04);
5           comRxBuffer.status = NOT_IN_USE;
6           }
7  }
```

The performance of the debugging tools is shown in Table 1.

Table 1: Performance of two debugging tool designs in packet loss user case

	Execution time/s	Memory consumption/MB
Simple design	0.64	21.10
Improved design	0.48	27.78

It shows that the execution speed of the improved debugging tool design is faster than the simple one. However, it consumes more memory due to the multiple KLEE instances in the same OS process.

7.2 Invalid sensor data

This use case demonstrates an application of one failure model, remote invalid path invocation. It explores the impact of invalid data on the correctness of the AUTOSAR system.

7.2.1 Experiment setting

In this experiment, we consider the scenario where two nodes running AUTOSAR systems are involved in the communication as depicted in Figure 22. `CanTp` is not integrated into the AUTOSAR communication stack. These two nodes cooperate on an easy task: Node A(sensor) receives an environment value, then assigns different computed values to the result. This result is transmitted to Node B(actuator) through the AUTOSAR CAN communication stack. Node B chooses one of the execution paths based on this value after receiving the result.

There is one erroneous execution path leading to the crashing of Node B.

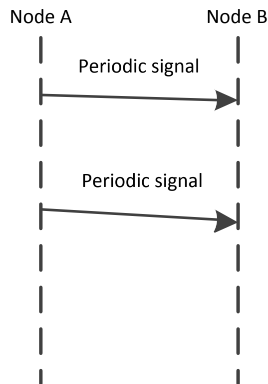


Figure 22: Node A(sensor) send a periodic signal to Node B(actuator)

We highlight two important pieces of code to configure the proprieties of this AUTOSAR application:

1. `SetAbsAlarm` The AUTOSAR application uses it to configure the starting point and the period of the task by associating it with an alarm.

2. `ComTxModeMode` & `ComTxModeTimePeriodFactor` They are the members of the struct `ComIPdu`, which defines the property of the `signal`. It can be used in a periodic way. Compared with the concept of the period of a task, the *period* of a signal expires after a certain number of

invocation of the function Com_SendDynSignal instead of the system ticks.

7.2.2 Experiment result

For the first attempt of the experiment, we set the starting point of the task to 5 and the period of the task to 50 system clock ticks. The period of the Tx signal is set to 10.

Both of the KLEE-based debugging tools successfully find the faulty execution path in Node B. One of the distributed test cases, distributed-Test000003.dvtest, shows that the crashing of Node B also causes the early termination of the execution states of Node A in the same distributed view.

The number of generated cases is shown in Table 2. The simple debugging tool solution only generates test cases for the entire AUTOSAR networked system. The improved solution separately generates test cases for every node in the system.

Table 2: Number of the test cases generated by the simple tool design and the improved tool design

	Node A	Node B
Simple design	3	
Improved design	3	4

The test cases are recorded in different .dvtest files to identify the execution paths of nodes in the same distributed view, as in Table 3.

Table 3: Distributed view's references to ktest files

	Node A	Node B
DView 0		test000002.ktest
DView 1	test000002.ktest	test000003.ktest
DView 2	test000003.ktest	test000004.ktest
DView 3	test000001.ktest	test000001.ktest

We also focus on the execution time and the memory consumption. Table 4 shows results for the both system designs.

The experiment results also show that the simple design outperforms the improved design in the aspect of total memory usage. This phenomenon happens in the scenario where there are communication activities but only a few number of execution states branch on conditional statements. It is due to the complex internal structure of the improved tool design. Besides, on-demand branching strategy brings the extra one execution state of Node B to avoid the communication log conflict.

Table 4: Performance of the two debugging tool designs in the packet loss use case

	Execution time/s	Memory consumption/MB
Simple design	0.75	17.17
Improved design	0.55	21.46

On the contrary, the execution speed of the improved design is faster than the simple design by 62%. The built-in OS model and network model dramatically reduce the workload of the whole system by avoiding interpreting the code of the runtime layer.

7.2.3 Further test and result

The further testing aims at exploring the performance of the debugging tool designs when they are handling the heavy workload of execution path branching. There could be thousands of different execution states caused by symbolic values during the execution of a program if it contains complex conditional statements. For example, in the AUTOSAR system, the program on the sensor side implements a complicated algorithm to process the input environment data, which results in thousands of branches. Currently, we consider a case where only a few branched execution states of Node A send the computed value to Node B, as in Figure 23.

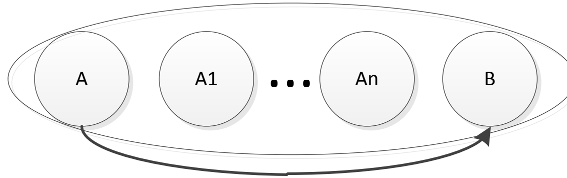


Figure 23: The distributed view containing redundant execution states

To compare the performance of the debugging tool designs, we choose to branch the execution paths of Node A in an *exponential* way. They branch the execution states of Node A for the same times. Figure 24 and Figure 25 describe the changes of execution time and memory consumption with the number of execution states of Node A.

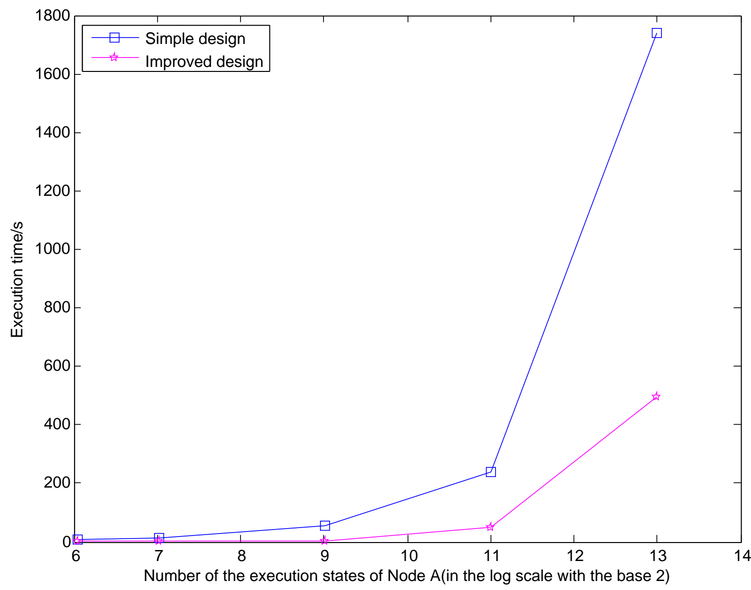


Figure 24: Execution time of the debugging tool changes with the number of execution states of Node A

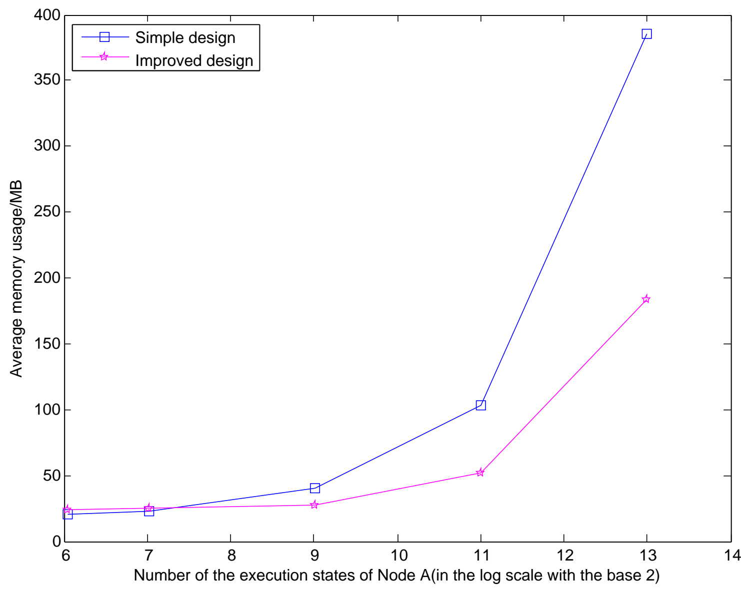


Figure 25: Memory usage of the debugging tool changes with the number of execution states of Node A

The results in Figure 24 and Figure 25 indicate that the improved debugging tool design outperforms the simple one as the number of the execution states of Node A increases. The built-in OS model and network model contribute to the fast speed of the improved debugging tool. The redundant execution states of Node B in every distributed view are removed to save memory resource. Hence, we conclude that the improved debugging tool design accelerates the simulation speed in most of the scenarios. It reduces the runtime memory cost if there exist a large number of branched execution states in distributed views.

8 Conclusion

This section discusses the limitations of the present work, shows the ideas for future research and summarizes this thesis work.

8.1 Discussion and future work

KLEE supports a replay framework to execute programs on the concrete values in test cases. The simple debugging tool design retains this feature since it is inherited from original KLEE, and the whole networked system resides in one execution state. It is difficult to provide the same feature if an execution state represents one node as in [25] and [14]. Currently, the improved tool design does not have this feature either. Adding a replay framework to the improved debugging tool design is a challenge in the future work. This thesis work only considers the scenarios involving two AUTOSAR nodes to investigate the basic concepts. Supporting more nodes requires better designs of the debugging tool kernel and APIs. In this case, there still exist some redundant execution states to be eliminated. Besides, both of the system designs choose to synchronize the AUTOSAR system clocks in a tick-by-tick way. They ignore the possible clock drift, which is also an interesting topic for future improvement. As it is mentioned in the evaluation, we create use cases to test the performance of this KLEE-based debugging tool. It is necessary to test it on programs from the automobile industry by cooperating with the automotive companies.

The Arctic core platform, the implementation code of the AUTOSAR standard, is constantly updated by the ARCCORE AB. The APIs provided by this debugging tool may be outdated in the future, but the methods proposed in the report shall be still applicable. In addition, the automobile industry is enriching the AUTOSAR standard to support more features. For example, the AUTOSAR standard release 4.0 has come into the market. This debugging tool can be further developed to support new AUTOSAR standards.

8.2 Conclusion

In this thesis, we introduce a KLEE-based symbolic debugging tool for AUTOSAR programs. It detects the faulty execution paths existing in the applications built upon the Arctic core platform.

We propose two debugging tool designs to simulate the behaviors of both standalone AUTOSAR programs and a networked system consisting of two ECU nodes. They allow us symbolically execute AUTOSAR applications and automatically generate test cases. We also demonstrate the implementation details of the tool designs. These technical materials not only reveal the structures of this debugging tool but also help other researchers have an

in-depth knowledge of KLEE and AUTOSAR code.

The use cases and evaluation results show that this KLEE-based tool can find the buggy parts of AUTOSAR applications when non-deterministic events happen. The comparison between the different designs indicates that the improved tool design in section 5 is more promising than the simple tool design.

References

[1] The klee symbolic virtual machine. http://klee.llvm.org/.

[2] Kquery language reference manual. http://klee.llvm.org/KQuery.html.

[3] Specification of can driver - autosar. http://www.autosar.org/download/AUTOSAR_SWS_CAN_Driver.pdf.

[4] Specification of can interface - autosar. http://www.autosar.org/download/AUTOSAR_SWS_CAN_Interface.pdf.

[5] Specification of can transport layer - autosar. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_CANTransportLayer.pdf.

[6] Specification of communication - autosar. http://www.autosar.org/download/R2.0/AUTOSAR_SWS_COM.pdf.

[7] Specification of pdu router - autosar. http://www.autosar.org/download/AUTOSAR_SWS_PDU_Router.pdf.

[8] Manfred Broy. Automotive software and systems engineering. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, pages 143–149. IEEE, 2005.

[9] Manfred Broy, Samarjit Chakraborty, S Ramesh, M Satpathy, S Resmerita, and W Pree. Cross-layer analysis, testing and verification of automotive control software. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 263–272. IEEE, 2011.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[11] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. Technical report, Citeseer, 2006.

[12] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[13] Utsav Drolia, Zhenyan Wang, Yash Pant, and Rahul Mangharam. Autoplug: An automotive test-bed for electronic controller unit testing

and verification. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 1187–1192. IEEE, 2011.

[14] Milen Dzhumerov. Symbolic execution of distributed software.

[15] Ambar A Gadkari, Anand Yeolekar, J Suresh, S Ramesh, Swarup Mohalik, and KC Shashidhar. Automotgen: Automatic model oriented test generator for embedded control systems. In *Computer Aided Verification*, pages 204–208. Springer, 2008.

[16] OSEK Group et al. Osek/vdx operating system specification, 2009.

[17] Florian Leitner and Stefan Leue. Simulink design verifier vs. spin–a comparative case study. In *Proceedings of FMICS*, 2008.

[18] Natalia Lestre, Antoine Pouthier, Gregory Nice, and NigelJ. Tracey. Hybrid drives with autosar-compliant control units. *ATZ worldwide eMagazine*, 113(9):22–27, 2011.

[19] Alexander Michailidis, Uwe Spieth, Thomas Ringler, Bernd Hedenetz, and Stefan Kowalewski. Test front loading in early stages of automotive software development based on autosar. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 435–440. IEEE, 2010.

[20] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[21] Gwangmin Park, Daehyun Ku, Seonghun Lee, Woong-Jae Won, and Wooyoung Jung. Test methods of the autosar application software components. In *ICCAS-SICE, 2009*, pages 2601–2606. IEEE, 2009.

[22] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339–353, 2009.

[23] Scott K Peterson. Consideration of patents during the setting of standards. *FTC and DOJ Roundtable on Standard Setting Organizations: Evaluating the Anticompetitive Risks of Negotiating IP Licensing Terms and Conditions Before a Standard is Set*, 2002.

[24] Raimondas Sasnauskas. The cooja/kleenet project. `http://sourceforge.net/p/contikiprojects/code/297/tree/sics.se/coojakleenet/`.

[25] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 186–196. ACM, 2010.