UDK 519.713

Synchronizing and Homing Experiments for Input/output Automata

Kushik N.G. (SAMOVAR, CNRS, Télécom SudParis / Université Paris-Saclay),
Yevtushenko N.V. (Tomsk State University, Institute for System Programming of the Russian Academy of Sciences)
Burdonov I.B. (Institute for System Programming of the Russian Academy of Sciences)
Kossatchev A.S. (Institute for System Programming of the Russian Academy of Sciences)

The paper is devoted to studying the (‘gedanken’) experiments with input/output automata. We propose how to derive proper input sequences for identifying the final (current) state of the machine under experiment, namely synchronizing and homing sequences. The machine is non-initialized and its alphabet of actions is divided into disjoint sets of inputs and outputs. In this paper, we consider a specific class of such machines for which at each state the transitions only under inputs or under outputs are defined, and the machine transition diagram does not contain cycles labeled by outputs, i.e. the language of the machine does not contain traces with infinite postfix of outputs. Moreover, for each state where the transitions under inputs are defined, the machine has a loop under a special quiescence output. For such class of input/output automata, we define the preset synchronizing and homing experiments, establish necessary and sufficient conditions for their existence and propose techniques for their derivation. The procedures for deriving the corresponding (‘gedanken’) experiments for input/output automata are based on the well-studied solutions to these problems for Finite State Machines.

Keywords: *Input/Output Automata, Synchronizing Sequence, Homing Sequence*

1. Introduction

The state identification problem using ‘gedanken’ experiments with Finite State Machines (FSMs) is a long standing problem. The first results were obtained by Moore [10] and have been then improved by many researchers [3, 6, 9]. A (‘gedanken’) state identification experiment with an FSM consists of applying an input sequence to a machine under investigation, observing the output response and drawing a conclusion about initial or current state. If the conclusion is drawn about the

initial state (a state before the experiment) then the experiment is called *distinguishing*. When the conclusion is drawn about the current FSM state (a state after the experiment) then the experiment is called *homing* or *synchronizing*. Depending on the way how an input sequence is applied an experiment can be *preset* or *adaptive*. In this paper, we discuss only preset experiments when an applied input sequence is derived in advance.

There are many applications of such 'gedanken' experiments and a big body of work is developed for constructing preset and adaptive experiments [4, 7, 11]. Most applications are related to decreasing the complexity of deriving a test suite with the guaranteed fault coverage when the specification FSM has homing/synchronizing/distinguishing sequences and there are many papers how such sequences can be derived for deterministic and nondeterministic, complete and partial FSMs [2, 5, 7, 11]. In [8], the authors propose how homing and synchronizing sequences can accelerate/optimize the monitoring of communicating systems. When the initial/current state of an Implementation Under Test (IUT) is known, the set of properties that should be verified at a given IUT state can be dramatically reduced.

However, FSMs have limited capacity when describing software component behavior. The reason is that the next input can be applied only when the FSM under investigation produced an output to the previous input. On one hand, this allows to escape races between inputs and outputs and it is one of the reasons why test suites with the guaranteed test coverage are derived mostly against FSMs. On the other hand, FSM notion does not allow to consider the situations when an output can be produced only after a sequence of inputs has been applied to an IUT and moreover, not a single output can be produced but a sequence of outputs. Such situations can be described when using Input/Output automaton as a model; an Input/Output automaton has the finite number of states but differently from FSMs, transitions between states are labeled not by a pair <input, output> but by a single input or output. To the best of our knowledge there are no investigations on homing/synchronizing sequences for such model.

In this paper, we study the state identification problem for Input/Output automata when at each state, only inputs or only outputs are allowed. We define the notions of homing/synchronizing sequences for such Input/Output automata and adapt the known techniques for deriving such sequences for a new model. Therefore, the main contribution of this paper is the definition of homing/synchronizing sequences for Input/Output automata and the development of techniques for the existence check and derivation.

The rest of the paper is structured as follows. Section 2 contains preliminaries. Techniques for deriving homing and synchronizing sequences for input/output automata are proposed in Section 3. Section 4 concludes the paper and has a brief discussion on the directions of the future work.

We note that this work is partially supported by the Russian Science Foundation (RSF), project № 16-49-03012.

2. Preliminaries

An Input/Output *Automaton* (or an *automaton* in this paper) is a 4-tuple $\boldsymbol{S} = (S, I, O, T_S)$ where S is a finite set of states; I and O are finite non-empty disjoint sets of inputs and outputs, respectively; $T_S \subseteq S \times I \times S \cup S \times O \times S$ is a *transition relation* where 3-tuples $(s, i, s') \in T_S$ and $(s, o, s') \in T_S$ are *transitions*.

In this paper, we consider a specific class of automata for which the following holds:

i) At each state only inputs or only outputs are allowed, i.e. $S = S_1 \cup S_2$, $S_1 \cap S_2 = \varnothing$ and $T_S \subseteq S_1 \times I \times S \cup S_2 \times O \times S$;

ii) The transition diagram does not contain cycles/loops labeled with outputs, i.e. the language of the machine does not contain traces with infinite postfix of outputs;

iii) The machine has a special output $\delta \notin O$ that represents the quiescence [12] at the states where the transitions under inputs are defined; at each state $s \in S_1$, there is a loop under δ, namely $(s, \delta, s) \in T_S$.

As an example of an Input/Output automaton, consider a machine in Fig. 1. The automaton $\boldsymbol{S}$ has five states, namely $S = \{s_1, \ldots, s_5\}$, where $S_1 = \{s_1, s_2, s_5\}$ and $S_2 = \{s_3, s_4\}$. At each state from the set S_1 the automaton accepts inputs i_1 and i_2. However, when the machine is at state s_3 or s_4 no inputs can be accepted and only outputs o_1 or o_2 can be produced.

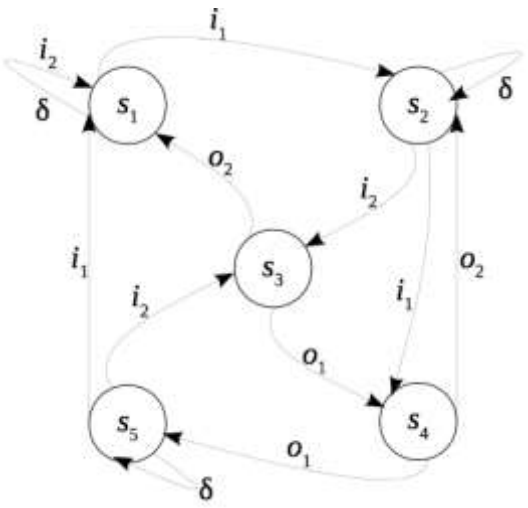

Fig. 1 – An input/output automaton $\boldsymbol{S}$

As usual, synchronizing and homing experiments are used to identify the final (current) state of the machine under experiment, i.e., the state reached by the machine after an appropriate input sequence has been applied when the initial state of the machine was unknown. In this paper, we adapt the notion of a 'gedanken' experiment in the way that it can be used for the input/output automata for which corresponding synchronizing and homing sequences can be defined.

The experiment is performed under the following **hypothesis**:

We assume that before applying any input, a tester (or any experimenting entity) waits for a given maximal output timeout *t*. The experiment is performed as follows: the tester expects an output in *t* time units; if the machine produces one, then the timer is reset and the tester waits for another *t* time units. If no output is produced by the system in *t* time units then the tester applies the next input (if any) and resets the timer.

The latter explains the necessity of introducing the specific output $\delta \notin O$, namely whenever the output is not observed we assume that the system/machine produced the output δ. Such extension of the output alphabet allows to define the corresponding synchronizing and homing sequences for an Input/Output automaton.

As usual, a synchronizing sequence is an input trace such that after its application independent of the initial state, the current state of the machine is known. In other words, a sequence $\alpha = i_1 i_2 \ldots i_k$ is *synchronizing* for the automaton $\boldsymbol{S}$ if there exists a state $s \in S$ such that for each trace $\beta_1 i_1 \beta_2 i_2 \ldots \beta_k i_k \beta_{k+1}$ where p is the length of a longest sequence of consecutive outputs and $\beta_j \in (O \cup \{\delta\})^p$, $j = 1,\ldots, k+1$, it holds that the $\beta_1 i_1 \beta_2 i_2 \ldots \beta_k i_k \beta_{k+1}$-successor of the set S ($\beta_1 i_1 \beta_2 i_2 \ldots \beta_k i_k \beta_{k+1}$-state-after-$S$) is either empty or equals $\{s\}$. We note that hereafter the ɣ-successor of the state $s \in S$ is the set of states that can be reached from state s through the trace ɣ while the ɣ-successor of S has every state that is reached from some state of S through the trace ɣ.

A homing sequence allows to determine the final (current) state of the machine under experiment via the observation of its output response. Therefore, a sequence $\alpha = i_1 i_2 \ldots i_k$ is *homing* for the automaton S if for each trace $\beta_1 i_1 \beta_2 i_2 \ldots \beta_k i_k \beta_{k+1}$, $\beta_j \in (O \cup \{\delta\})^p$, $j = 1,\ldots, k+1$, it holds that the $\beta_1 i_1 \beta_2 i_2 \ldots \beta_k i_k \beta_{k+1}$-successor of the set S is either empty or is a singleton.

For an automaton $\boldsymbol{S}$ in Fig. 1 a homing sequence α is $\alpha = i_1 i_1$.

3. Deriving synchronizing and homing sequences for input/output automata

In this section, we discuss how homing and synchronizing sequences defined above can be derived against input/output automata. We also establish necessary and sufficient conditions for the existence of such sequences for the machines of the class/type described above.

3.1. Deriving synchronizing experiments

We propose to derive a synchronizing sequence for an automaton $\boldsymbol{S}$ where actions are divided into inputs and outputs via an iterative elimination of the transitions labeled by outputs. Such transition can always be omitted as for the automata class considered in this paper, there does not exist a state where transitions under inputs and outputs are defined at the same time. In other words, we propose to derive an automaton where only the transitions under inputs are left. Synchronizing sequences for such kind of automata are well studied [2, 5, 11, 13] and thus, classical methods for their derivation can be further applied.

Procedure 1

Input: Input/Output automaton $\boldsymbol{S} = (S, I, O, T_S)$

Output: Synchronizing sequence α or a message "The automaton $\boldsymbol{S}$ is not synchronizing"

Step 1. Derive an automaton $\boldsymbol{A} = (S_1, I, T_A)$ with the empty set of transitions, i.e. $T_A = \varnothing$.

Step 2. For each transition $(s, i, s'') \in T_S$, where $s, s'' \in S_1$, add to T_A the transition (s, i, s''); for each transition (s, i, s''), where $s \in S_1$ and $s'' \in S_2$, add to T_A the transition (s, i, s''') where state $s''' \in S_1$ and s''' is in a β-successor of s'' in the automaton $\boldsymbol{S}$, $\beta \in O^*$.

Step 3. Check the existence and derive, if possible, a synchronizing sequence α for the automaton $\boldsymbol{A}$:

If the sequence α is derived then **Return** α;

Else Return the message "The automaton $\boldsymbol{S}$ is not synchronizing"

□

Proposition 1. The automaton $\boldsymbol{S}$ is synchronizing if and only if the automaton $\boldsymbol{A}$ in Procedure 1 is synchronizing. Moreover, each synchronizing sequence α for $\boldsymbol{A}$ is a synchronizing sequence for $\boldsymbol{S}$.

□

As an example, check the existence of a synchronizing sequence for an automaton in Fig. 1. The corresponding automaton ***A*** derived over the inputs of ***S*** is presented in Fig. 2.

By direct inspection, one can assure that the automaton ***A*** in Fig. 2 is not synchronizing. Therefore, due to Proposition 1, the automaton ***S*** does not have a synchronizing sequence.

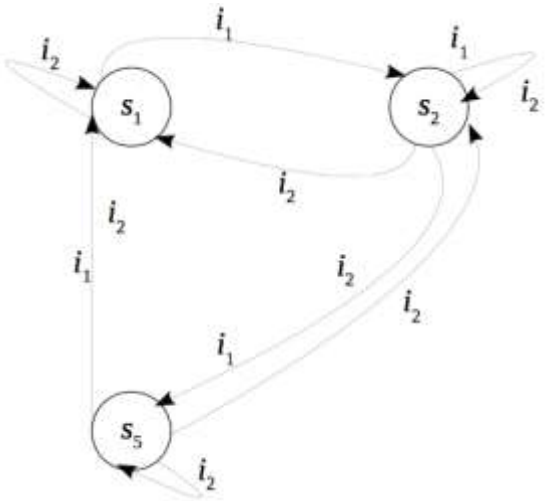

Fig. 2 – An automaton ***A*** derived through the application of Procedure 1

3.2. Deriving homing experiments

Similar to the derivation of synchronizing sequences we propose to reduce the problem of checking the existence and derivation of a homing sequence for an input/output automaton to that one for classical Finite State Machines (FSMs) as there are a number of such techniques for FSMs [see, for example 6-8].

Procedure 2

Input: Input/Output automaton ***S*** = (S, I, O, T_S)

Output: Homing sequence α or the message "The automaton ***S*** is not homing"

Step 1. Derive an FSM ***M*** = (S_1, I, $O \cup O^2 \cup \ldots \cup O^p \cup \{\delta\}$, T_M) with the empty set of transitions, i.e., $T_M = \varnothing$, where p is the length of a longest output trace of the automaton ***S***.

Step 2. For each state $s \in S_1$, such that $(s, i, s') \in T_S$, $s' \in S_1$, add to the T_M the transition (s, i, δ, s').

Step 3. For each state $s \in S_1$, such that $(s, i, s') \in T_S$, $s' \in S_2$, add to the T_M the transition (s, i, o_1 o_2 ... o_k, s''), $k \leq p$, where $s'' \in S_1$ is the o_1 o_2 ... o_k-successor of state s'.

Step 4. Check the existence and derive, if possible, a homing sequence α for the FSM ***M***:

If the sequence α is derived then **Return** α;

Else Return the message "The automaton ***S*** is not homing"

□

Proposition 2. A sequence α is homing for the automaton ***S*** if and only if α is a homing sequence for the FSM ***M***.

□

As an example, check the existence of a homing sequence of an automaton ***S*** in Fig. 1. This automaton does not have a synchronizing sequence, nevertheless, a homing one can still exist. In order to check the existence of a homing sequence we derive an FSM ***M*** of Procedure 2 for the automaton ***S*** in Fig. 1. The obtained FSM ***M*** is shown in Fig. 3.

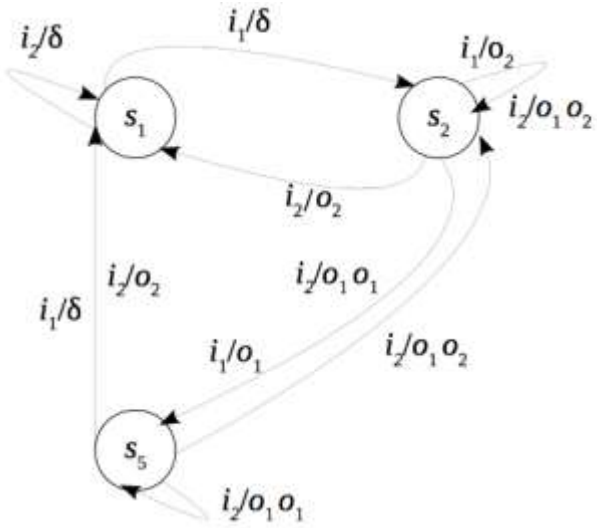

Fig. 3 – An FSM ***M*** derived from ***S*** using Procedure 2

The application of classical methods [6, 7, 11] for the homing sequence derivation for the FSM ***M*** can return a sequence $\alpha = i_1 i_1$. Therefore, the sequence $\alpha = i_1 i_1$ is a homing sequence for the automaton ***S*** in Fig. 1.

We note that the computational complexity of the existence check is the same (or at least not better than) as for corresponding automata and FSMs. The reason is that when deriving a synchronizing sequence an automaton that does not have output actions can be considered while for deriving a homing sequence a classical FSM can be represented as an Input/Output automaton by

presenting each FSM transition *i*/*o* as a pair of consecutive transitions under *i* and *o*. The latter means that for input/output automata that lead to the derivation of synchronizing/homing sequences for (partial) nondeterministic automata/FSMs the length of such sequences is exponential with respect to the number of states of the machine. Therefore, one of interesting issues for the future work is the development of the approaches for decreasing this complexity or to specify classes of Input/Output automata where the complexity can be reduced.

4. Conclusion

In this paper, we have studied the problem of deriving homing/synchronizing sequences for Input/Output automata. We have limited the class of such automata with automata where at each state, only inputs or only outputs are defined. We have shown how for such automata, the known techniques for deriving homing/synchronizing sequences for automata and FSMs can be adapted. To the best of our knowledge there are no papers where homing/synchronizing sequences are derived for Input/Output automata. We also mention that a number of examples of using Input/Output automata for describing the behavior of (components) of discrete event systems can be found in [1, 12].

As for the future work, we are mostly concerned how to extend the obtained results to adaptive sequences including distinguishing sequences as well as how to expand the class of Input/Output automata for such ‘gedanken’ experiments.

An interesting question is about Input/Output automata with a nonobservable action τ. In this case, the nondeterminism degree of the automaton will be increased and more assumptions on the automaton behavior should be made when deriving homing/synchronizing sequences.

Another question is about considering Input/Output automata where both inputs and outputs are specified at some state. It is quite possible that in this case, more assumptions have to be made about the implementation/execution of a ‘gedanken’ experiment and we are going to try our hand in establishing such minimum assumptions.

As most problems of checking the existence and derivation of synchronizing and homing sequences for partial and non-deterministic automata and FSMs are PSPACE-complete, there is an interesting question about defining Input/Output automata classes where the complexity can be decreased. One possible way to decrease this complexity can be to consider adaptive synchronizing and homing experiments instead of preset, however this issue needs more investigation.

Based on the experience of constructing adaptive state identification experiments for FSMs, we suppose that in this case, we will face the same problems as discussed above and it is our first

priority to define and construct adaptive homing/synchronizing experiments at least for the class of Input/Output automata described in this paper.

The problems listed above, as well as many others, form the directions for the future work.

References

1. Бурдонов И.Б. Теория конформности для функционального тестирования программных систем на основе формальных моделей : дис. … док. физ.-мат. наук. Москва, 2008. 596 с.
2. Мартюгин П.В. Нижние оценки длины кратчайших бережно синхронизирующих слов для двух- и трёхбуквенных частичных автоматов // Дискретн. анализ и исслед. опер. 2008. №4. Т. 15. С. 44-56.
3. Gill A. State-identification experiments in finite automata // Information and Control. 1961. P. 132-154.
4. Hierons R. M., Jourdan G.-V., Ural H., Yenigun H. Using adaptive distinguishing sequences in checking sequence constructions // ACM symposium on Applied computing : proceedings. 2008. P. 682-687.
5. Ito M., Shikishima-Tsuji K. Some Results on Directable Automata // Theory Is Forever. LNCS. 2004. №3113. P. 125-133.
6. Kohavi Z. Switching and Finite Automata Theory. McGraw-Hill: New York, 1978.
7. Kushik N., El-Fakih K., Yevtushenko N., Cavalli A. R. On Adaptive Experiments for Nondeterministic Finite State Machines // Software Tools for Technology Transfer. 2016. 18 (3). P. 251-264.
8. Kushik N., López J., Cavalli A.R., Yevtushenko N. Improving Protocol Passive Testing through "Gedanken" Experiments with Finite State Machines // QRS : proceedings. 2016. P. 315-322.
9. Lee D., Yannakakis M. Testing finite-state machines: state identification and verification // IEEE Trans. on Computers. 1994. 43(3). P. 306-320.
10. Moore E.F. Gedanken-experiments on sequential machines // In Automata Studies (Annals of Mathematical Studies no.1). Princeton University Press. 1956. P. 129-153.
11. Sandberg S. Homing and Synchronization Sequences // Model Based Testing of Reactive Systems. LNCS. 2005. №3472. P. 5-33.
12. Tretmans J. Test Generation with Inputs, Outputs and Repetitive Quiescence // Software - Concepts and Tools. 1996. 17 (3). P. 103-120.
13. Volkov M. Synchronizing Automata and the Černý Conjecture // 2nd Int'l. Conf. Language and Automata Theory and Applications : proceedings. 2008. P. 11-27.

UDC 004.451,004.415.5

Verification of Operating System Components

Alexey V. Khoroshilov (Institute for System Programming of RAS)

Victor V. Kuliamin (Institute for System Programming of RAS)

Alexander K. Petrenko (Institute for System Programming of RAS)

The paper concerns recent advances in reaching the goal of industrial operating system (OS) verification. By industrial OS we mean a system actively used in some industrial domain, elaborated and maintained for a significant time, not a proof-of-concept OS developed with mostly research intentions. We consider decomposition of this goal into tasks related with various functional components of OS and various properties under verification, and application of different verification methods to those tasks. This is a trial to explicate and summarize the experience of several projects on various OS components and different OS features verification conducted in ISP RAS.

Keywords: Operating system, verification, testing, monitoring, static analysis, deductive verification

1. Introduction

Modern industrial operating systems, which are used for plenty of real-life applications, are rather complex. They are not just very large pieces of code, they also have a great number of heterogeneous features, should operate on a large variety of hardware from diverse manufacturers, and are to provide for application developers numerous interfaces, which are expected not only to work correctly, but also to use underlying hardware in effective, efficient, and fault-tolerant way. By industrial operating system (OS) we mean in this article an OS actively used in some industrial domain (or a general purpose one), elaborated and maintained for a significant time. We do not discuss here OSes developed with certain research purposes or as a proof-of-concept, they may be much more simple than industrial ones and have different specifics.

An OS is supposed to perform two main tasks.

- It should organize operation of multiple applications on some machine, managing hardware resources, and protect applications from interfering each other.
- It should provide interface for application developers to use those resources in a convenient way, and also to transfer data between applications, if needed.

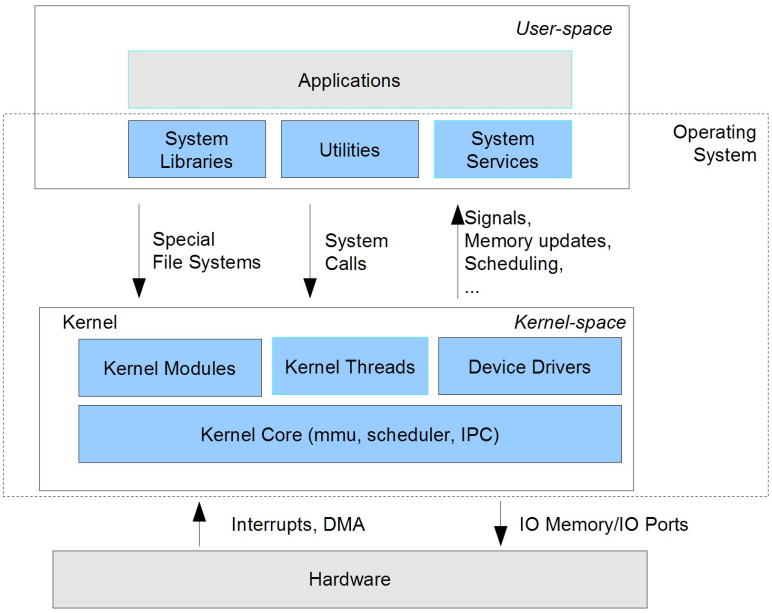

Fig. 1. Main parts of general purpose OS

The main part of an OS is *a kernel,* which works in privileged processor mode (kernel mode) and so has unbounded access to all system resources. Kernel manages application access to hardware resources, sets access policies, and prevents their violation. Some functions that do not require privileged mode are also sometimes included into kernel for efficiency.

Applications can interact with kernel mostly with *system calls,* which are calls of kernel functions with switch into privileged mode. There are additional ways to interact with kernel, like special file systems (procfs, sysfs, debugfs) in Linux. To provide convenient environment for application developers OS usually provides *system libraries and utilities,* implementing frequently used functions that require interaction with kernel. To solve tasks that need activity from the kernel side, *system services* are provided. Such tasks include communication protocols, managing special devices, etc. Corresponding services can work in kernel mode or in user mode. Figures 1 and 2 show the structure of general purpose OS and real-time OS correspondingly.

The above sketchy review of OS structure gives some hints on its complexity. The verification of industrial OS is also rather complex, especially if one takes into consideration the following.

- A plenty of features of modern OS provide various feature interaction cases and corner cases, where much more scrutinized inspection of required behavior is necessary.
- Multitasking support in modern OS makes checking behavior correctness much more intricate.
- Basic OS functionality must be available in spite of some faults in hardware or software

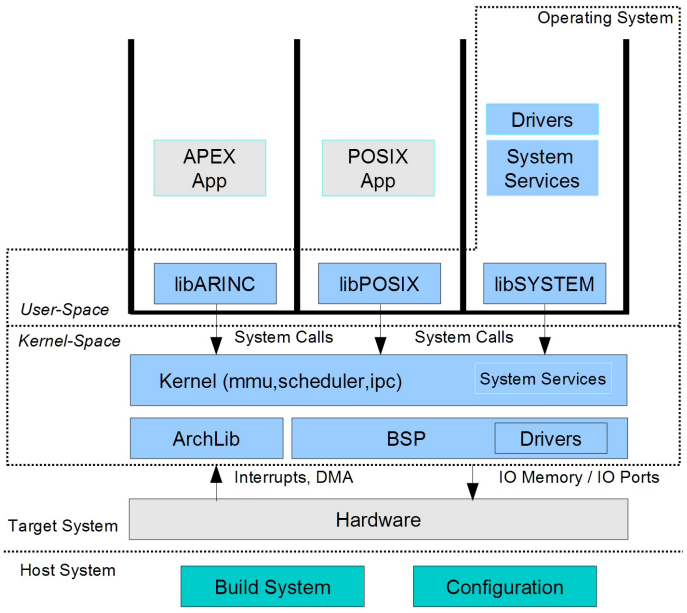

Fig. 2. Main parts of real time OS

components. This fault tolerance should also be verified.

- Modern OS usually supports network communications and provides a workplace for many users. It ensures certain security policies, setting restrictions on data and operations available for different users and processes. Those restrictions should be preserved in case of faults and attacks from malicious users or software components coming from network.
- Support of huge variety of heterogeneous hardware is usually implemented as deep configurability of an OS. So, one needs verification of OS behavior in various possible configurations, the total number of which is usually larger than astronomical numbers.
- The mere code size and number of functions in modern OS are great. The size of Linux kernel version 4.1 is reported [1] to be about 20.3 million lines of code (LOC), while drivers part, which is created and supported by diverse developers and is responsible for most of bugs is about 11.5 million LOC. The size of Windows XP is estimated as 45 million LOC [2]. The number of functions in system libraries of Debian 7.0 is about 720 thousand [3], while the number of system calls is about 350.

The numbers provided make the goal of thorough verification of industrial OS unreachable at this moment. Nevertheless, the developer community needs some methods to assure correctness, efficiency, security, and fault tolerance of modern OSes. The only reasonable way to help is to use various available verification methods to ensure those properties for some parts of OS code or functionality, or to ensure only a few critical properties on OS as a total. Important

results of partial verification are faults and errors found, so, although we cannot now guarantee strict correctness or security of an OS, we can come closer to them (if we do not forget about that ultimate goal in routine quality assurance processes). Thus, increasing scalability of used methods and tools step by step, we can enlarge the verified parts of code and functionality, targeting to reach the goal of complete verification somewhen in future.

In this paper we review the activity on verification of different parts and features of OSes performed in the Institute for System Programming of Russian Academy of Sciences in a dozen of projects conducted last 20 years. This activity uses and integrates different methods of verification for different OS properties and components, applying them to Linux OS and several specialized real-time OSes. The main methods used are as follows.

- **Testing and dynamic analysis.** Testing can be performed in different ways. The most lightweight testing focuses on producing test cases for basic behavior of functions, skipping consideration of complex or even non-so-often cases. The main target test completeness criterion for such testing is coverage of functions. The most thorough testing is intended to provide as strict and accurate checks as possible. It uses formal specification of required behavior, tries to formulate presumptions and strict guarantees concerning the correctness of tested system, and is targeted on coverage of all conditions met both in code and requirements, along with some corner cases (buffers overflow, failures of underlying hardware, processing simultaneous events, etc.). Various testing methods between these two extremes are also used. The main properties under test are functional, but testing is also the main method to check efficiency and fault tolerance. Along with testing other dynamic analysis methods, not requiring preparation of test suites, can be used.
- **Static analysis.** Static analysis also presents a wide range of approaches, from simple, quick, and lightweight checkers seeking a bounded number of bug patterns and producing a lot of false positives — reports on bugs, which actually aren't, to rather complex tools using formal specifications and configurable analyses, capable to catch very intricate bugs, requiring large effort during their configuration, usually with not-so-high numbers of false positives. Static analysis is widely used to check various code, but usually more complex and powerful techniques are applied to components with more strict requirements, like OS kernel modules.
- **Deductive verification.** Deductive verification is used to verify most important security or correctness properties. There are well-known examples of OS kernel verification [4–6],

but in all cases the verified code is much smaller than kernels of typical industrial OS. Nevertheless, deductive verification techniques can be applied to industrial OS code and provide valuable results.

Below we explicate and try to summarize the experience obtained by ISP RAS in dozens of projects, where some kind of verification was performed on various components of industrial OSes (Linux and several real-time OSes for specific domains). The paper organized as follows. In Section 2 we provide review of testing techniques used to check different OS components. Section 3 reports on application of various static analysis techniques, usually in conjunction with some dynamic analysis. Section 4 describes our results in deductive verification of OS security. Then the Conclusion sums up the exposition and describes possible further development.

2. Testing and dynamic analysis

ISP RAS develops OS components testing methods since its foundation in 1994. The first such results are related with KVEST [7], a method for test generation based on formal specifications of functional behavior in form of software contracts, used to construct several test suites for real-time OS developed and maintained by Nortel Networks.

2.1. Formal approaches

Later this approach was refined and extended into UniTESK method [8]. The basic ideas of the method are as follows.

- Requirements to library functions behavior are specified as *software contracts* — preconditions, postconditions, and data type invariants (they may be considered as common parts of pre- and postconditions of all functions dealing with those data types). Software contracts are written in extension of C language or with the help of specialized libraries in pure C/C++.
- Test completeness criteria are formulated as coverage of branches in postconditions. If there is a need to add some situations to coverage goals, they are formulated as specific additional branches, not related with behavior restrictions.
- Test scenarios are represented as extended finite state machines, for which execution of all reachable transitions guarantees coverage of all coverage goals (branches) specified in postconditions of functions called (each transition corresponds to a sequence of function calls). The control state of test scenario is a generalization of data structures used in

specifications of functions tested by this scenario.

- Testing is performed by automatic traversal of a state machine defined in test scenario. Each call to a function under test is augmented with call to the oracle function generated from postcondition and evaluating correctness of the results obtained.
- Testing of parallelism is based on interleaving semantics [9]. It is performed by gathering all the observed events (function calls, function returns, and others) and constructing a linear sequence of those events, in which all pre- and postconditions hold. If such a sequence cannot be constructed, a bug is recorded.

UniTESK was used for conformance test suite creation for Core part of Linux Standard Base (LSB), which describes system libraries and almost coincides with POSIX, in OLVER Project [10], where 1532 functions of LSB Core was formally specified and tested. The same method was applied in conformance test suite development for ARINC-653 part 1 standard [11] describing 54 functions.

Another test construction method, not using formal specifications, but based on formal investigation of requirements was used to create conformance tests for mathematical functions working with floating-point numbers in POSIX system libraries [12]. The method uses as test data specific floating point values, including numbers having patterns in mantissa (like 0000FFFFAAAA in hexadecimals), boundaries of domains of specific function behavior (such behaviors include monotonicity, sign preservation, well-known asymptotics), and so-called *worst cases,* numbers, for which correct function calculations requires much more precision than in average. For now test suites for 104 functions was developed.

2.2. Informal approaches

Several other methods used for test construction in ISP RAS are not based on formal specifications, but targeted on strict requirements traceability, so that tests are developed to check certain explicitly formulated requirements and they report on violation of this requirements (providing their ids) when find some bug.

The first method [13] is based on manual test case development with further parameterization making a test case a template. For test execution each template is supplemented with several arrays of arguments that are put in place of corresponding parameters. The method was used to create tests for more than 4000 functions in Linux system libraries, they detected about 40 bugs.

Another approach [14] provides automatic generation of sanity tests (checking only basic functionality) on the base of initialization procedures for data type values and libraries and preconditions of functions specified manually and stored in a database. This method provides rather surface testing, but can be used for massive test generation with little effort. It was applied to Linux libraries containing about 20000 functions.

2.3. Fault tolerance testing and dynamic analysis

For monitoring Linux kernel modules KEDR framework [15] was developed in ISP RAS. It makes possible to intercept calls from single kernel module, and so to observe its behavior in dynamics. On the base of KEDR the following verification techniques are implemented.

- KEDR Leak Check used to detect memory leaks in kernel modules. It is more convenient for leak detection then kmemleak [16] included in Linux distribution, but cannot be used to check kernel core code.
- Kernel Strider [17] used to detect data races, situations when several threads read and write one region of memory in unordered manner. Kernel Strider gathers information on module execution, which is then analyzed by ThreadSanitizer [18], data race detection tool developed by Google.
- KEDR Fault Simulation [19] used for fault tolerance testing. The testing organized in a following way. First, the module under test is executed in ordinary way and KEDR detects all calls to functions (system calls or calls of hardware-specific operations) that can fail, but very rare do this during real work. Second, for each call the test is executed, in which this call is simulated as failed. This approach helped to detect several bugs in mature file system drivers like ext4.

A specific example of monitoring used to detect data races is given by RaceHound tool [20], which implements the same idea as DataCollider [21]. It detects memory regions where a thread can write, sets hardware breakpoint on access to such regions, and inserts additional wait intervals around memory access operations in other threads in runtime. If this leads to an access to the tapped memory from another thread, a data race is reported.

3. Static analysis

To get more efficiency a large part of general purpose OS code is working in kernel mode, where it has many possibilities to damage important OS data structures. Since the code of

Linux drivers, which also works in kernel mode, is usually written by developers having good knowledge of hardware and not-so-good in rules of correct operation within Linux kernel, this naturally leads to the situation where more than a half of bugs detected in kernel is related with drivers code [22]. The similar relation is true for Windows OS [23].

To make development of kernel modules less error-prone, one needs specific tools that can check the rules of correct kernel application program interface (API) usage. Microsoft Research offers Static Driver Verifier tool [24] (called SLAM earlier) capable to solve this task for Windows. The similar solution is suggested by ISP RAS for Linux under the name of Linux Driver Verification (LDV) framework [25, 26]. The method used by LDV is the following.

- The rules of correct kernel API usage are specified as software contracts in specific notation extending C language. They are interpreted as aspect advices that should be inserted at the points where the specified API functions are called in the module under check. Being inserted in the module code, advice code creates error, if the rules specified are violated.
- The usage model is created for the module functions. This is important for driver modules, since their functions are not called explicitly. The usage model defines all possible sequences of function calls.
- The code of the module under check is processed by aspect weaver, which inserts rule checking code, and augmented by the usage model.
- The main check is performed by static verifier tool (most often BLAST [27] and CPAChecker [28] are used). The tool analyzes the code trying to solve reachability task — whether the error creation instruction can be reached in some execution. If it is reachable, then the corresponding execution scenario demonstrates a bug, incorrect use of kernel API, else the code uses the API functions correctly. Reachability task is solved with the help of counterexample guided abstraction refinement technique (CEGAR) [29], which constructs automatically more and more precise models of code execution, until the error-reaching path in model can be re-executed in real code, or becomes unreachable in the refined model.

LDV detects 5-8 bugs in almost each release of Linux kernel, for now the total number of found bugs is about 2500. It is used routinely to check about 4000 kernel modules.

Another example of static analysis usage is provided by a tool CPALocator [30] developed on the base of CPAChecker and used to search race conditions in OS code.

4. Deductive verification

Deductive verification is usually considered as the most strict and accurate verification technique, at the same time it requires a lot of effort and highly qualified staff to perform it in a productive way. A good review of deductive verification use for OS code is provided by [31].

In ISP RAS projects deductive verification was used to verify security properties of a Linux-based OS modified for specific use in government agencies [32, 33]. The OS is intended to implement a complex security model (called MROSL DP) integrating mechanisms of lattice-based mandatory access control, mandatory integrity control, and role-based access control. All the security mechanisms are implemented with the help of Linux Security Module (LSM) [34], which provides interceptor functions for all access operations in Linux.

First, MROSL DP model was formalized in Event-B and its main security properties (that no subject with less access level can get access to an object with higher confidentiality level; no subject can get access to an object, for which the subject has no a role having right to access to, etc.) were proved. Second, main LSM functions were also formally specified in so-called detailed model, for which the corresponding security properties were also proved. On the third step the contracts of LSM functions should be translated in ACSL, an extension of C language used in code verification framework Frama C/Jessie [35], and this framework should verify the behavior of C code on conformance with the contracts.

Althoug the project is not finished yet, a number of faults was found in the security model itself due to formalization, and several bugs were detected in code during its partial verification.

5. Conclusion

In this paper we provide a systemized review of verification activities used to check various components and features of industrial OS in ISP RAS projects. Although the ultimate goal — the thorough verification of an OS widely used in real-life — still remains unreachable, our experience shows that important advances in that direction were made by research and development community in last years.

The methods and tools developed for different purposes and using different basic approaches — testing, monitoring, static analysis, deductive verification — can enrich each other by borrowing specific modeling or reasoning technique, as it can be shown on example of memory modeling in static analysis and deductive verification tools [36].

One also can see during a last decade an impressive progress in verification techniques

applicable to real software. In the domain of OS verification such progress can be illustrated by a method for deductive verification of multithreaded C programs working with shared data proposed in ISP RAS [37]. We hope that in one-two years it will be implemented and we can see results of its experimental evaluation.

Another direction of future research concerns possibilities to reuse verification artefacts created by some methods in other ones [38].

References

1. Why is the Linux kernel 15+ million lines of code? https://unix.stackexchange.com/questions/223746/why-is-the-linux-kernel-15-million-lines-of-code/223770. Aug 2015.
2. How Many Lines of Code in Windows XP? https://www.facebook.com/windows/posts/155741344475532. Jan 2011.
3. Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. Testing of Operating Systems // Trudy ISP RAN/Proc. ISP RAS, 2014, 26(1):73-108 (in Russian).
4. Bevier W.R. Kit: a Study in Operating System Verification // IEEE Transactions on Software Engineering, 15(11):1382-1396, Nov 1989.
5. Alkassar E., Paul W.J., Starostin A., Tsyban A. Pervasive Verification of an OS Microkernel // In: Leavens G.T., O'Hearn P., Rajamani S.K. (eds) Verified Software: Theories, Tools, Experiments. VSTTE 2010. Springer, LNCS 6217:71-85.
6. Klein G., Andronick J., Elphinstone K., Murray T., Sewell T., Kolanski R., Heiser G. Comprehensive Formal Verification of an OS Microkernel // ACM Transactions on Computer Systems, ACM, 2014. 32(1), art. 2.
7. Burdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications // In: Wing J.M., Woodcock J., Davies J. (eds) FM'99 – Formal Methods. FM 1999. Springer, LNCS 1708:608-621.
8. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture // In: Eriksson L.H., Lindsay P.A. (eds) FME 2002:Formal Methods – Getting IT Right. FME 2002. Springer, LNCS, 2391:77-88.
9. Kuliamin V.V., Petrenko A.K., Pakoulin N.V., Kossatchev A.S., Bourdonov I.B. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems // In: Broy M., Zamulin A.V. (eds) Perspectives of System Informatics. PSI 2003. Springer, LNCS 2890:450-461.
10. Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., Rubanov V. Formal Methods in Industrial Software Standards Enforcement // In: Virbitskaite I., Voronkov A. (eds) Perspectives of Systems Informatics. PSI 2006. Springer, LNCS 4378:456-466.
11. Maksimov A. Requirements-based conformance testing of ARINC 653 real-time operating systems // Proc. of Data Systems In Aerospace (DASIA 2010), ESA SP-682.

12. Kuliamin V. Standardization and Testing of Mathematical Functions. // Proc. of Perspectives of System Informatics, PSI 2009. Springer, LNCS 5947:257-268.
13. Khoroshilov A., Rubanov V., Shatokhin E. Automated Formal Testing of C API Using T2C Framework // Proc. of International Symposium Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), pp.56-70.
14. Zybin R.S., Kuliamin V.V., Ponomarenko A.V., Rubanov V.V., Chernov E.S. Automation of broad sanity test generation // Programming and Computer Software, 34(6):351-363, 2008.
15. Shatokhin E. Using Dynamic Analysis To Hunt Down Problems in Kernel Modules // Presentation at LinuxCon Europe 2011, Czech Republic, Prague, 26-28 October 2011.
16. kmemleak description. https://www.kernel.org/doc/Documentation/kmemleak.txt.
17. Kernel Strider. https://code.google.com/p/kernel-strider/.
18. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice // In Proc. of Workshop on Binary Instrumentation and Applications (WBIA 2009). ACM, 2009, pp.62-71.
19. Tsyvaerv A., Khoroshilov A. Using Fault Injection for Testing Linux Kernel Components // Trudy ISP RAN/Proc. ISP RAS, 2015, 27(5):157-174 (in Russian).
20. Race Hound tool. http://forge.ispras.ru/projects/race-hound.
21. Erickson J., Musuvathi M., Burckhardt S., Olynyk K. Effective data-race detection for the kernel // Proc. of USENIX conference on Operating systems design and implementation, OSDI 2010, pp. 151-162.
22. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers // Trudy ISP RAN/Proc. ISP RAS, 2012, 22:349-374 (in Russian).
23. Ball T., Levin V., Rajamani S.K. A decade of software model checking with SLAM // Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011.
24. Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S.K., Ustuner A. Thorough static analysis of device drivers // Proc. of ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), pp. 73-85, 2006.
25. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Architectire of Linux Driver Verification // Trudy ISP RAN/Proc. ISP RAS, 2011, 20:163-187 (in Russian).
26. Zakharov I.S., Mandrykin M.U., Mutilin V.S., Novikov E.M., Petrenko A.K., Khoroshilov A.V. Configurable Toolset for Static Verification of Operating Systems Kernel Modules // Trudy ISP RAN/Proc. ISP RAS, 2014, 26(2):5-42 (in Russian).
27. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST: Applications to software engineering // International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007.
28. Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification // Proc. of International Conference on Computer Aided Verification (CAV 2011), Springer, LNCS 6806:184–190.
29. Clarke E., Grumberg O, Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement // In: Emerson E.A., Sistla A.P. (eds) Computer Aided Verification. CAV 2000. Springer, LNCS 1855:154-169.

30. Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Adjustable method with predicate abstraction for detection of race conditions in operating systems // Trudy ISP RAN/Proc. ISP RAS, 2016, 28(6):65-86 (in Russian).
31. Klein G. Operating system verification – An overview // Sadhana, 2009, 34(1):27-69.
32. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B // Proc. of Int. Conf. on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014), pp. 309–313.
33. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of specification decomposition methods in Event-B // Programming and Computer Software, 2016, 42(4):198-205.
34. Wright C., Cowan C., Morris J., Smalley S., Kroah-Hartman G. Linux Security Module Framework // In: Ottawa Linux Symposium, vol. 8032, 2002.
35. Marhé C., Moy Y. The Jessie Plugin for Deductive Verification in Frama-C // INRIA Saclay Île-de-France and LRI, CNRS UMR, 2012.
36. Mandrykin M.U., Mutilin V.S. Survey of memory modeling methods in static verification tools // Trudy ISP RAN/Proc. ISP RAS, 2017, 29(1):195-230 (in Russian).
37. Mandrykin M.U., Khoroshilov A.V. Towards deductive verification of C programs with shared data // Programming and Computer Software, 2016, 42(5):324-332.
38. Petrenko A.K., Kuliamin V.V., Khoroshilov A.V. Integration Points of Operating System Verification Techniques // Trudy ISP RAN/Proc. ISP RAS, 2015, 27(5):175-190 (in Russian).

УДК 004.94, 004.724.4

Security of Grid Structures with Cut-through Switching Nodes

Shmeleva T.R.

(A.S. Popov Odessa National Academy of Telecommunications)

Store-and-forward buffering of packets is traditionally used in modern network devices such as switches and routers. But sometimes it is a significant obstacle to the quality of service improvement because the minimal packet delivery time is limited by the multiplier of the number of intermediate nodes by the packet transmission time in the channel. The cut-through transmission of packets removes this limitation, because it uses only the head of packet, which contains the destination address, for the forwarding decision. Thus, the cut-through technology of packets transmission has considerable opportunities for the quality of service improving. Models for the computing grid with the cut-through forwarding have been developed in the form of colored Petri nets. The model is composed of packet switching nodes and generators of traffic; it can be supplied with malefactor models in the form of traffic guns disguised under regular multimedia traffic. The present work is the further development of methods of the rectangular communication grids analysis for nodes performing the cut-through switching. The methods are intended for application in the design of computing grids, in the development of new telecommunications devices, and in intelligent defense systems. Preliminary estimations show that the cut-through technology inherits some of the negative effects, which are associated with the traditional store-and-forward delivery of packets. A series of simulations revealed conditions of blocking a grid with its regular traffic. The results are applicable in the intellectual detection of intrusions and counter-measures planning.

Keywords: *computing grid security, cut-through switching, traffic attack defence, performance evaluation, colored Petri net, deadlock.*

1. Introduction

Intelligent Defense/Security Systems considerably relay on trustworthy models of networks and intrusion (malefactor). Colored Petri nets are prospective formalism for intellectual systems, because they allow simulating neural networks [2] and other facilities of knowledge representation. At the initial stage of research, models of underlying grid and intrusion of a specific (disgusted)

form are developed [5] where an overview of the related work on the grid security aspects has been presented.

Store-and-forward (SAF) buffering of packets is traditionally used in modern network devices such as switches and routers. But sometimes SAF is a significant obstruction to the quality of service (QoS) improvement. Minimum time of the packet delivery for SAF is limited by the product of the number of intermediate nodes to the packet transmission time in the channel. The cut-through transmission of packets [3] removes this limitation, because it uses only the head of packet, which contains the destination address, for the forwarding decision. Thus, the cut-through technology of packets transmission has considerable opportunities for QoS improving.

However, preliminary estimations suggest that the cut-through technology can inherit some of the negative effects, which are associated with the traditional store-and-forward forwarding of packets.

The present work is the further development of methods for analyzing of the rectangular communication grid model, which nodes perform the cut-through switching. The methods are intended for application in the design process of computing grids [4], in the development of new telecommunications devices, and in intelligent defense systems. In [5, 8] the blocking of computing grids was studied. The prospects of grid models application lay in control tools and intelligent network security. The model is developed using a colored Petri nets (CPN) and modeling system CPN Tools [1]. CPN is a graphical oriented language for design, specification, simulation and verification of systems. This language is particularly well-suited to illustrate and simulate systems in which communication and synchronization between components and resource sharing are primary concerns. Telecommunication networks and different network technologies were modeled and investigated via CPN [6, 9].

2. The application of cuts-through packet switching

Two main methods of packet switching dominate in modern telecommunication systems [3]: the first is with the compulsory buffering of the packet or store-and-forward (SAF), and the second is without buffering or cut-through, another popular name is “on the fly”. Hybrid switches are also applied in networks; they can be automatically reversed from the cut-through mode to the SAF mode and vice versa. Switching between the modes is based on the determination of performance and the integrity of the package. Most of the modern switches support concurrently different packet rates.

The SAF technology is traditional for most networks. It provides the packet transmission to the sender only after receiving of the packet and the check the control sum (CRC). The packet is

deleted if it shorter than 64 bytes or longer than 1518 bytes or the control sum is invalid. For the SAF method, the packet delivery time increases in proportion to the size of the packet.

The switching technology “on the fly” buffers the packet head only. The cut-through switches do not produce the packets selection; therefore they are the fastest in its class. The disadvantage of this switching is that it transmits any packets including with incorrect control sum. In some cut-through switches, ICS (interim cut-through switching – intermediate switching on the fly) method is used, which filters packets with a length less than 64 bytes. The cut-through switches [3] are primarily used in data centers, where it is necessary to ensure the continuous transmission of a large traffic value with minimal delays.

3. Model of grid structures with cut-through switching nodes

In telecommunication networks, one of the basic components is the active equipment such as switches or routers. Models of communication rectangular grids [4] with the basic element represented by the switch model with SAF method are studied in [5, 8]. Let us consider the construction of the node model with a direct transmission of a packet from port to port or the cut-through switching.

The used color sets, functions, variables and value are described in [5]. For construction of grid model we use two main models: node model with cut-through switching, as a communicational device, and model of a traffic generator, as a terminal device. All models were constructed in CPN Tools.

3.1. Node model with cut-through switching

The node model is based on the standard packing switching procedures [3] of the modern networks and grids which provide the model relevance. The model of the node with the cut-through switching is shown in Fig. 1. It is a model of network device for composition of the rectangular grid model. There are four ports in the node model which provide the full-duplex mode of work in two-directional mode for transmitting and receiving packets simultaneously. Each port consists of four places: output port buffer *po* and its capacity limit place *pol*, input port buffer *pi* and its capacity limit place *pil*. For specification of all ports, an index of port is added to the port name. The node ports’ places are situated on the sides of a square for a future composition of the grid: the upper port is the first port with places *po1, pol1, pi1, pil1*; the right port is the second port with places *po2, pol2, pi2, pil2*; the bottom port is the third port with places *pi3, pil3, po3, pol3*; the left port is the fourth port with places *pi4, pil4, po4, pol4*. According to the cut-through switching method, there is

no buffer in the model. The ports places have a color set *pkt*, the limit places have a color set *cc*, and they are contact places for the grid composition.

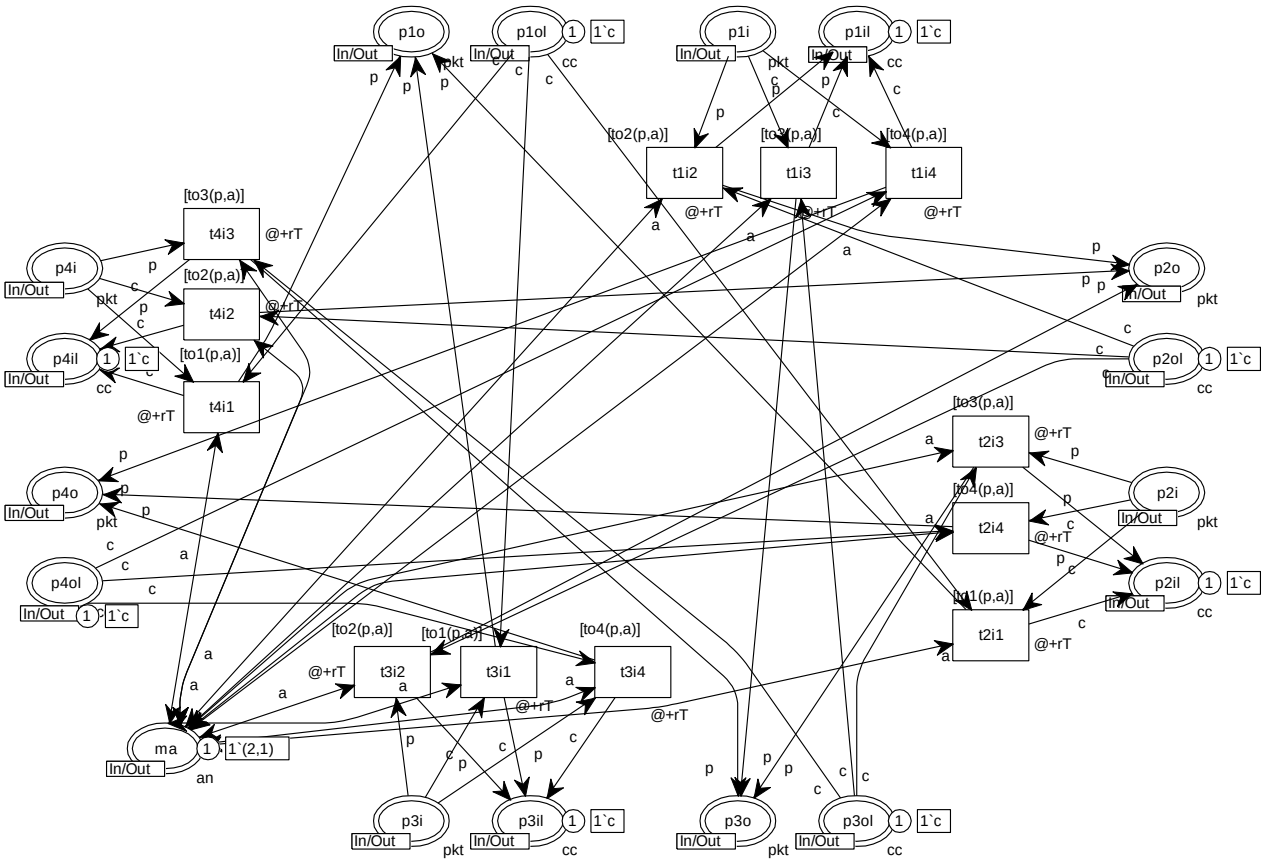

Fig. 1. Model of a communication node.

The system of nodes' addressing uses two integer numbers *(i, j)*, where the first number is a number of row and the second is a number of column in the grid. Contact place *na* contains address of the node.

The output channel of a port is modeled by the two places: *po* and *pol*. The input channel of a port is modeled by the two places *pi*, *pil* and three transitions for each possible direction of transmission (upper, bottom, left or right). The name of transition is *ti*; for describing the redirection from the input to output port, two indexes are added. For example, transition *ti34* transmits a packet from the input port *pi3* to the output port *po4*.

Each transition has a guard function for the packet redirecting and two timed parameters *rT*, the receiving delay time of a packet, and *chT*, the transmitting delay time of a packet.

According to the cut-through switching algorithm [3], a packet is redirected from an input port to an output port if the output port is free. In the node model, special predicates are used for definition

of the destination output port [5], they are represented as the guard functions of transitions. For instance, the predicate *to3(p,a)* defines output port number three for a packet forwarding, were *p* contains the information of the packet (destination address, sender address) and *a* is address of a current node. Modeling of static switching and routing tables is studied in [6, 9].

In the initial marking, all the limit places of ports *pil** and *pol** contain a token *1`c* which defines the port capacity; all input *pi** and output *po** places of ports are empty, there are no tokens in the corresponding places. Communication node model has a name according to the number of row and column in the grid, for example node *n*2-1 is a first element of second row in the grid structure.

3.2. Model of traffic generator

For investigation of QoS parameters of the grid structure the model of the traffic generator was constructed. This model consists of the following parts: receiving, sending and computing [5] submodels.

The sending part describes the process of traffic generation, the intensity and type of the traffic function distribution, rules of packet sending. Each packet consists of a destination address, a sender address, a string with some content and timed stamp of the sending time.

The receiving part of the model does not process an incoming packet; all packets are used in the computing part for QoS parameters calculation. The model of the computing part is shown in Fig. 2.

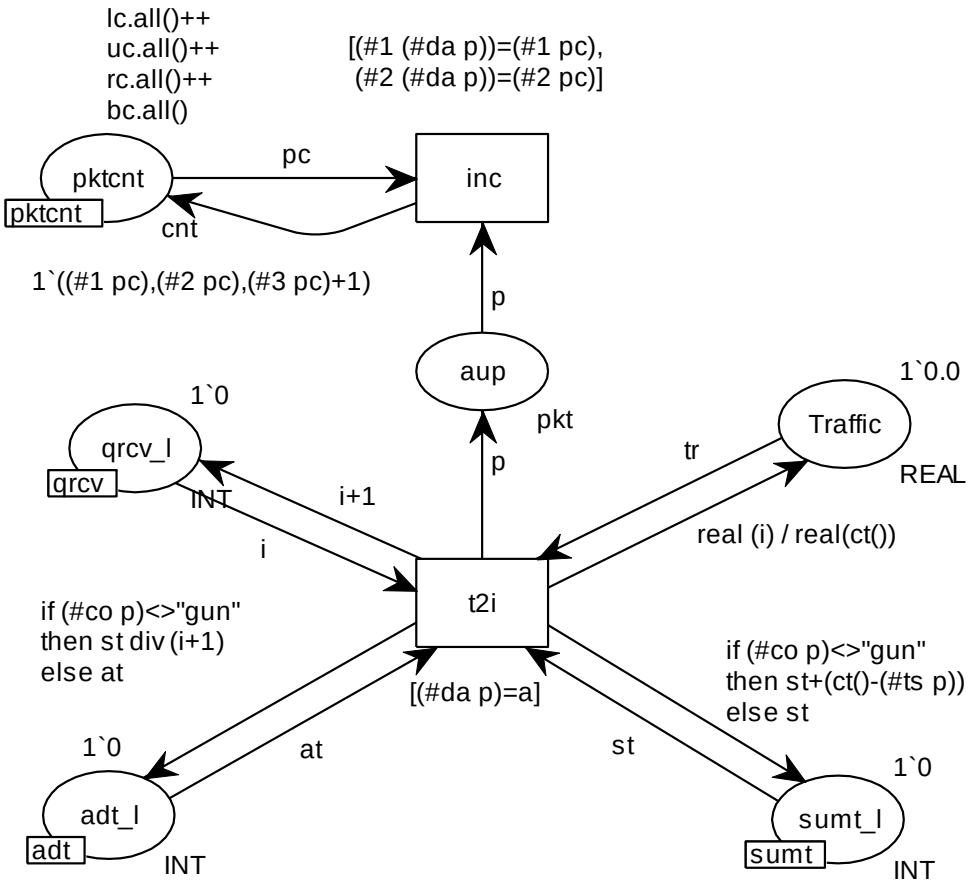

Fig. 2. Model of traffic generator: computing part.

Place *qrcv* contains a number of all received packets in the grid, place *pktcnt* contains a number of received packets for each terminal device, place *Traffic* describes a grid performance (packets/MTU), place *adt* is an average packet delivery time; abbreviation MTU denotes a model time unit applied for the time scalability. Terminal devices are named according to the first letter of border names, for example “right” border device has name *r*-indexes, *r*3-1 is a first right terminal device in third column of grid. Model construction for a measuring fragment (computing part of a model) was studied in [6, 7].

3.3. Model of grid structure

Model of grid structure is a composition of a communicational device models and a terminal device models [4, 8]. Device models are submodels and according to a hierarchical structure of CPN Tools [1], all submodels are represented as transitions; in our case they are supplied with address places situated in the main page of the grid model. Model of the grid structure with the cut-through switching is shown in Fig. 3. It is a model with current marking of the simulation process; there are three packets in the grid.

For example, the submodel of node with index (1,2) is represented on the main page as transition *n1-2* and address place *12a*. The submodel of bottom terminal device with index (3,2) is represented on the main page as transition *b3-2* and address place *32a*.

.

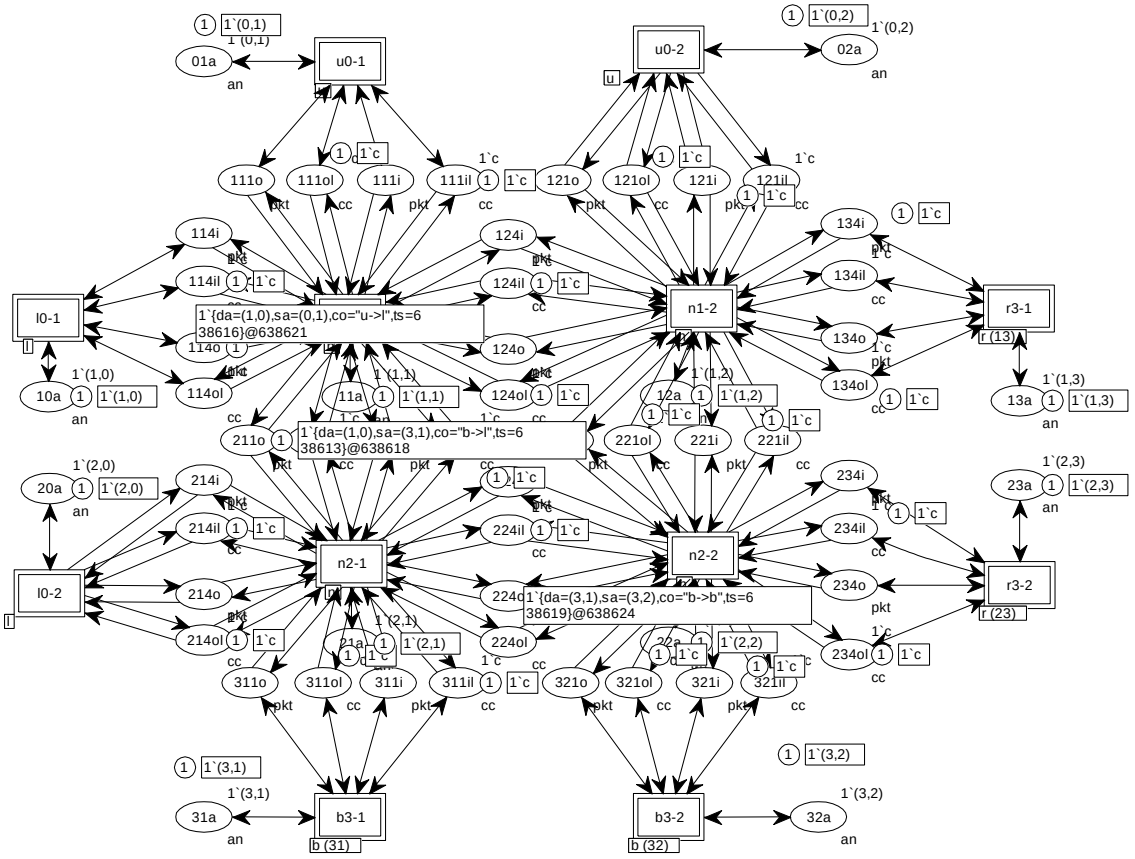

Fig. 3. Model of grid structure of size 2x2

Places with index (*) and suffix **o, *ol, *i, *il* are contact places, which describe output and input ports of the grid nodes and terminal devices. According to the composition rules [6, 8], all ports of communication devices have two description forms on the main page with respect to the enumeration in each of two connected nodes. The first and fourth ports of each node are represented with an index of this node; the second and third ports are union with the first and fourth ports of next nodes and have indexes of next nodes. For example, the place *221i* is the first input port of node *n2-2* and the third output port of node *n1-2*. The bottom row and right column contact places describe the first and fourth ports of nodes, which do not exist in the model. These places are used for connecting of bottom and right terminal devices. For example, the place *234oi* is the fourth output port of node *n2-3*, but there is no node with indexes (2,3) in the grid. This place is merged

with the input port of terminal device *r3-2*. In the model, there are no contact places with indexes of border devices.

4. Simulation of Grid Workload

For QoS parameters estimation, simulation of the grid workload was implemented. The grid workload is obtained using the traffic generations, described in the previous section, attached to the grid border. Intensity of the workload and timed delay *rT* of sending packets are basic parameters of the model, whose influence on the grid behavior was estimated. For Poisson distribution with different intensity, a grid performance and an average packet delivery time were studied. The obtained results were compared with characteristics of the grid model having SAF forwarding. Buffer size in this model is supposed equal 10 packets.

Table 1 shows the result of the grid investigation via regular workload for SAF and cut-through switching modes.

Table 1. Grid characteristics under workload

Workload intensity (*wl*)	Type of switching	Average packet delivery time (MTU)	Grid performance *gp* (packets/MTU)
50.0	cut-through*	10	0,14
50.0	SAF	21	0,14
30.0	cut-through*	11	0,23
30.0	SAF	21	0,23
16.0	cut-through*	11	0,44
16.0	SAF*	22	0,42

*Step=1000000, rT=5, bs=10, k1=2, k2=2; * – the grid comes to a full deadlock – no permitted transitions.*

Workloads with *50.0* and *30.0* intensities are light workloads for investigated grids. The grid performance is equal for two switching modes; the average packet delivery time for SAF mode is twice greater than for cut-through mode. Workloads with intensity about *16.0* are middle workloads for the investigated grids. The grid performance of cut-through mode is greater than for SAF mode, the average packet delivery time for SAF mode is twice greater than for cut-through mode. For big size grids, the average packet delivery time for SAF mode will be a few times greater than for cut-through mode.

Cut-through mode switching works faster than SAF mode, but it has the important disadvantage: network with cut-through mode switching is blocked if destination ports are busy. Ports are cleared after executing a special system time procedure (TTL). Some incoming and outgoing packets are lost. Ports clearing function is not simulated in this paper.

An example of a full deadlock is shown in Fig. 4, where inscriptions on the arcs indicate the number of blocked ports in nodes.

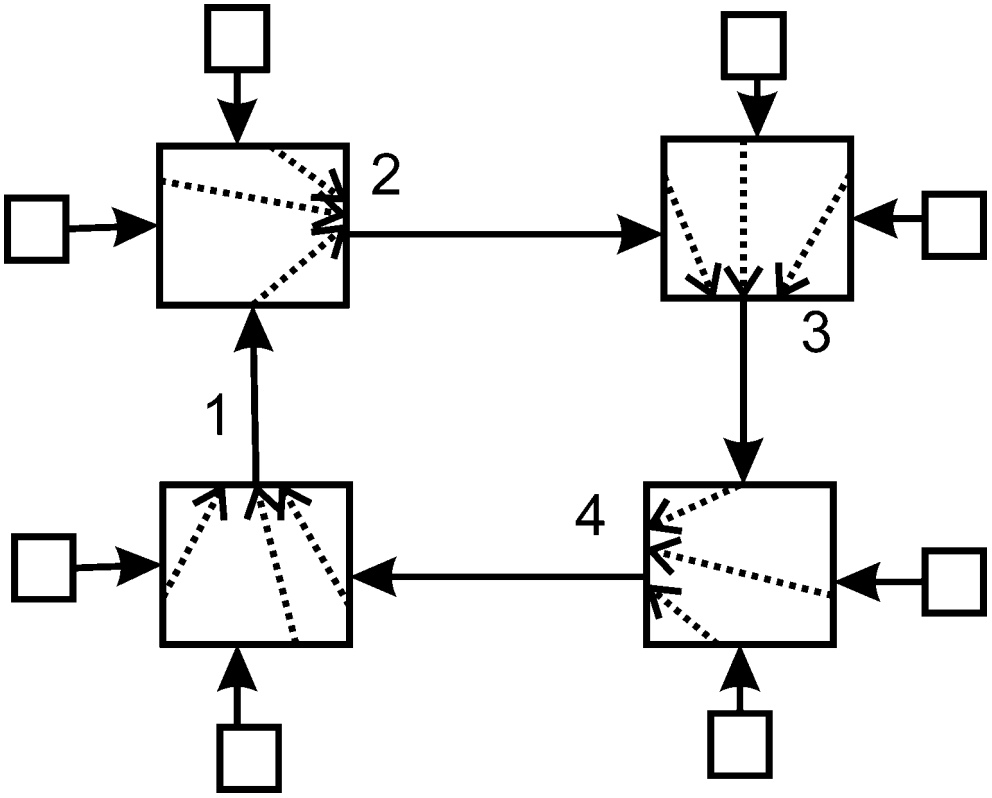

Fig. 4. An example of a full deadlock.

There are four packets in each node: one packet is in the output port, three packets are in the input ports and the destination ports of these packets are the same. The current node can not transmit the packet from output port to the next node, because the next node cannot redirect the incoming packet, because the destination port of this packet is busy. As a result of this clinch is the full deadlock of grid. Thorough explanations could be illustrated by a sequence of pictures from the first mutual blocking of a few nodes via extending the blocked areas to the complete deadlock shown in Fig. 4.

Grid behavior under traffic attacks and workload was studied for grid structures with store-and-forward mode [5, 8].

5. Conclusions

Models of grid structures with cut-through switching nodes were constructed in the colored Petri net form. Security of grid structures, in particular possibility of deadlocks, was investigated under workload in the environment of modeling system CPN Tools. The importance of obtained results for the grid computing domain consists in the conclusion that modern architecture of the switching devices does not guarantee the grid security. Special protocols which involve interoperability of a several nodes should be developed for the deadlocks detection and avoidance.

A future research direction will be to investigate the grid structures with cut-through switching nodes under a workload and traffic attacks; to study types of deadlocks and QoS characteristics of grid under disguised traffic attacks; to construct a re-enterable model [9] for investigation of grid structures with a big size, where initial characteristics of grid are model parameters.

The models are applicable in the intelligent defense and security systems of computing grids.

References

1. Jensen, K., Kristensen, L.M. Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, 2009, 384 p.
2. Koriem Samir M. CN-Nets for Modeling and Analyzing Neural Networks. J. King Saud Univ., Vol. 13, 2001, Comp. & Info. Sci., pp. 19-47.
3. Liberzon Daniel. Switching in Systems and Control. Published by Birkhäuser Boston, 2003, 230 p.
4. Preve, N.P. (Ed.). Grid Computing: Towards a Global Interconnected Infrastructure. Springer, 2011, 312 p.
5. Retschitzegger W., B. Pröll, D. A. Zaitsev , T. R. Shmeleva. Security of grid structures under disguised traffic attacks. Cluster Computing, 19(3) 2016, pp. 1183–1200.
6. Sakun A.L., Zaitsev D.A. An Evaluation of MPLS Efficacy using Colored Petri Net Models. Proceedings of International Middle Eastern Multiconference on Simulation and Modelling (MESM'2008), Amman (Jordan), August 26-28, 2008, pp. 31-36.
7. Shmeleva T.R., Zaitsev D.A. Switched Ethernet Response Time Evaluation via Colored Petri Net Model. Proccedings of International Middle Eastern Multiconference on Simulation and Modelling, August 28-30, 2006, Alexandria (Egypt), pp. 68-77.
8. Zaitsev D.A., Shmeleva T.R., Retschitzegger W. and Proll B. Blocking Communication Grid via Ill-Intentioned Traffic. 14th Middle Eastern Simulation & Modelling Multiconference, February 3-5, 2014, Muscat, Oman, pp. 63-71.
9. Zaitsev D.A., Shmeleva T.R. Parametric Petri Net Model for Ethernet Performance and Qos Evaluation. Proceedings of 16th Workshop on Algorithms and Tools for Petri Nets, September 25-26, 2009, University of Karlsruhe, Germany, pp. 15-28.

UDC 004.05

Design and implementation a software for water purification with using automata approach and specification based analysis

Sergey Staroletov (Polzunov Altai State Technical University)

The paper covers design and developing software for hardware plant for water purification, the architecture for it, received automaton diagrams of water preparing and normalization based on customer specifications and requirements. Discussing the components of the system, layers of abstractions, verification points, issues to build it. The way of developing well-qualified suchlike systems based on specifications is given.

Keywords: Water purification, software, automaton, energy conservation, verification, requirements engineering

1. Purpose and novelty of the project

Our university received an order for the research work to design and development software under the UN grant for developing countries in the field of energy conservation.

A customer is developing the hardware stand providing preparation of distilled water of the given temperature and testing the energy consumption and water consumption of various connected devices (for example, washing machines and dishwashers).

Water purification is necessary for this project because existing standards [1] for measuring energy consumption and water consumption presuppose to work with distilled water (water preparation process) at a given temperature (water normalization process).

It was necessary to design and implement software that automatically control preparing large volumes of purified water, also measures the energy and water consumption of the connected devices and generates some reports.

The novelty of the project is following: all water preparation and equipment testing should be performed without user intervention by the automatic operations in the hardware stand and the operator's job is only to select the mode, to set parameters and then run the process.

From the programmatic point of view, novelty consists primarily in mandatory to implement algorithms for water purification and control of hardware devices to it, these algorithms must be primarily reliable because water is supplied under high pressure, is heated using amperage

of tens of amperes, and all possible exceptional operations must be processed.

These algorithms must include various multithreaded interactions because the states of the devices are not correlating to each other, and it is necessary to examine at any time the devices, update the interface, make decisions about the further operational logic.

Therefore, it was necessary to design the software architecture properly and verify the proposed algorithms before using them.

This paper covers the software architecture design and algorithms developed by the author, also verification and testing, main issues and the questions about formulation the requirements for building these systems.

The results partially were obtained within the RFBR grant (project No. 17-07-01600).

2. A bit about the hardware

In this article, there is no purpose to describe the hardware. The author did not design and develop it but implemented software for the available hardware. Here will be given information about the components to get an idea of the operation of the equipment as a whole.

Here is a water purification hardware stand (Figure 1):

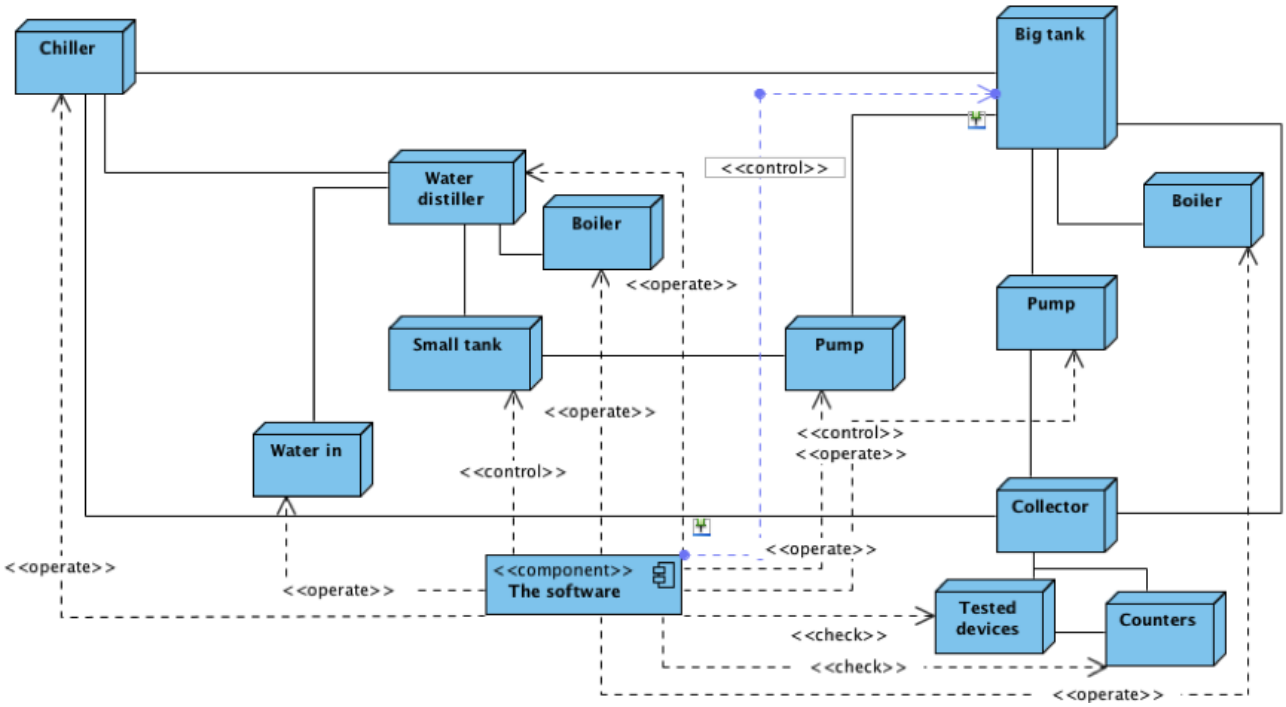

Fig. 1. Diagram of hardware components using in the stand

1. A chiller (conditional element) for cooling the water. Chiller is operated by a special software to set the output temperature based on current air temperature and desired temperature.

2. An aqua-distiller (with heating element) to create a distillate from given water.

3. A small tank for obtaining the current distillate with a level sensor.

4. Large storage tank for obtaining final water with level sensors and a heating element for heating.

5. A small pump for pumping water from a small tank into a large storage tank.

6. Main pump for water circulation in the whole system (a variable frequency drive).

7. Digital equipment for measuring water consumption (water flow meters) and electricity meters.

8. Digital devices with digital and analog I/O ports to connect devices, start relays and sensors to serial port adapters.

9. A collector, through which the devices checked for energy saving are connected and through which the prepared water circulates from the tank.

10. Various valves for switching water flows between the aqua-distiller, the storage tank, the collector.

The interaction with the hardware is implemented on the basis of the Modbus [2] protocol by using digital multi I/O devices, on/off switching relays, obtaining status data via analog and digital inputs. Linear interpolation algorithms are used to translate analog sensor values into the digital form.

The special software component called Devices Map has been implemented to link the devices from devices list (Figure 1) into concrete I/O ports on concrete digital I/O modular devices connected to concrete serial ports on the computer.

3. System architecture

The designed application inside consists of:

- **A device(hardware) layer** - a single software interface for the devices is created and implementation classes for each device in the system are created. This layer is built on the top of the Modbus protocol (and GOST IEC protocol [3] for the electric meters) and own actions for every device (to read some values from device's registers).
- **Serial port layer**. Every device is known by its port (Device Map tool causes a hash table device - port) and for the correct work, it must reserve the port because at any time only one device can operate on the port. Each port layer abstraction should work in **Device Getter Thread**, and to acquire the port to transmit data we must wait for

any other device freeing the port. It could be implemented as a synchronization primitive (mutex with lock/unlock).

- **A layer of objects**. Every device state is represented as an object. If we want to acquire a device state we can get the state in every moment by accessing a getter method in a corresponding object which stores the last received state from a device. Object stores current data and periodically **Device Getter Thread** updates it.
- **Updater Thread** is a foreground process that periodically asks all the objects and gets their state, then updates corresponding UI elements.
- **UI** is a mnemonic representation of the system; it has background images and dynamic elements (text values of system parameters such as temperature, pressure, on/off boxes to show states, graphic objects to draw some primitives like level meter, animations to show chiller's fans) which are updated by Updater Thread. UI also can work in manual operator's mode when operator press to the devices on the mnemonic diagram to start and stop them.

An example of components inter-operation sequence is shown on Figure 2.

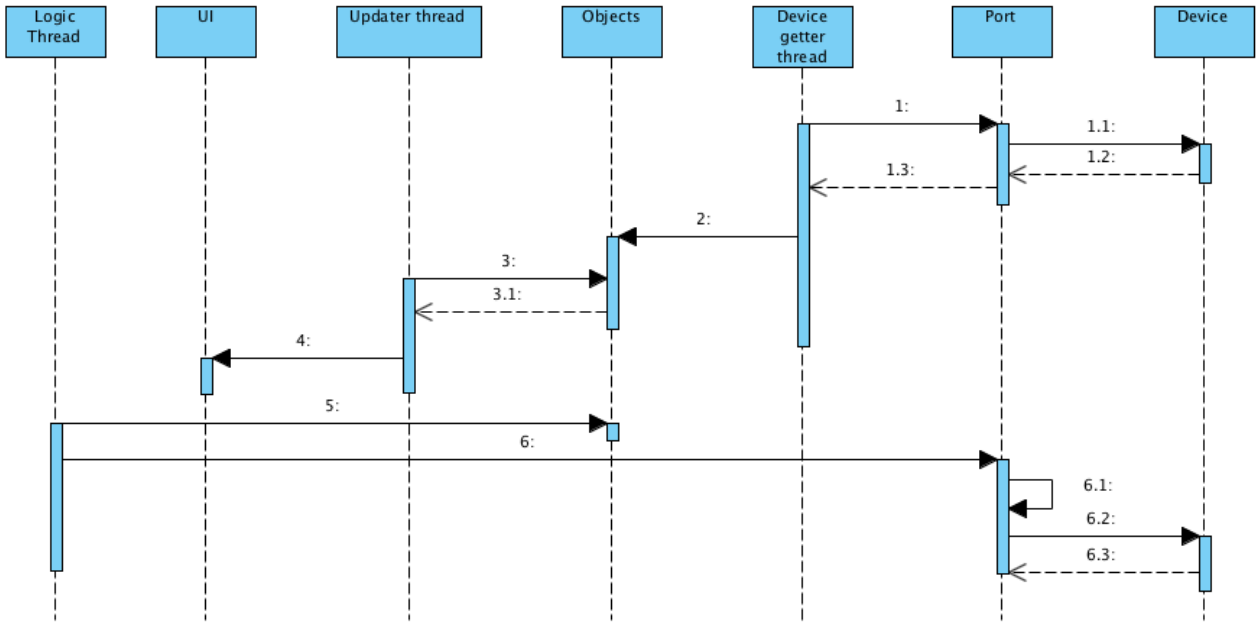

Fig. 2. Sequence diagram of possible components inter-operation

Here, **Device Getter Thread** starts to update the device's state. It requests a **Port** for given device, possible wait for the port, then when the port is free it runs device state update routine (which constructs Modbus or different protocol message corresponding to the device number and device updating algorithm), then gets the state and updates the **Object** to its

device. Then **Updater Thread** gets the current state of the device object and updates the **UI** corresponding to the new device state value.

Described in the next sections **Logic Thread** can get the object state and make decisions based on it. During the logic cycle, it can turn on/off a special device (for example, starts/stops the main pump). It asks port when it becomes free for the corresponding device, then acquires it and executes a routine in the **Device layer**.

Implementing these ideas is not very difficult to a developer who knows how to operate with the threads and how to lock resources from data races with using simple locking. Implementing the device layer requires reading its specifications, understanding the protocols by using original software and looking to transmitting and receiving data. The correctness of it is checking by testing techniques.

4. Requirements for development

Requirements engineering [4] is a way to collect customer's requirements for the system. Our goal now to help software engineers work together with the customers to build adequate error free software because the definition of the word "error" is nonconformity of program's behaviour to the original specification of requirements. Why not use this project to analyse such documents like "Requirements for development" to extract the correct specifications from it and integrate the requirements engineering, developing and verification processes to minimize troubles of producing such systems?

About the water purifying software. The software engineer can create the software to start/stop the devices and watch the states based on devices specifications but cannot create the whole inter-operation system because he usually doesn't know anything about purifying the water of even what is the chiller or aqua-distiller, he needs a customer and his needs. In this project, firstly, the task of implementation of given system architecture has been done, so that the operator could run the whole process of water purifying manually by pressing keys in the application and visually monitor the result on the screen and a real installation. Further, as a result of negotiations, the algorithm of automatic actions of the program was approved in the form of the following document (some excerpts are given in Appendix A).

Here we see the description of the algorithm in some high-level form. The algorithm is based on the management of devices that were specified in the picture of the interface (like in Figure 1), and the functional is predetermined by water purifying specialist.

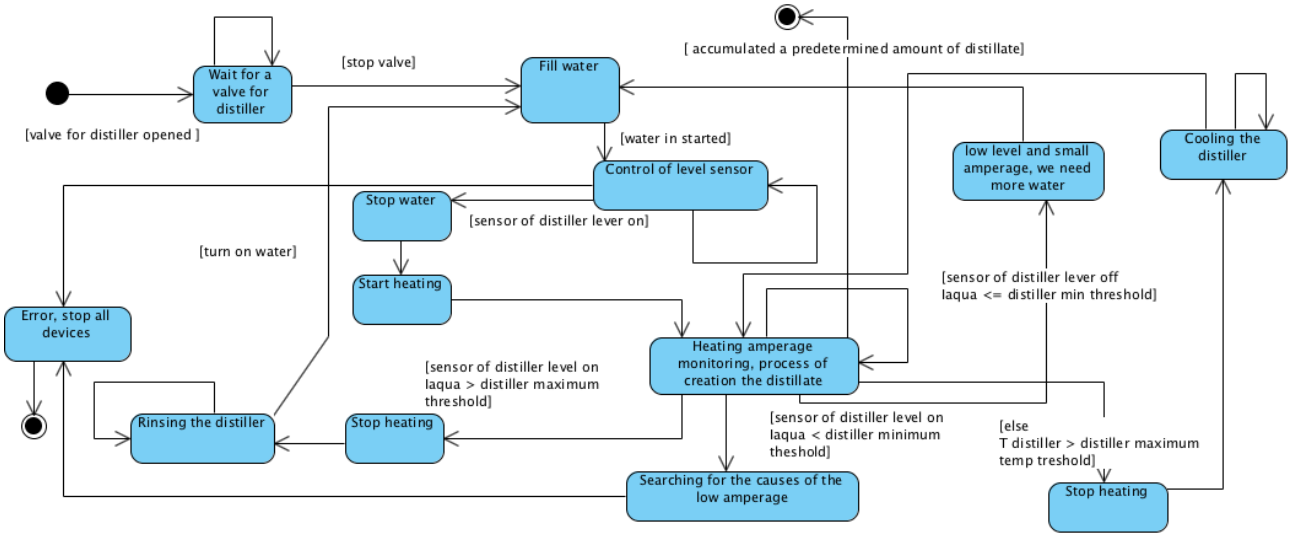

Fig. 3. State machine for water preparation

Next, based on given specification, it was decided to build the program in the form of a finite automaton, because it is clear how to implement a state-based algorithm as an automaton: each action corresponds to one or more states, and the actions of the automaton also correspond to actions as a result. Principal diagrams of logic are given below (Figure 3 and 4). The final step of programming is to implement the automatic purification algorithms. It is now not a difficult thing because the software architecture is designed than developed, device layer is developed and tested, and we have state machine diagrams as a model. Developing the algorithms for testing connected devices for energy and water consumption is not covered here because of simplicity (running water circulation between collector and storage tank; counting the meters, and plot the graphs).

5. Verification of the automata model

During this process we had to verify some verification points:

1. Actions to start/stop devices do not affect the current process of obtaining information.
2. Check pumping: if we got some level of water in the small tank all the water must be pumped to the storage tank.
3. Distillate preparing cycle will be completed, or an error message will be displayed.
4. Distillate normalization cycle will be completed, or an error message will be issued.
5. Will not fill water above the edge of distiller / tanks.
6. Pumps do not work without water.

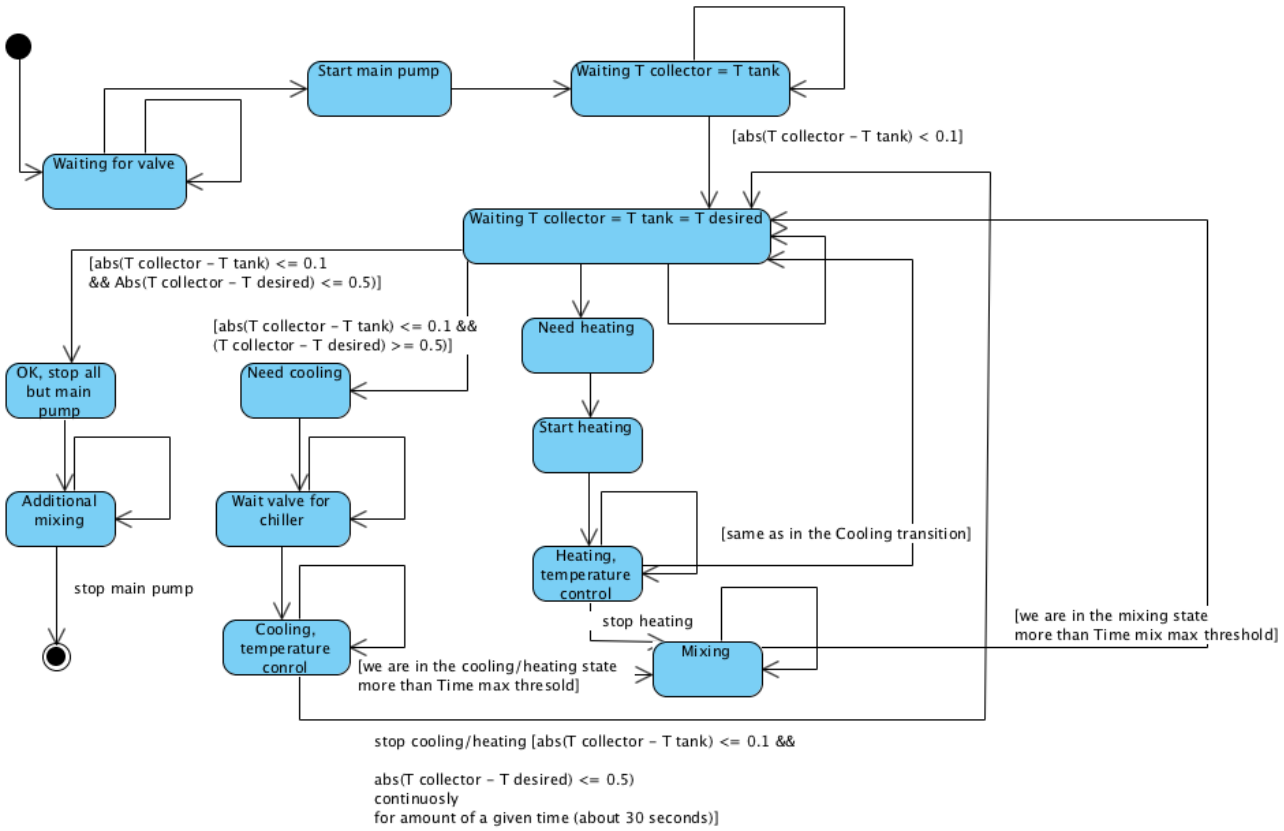

Fig. 4. State machine for water normalisation

7. You cannot start a heater / distiller if the level of water in the tank is small.

8. In each mode, water flows are redirected correctly.

9. Some devices can be turned on/off after some time after the control impact and the system must work correctly.

These points were given by the customer after implementing the system architecture and running it in manual mode by the operator and doing some experiments. Also, some of these checks were implemented in hardware PID control [5] methods.

To verify these key points with Model Checking approach, we need a checker which supports multiple processes and has abilities to implement a model of a finite automaton. Almost all checkers offer it. Moreover, also we need some predicates based on sensors values, automata states and system's states expressed in Linear-Time Logic because of delayed time in our verification points. We use Spin Model Checker [6] because of simplicity to achieve the desired result.

For example if we need to model the state changing process in the water normalization process and emulate the temperature mixing, we could write the following code in Promela

based on states from Figure 4, mtype state definition, loop and switch of possible states:

```
mtype = {StartMainPump,
WaitingTcollectorTtank,
WaitingTcollectorTtankTdesired,...};

mtype state = StartMainPump;

byte Tcollector = 10;
byte Ttank = 20;
bool mainPumpStarted = false;

active proctype WaterNormalProc() {
do
:: {
if
::(state == StartMainPump) -> {
mainPumpStarted = true;
state = WaitingTcollectorTtank;
}
::(state == WaitingTcollectorTtank) -> {
printf("WaitingTcollectorTtank");
if :: (Tcollector == Ttank) ->
{
state = WaitingTcollectorTtankTdesired;
};
:: (Tcollector != Ttank) ->
{
if ::mainPumpStarted -> Tcollector = Tcollector + 1; fi
}
:: (Tcollector != Ttank) ->
{
if ::mainPumpStarted -> Tcollector = Tcollector - 1; fi
```

```
}
:: (Tcollector != Ttank) ->
{
if ::mainPumpStarted ->Ttank = Ttank + 1; fi
}
:: (Tcollector != Ttank) ->
{
if ::mainPumpStarted -> Ttank = Ttank - 1; fi
}
fi
}
::state = WaitingTcollectorTtankTdesired -> {
...
}
fi
}
od
}
```

Here we use the non-determinism in Promela to achieve the random temperature mixing.

During the using the Model Checking approach we found some lacks in the transitions in the specification, it did not cover all the possible transitions to the final states.

6. Analysis, results and issues

As a result, the software has been developed successfully. We use the methods here ("→" means the sequence in the design and developing process): design of architecture → device level implementation and testing → overall internal architecture implementation → testing → specification of control algorithms → automata model → implementation of water purifying by given automata model → verification → overall system testing. It seems that the automata approach has been chosen extremely right: the given specification could be easily translated to automatons and each automaton could easily run from the thread in an infinite loop. Also, the proper architecture planning before construction of the logic, plays a huge role. Verification can help to refinement the specification and set-up critical hardware checks.

The main issues during project implementation were:

1. We cannot simulate the device layer, so we need to test on real devices.

2. The specification of control algorithms was given from the customer only after the overall internal architecture implementation because he is not sure about the functionality of the hardware so it was difficult to estimate the effort and the software cost before starting the project.

3. The specification of control algorithms was changed several times after testing the functionality because of lacks in the specification.

The results of the current project show us the importance of the specification for the control systems and necessity to develop a special approach to design such systems. A BDD (Behaviour-driven developing) [7] is only one approach in software engineering now which is being used to develop and test the software based on a specification in natural form. It is proposed to extend the BDD language Gherkin to describe steps in the specifications in natural language and to describe the requirements as verification points. It will allow a software engineer to start work with the specification, implement the tests based on it, then implement the software based on steps and verify this software by generating necessary automatons and set-up the requirements. It is a subject of further research.

References

1. IEC 60456:2010. Clothes washing machines for household use - Methods for measuring the performance. https://webstore.iec.ch/publication/2188
2. Modbus protocol. http://www.modbus.org/specs.php
3. GOST IEC 6107-2011.Data exchange while reading meter values, tariffing and load management.
4. D. Alrajeh, J. Kramer, A. Russo, S. Uchitel. Elaborating Requirements using Model Checking and Inductive Learning. IEEE Transactions on Software Engineering (Volume: 39, Issue: 3, March 2013). pp. 361-383
5. Ang, K.H., Chong, G.C.Y., and Li, Y. (2005). PID control system analysis, design, and technology" (PDF). IEEE Trans Control Systems Tech, 13(4), pp.559-576.
6. Spin – Formal Verification Tool. http://spinroot.com
7. M.Wynne, A.Hellesoy, S.Tooke. The Cucumber Book, Second Edition. Behaviour-Driven Development for Testers and Developers. The Pragmatic Bookshelf, 2017, 336p. ISBN: 978-1-68050-238-1

A. A fragment of customer's specification of water normalisation process

```
Algorithm: Distillate cycle.
(Active before filling the large / storage tank)

Step 0: Start preparation ...
Step 1: The water pressure is checked at the input
(check sensor No ...) ...

Step 2: Filling the distiller
The level sensor in the distiller is monitored for
(time of opening the valve) after opening valve N...
Yes / parameters OK:
1) Valve ... is closed;
2) Transition to step 3.
No / the parameters are not normal:
1) Message (Filling of filling of the distiller);
2) The automatic mode is switched off.

Step 3: Distillation
1) The heating of the distiller is activated (Button ...);
2) Parameters are monitored:
- amperage;
- temperatures of the distiller;
- filling sensor;
- the sensor for filling the storage tank.
Yes / parameters OK:
1) Accumulation distillate in the tank
2) Executing step 3 again
No / the parameters are not normal:
- The amperage exceeds the set maximum:
```

```
1) The heating of the distiller is switched off;
2) The transition to 3.1.
- The amperage with the fill sensor turned on is below
the set minimum:
1) The heating of the distiller is switched off;
2) The transition to 3.2.
- The amperage with the filling sensor switched off is below the
set minimum:
Go to step 1.
- The temperature is higher than the given maximum:
1) The heating of the distiller is switched off;
2) The transition to 3.3.

Step 3.1: Draining of salted water ...
```

УДК 004.052.42, 004.4'6, 004.423.42, 004.432.2, 004.438 Eiffel, 519.681.2, 519.682.1

Making void safety practical

Alexander Kogtenkov (ETH Zürich, Switzerland; Eiffel Software, USA)

Null pointer dereferencing remains one of the major issues in modern object-oriented languages. An obvious addition of keywords to distinguish between never null and possibly null references appears to be insufficient during object initialization when some fields declared as never null may be temporary null before the initialization completes. The proposed solution avoids explicit encoding of these intermediate states in program texts in favor of statically checked validity rules that do not depend on special conditionally non-null types. Object initialization examples suggested earlier are reviewed and new ones are presented to compare applicability of different approaches. Usability of the proposed scheme is assessed on open-source libraries with a million lines of code that were converted to satisfy the rules.

Keywords: null pointer dereferencing, null safety, void safety, object initialization, static analysis, library-level modularity

1. Introduction

Tony Hoare [5] called his invention of the null reference a "billion-dollar mistake". The reason is simple: most object-oriented languages suffer from a problem of null pointer dereferencing. Even in a type-safe language, if an expression is expected to reference an existing object, it can reference none, or be *null*. Given that the core of object-oriented languages is in the ability to make calls on objects, if there is no object, the normal program execution is disrupted.

Not prevented at compile time, it remains one of the day-to-day issues. My analysis of the public database of cybersecurity vulnerabilities known as Common Vulnerabilities and Exposures (CVE®)[1] operated by *MITRE* and funded by Computer Emergency Readiness Team (*CERT*) reveals that in the past 10 years entries mentioning null pointer dereference bugs appear at consistent rate of about 70 bugs a year. As the database covers only software affecting the whole planet, real economy losses, caused by unlisted projects, are much higher.

To distinguish types of expressions that may return `null` from always returning an object, Raymie Stata [14] proposed a notation `T ?` for Java. Developers of the Checkers Framework[2]

[1] Common Vulnerabilities and Exposures. 2017. URL: `http://cve.mitre.org/` (visited on 2017-04-27).

[2] The Checker Framework 2.1.10. 04/03/2017. URL: `https://checkerframework.org/` (visited on 2017-05-08).

mention that now most static analyzers for Java use annotations `@Nullable` and `@NonNull`. Manuel Fähndrich and Rustan Leino [3] used C# attributes `[NotNull]` and `[MayBeNull]`. In different forms similar marks are used in Eiffel [6] (with type marks **attached** and **detachable**) and Kotlin [7] (with a mark `?`). Unfortunately, sequential initialization of object fields does not permit for non-null fields to be initialized with object references atomically.

Most solutions of the object initialization issue extend type systems to identify incompletely initialized objects. My review of open libraries showed that most code could be made null-safe without new type marks. Instead of tweaking the type system, I introduced compile-time validity rules for the remaining cases. With them, not only all examples from relevant publications [3; 4; 12; 16] could be compiled as expected, but new scenarios became feasible.

Together with removal of annotations for local variables [8], based on typing rules similar to those used in security data flow [17] and known as *flow-sensitive typing* [11], reduced annotation overhead simplifies adaptation of legacy code and makes null-safe programming more accessible.

2. Motivating examples

I Polymorphic call from a constructor. Manuel Fähndrich and Rustan Leino [3] describe a call to a virtual method on `this` in a superclass constructor. Because subclass fields of the object are not initialized yet, accessing them in the polymorphic call causes `NullReferenceException`. Xin Qi and Andrew C. Myers [12] consider a similar example with a class `Point` and its subclass `CPoint` that adds a color attribute.

II Polymorphic callback from a constructor. Accesses to an uninitialized object can be done indirectly. If a superclass constructor passes a reference to the current object as an argument to create another object, this "remote" constructor can call-back on the object where not all fields are initialized yet. A reasonable solution should distinguish between legitimate and non-legitimate calls to "remote" constructors to be sufficiently expressive and sound.

III Modification of existing structures. Convenience of the ability to invoke regular procedures inside a creation procedure can be demonstrated with a mediator pattern [1]. It decouples objects so that they do not know about each other, but still can communicate using an intermediate object, *mediator*. Concrete types of the communicating objects are unknown to the mediator, and, therefore, it cannot create them.

On the other hand, communicating objects know about the mediator and can register according to their role. If the registration is done in their constructors, clients do not need to clutter their code with calls to a special feature *register* after creating every new communicating object. The assignment like `x = new Comm (mediator)` should do both – recording a reference to the mediator and registration of the communicating object.

Registration of a new object may also be required in GUI libraries where a GUI-specific toolkit object has to keep references to the user-created object for event-based communication.

IV Safety violations. In addition to valid cases, authors usually mention examples that should trigger a compiler error (*e.g.*, Alexander J. Summers and Peter Müller [16]). This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

V Circular references. Another issue arises when two objects reference each other. If the corresponding fields have non-null types, access to them should be protected to avoid retrieving `null` by the code that relies on the field type.

Manuel Fähndrich and Songtao Xia [4] review a linked list example with a sentinel. When a new list is constructed, a special sentinel node is created and it should reference the original list object. In other words, an incompletely initialized list object has to be passed to a node constructor as an argument. An attempt to access field `sentinel` at this point would compromise null safety, so there should be means to prevent such accesses or to make them safe (*e.g.*, by treating field values as possibly null and as referring to uninitialized objects).

VI Self-referencing. This is a variant of circular references when an object references itself rather than another object. Xin Qi and Andrew C. Myers [12] review a binary tree where every node has a parent, and the root is a parent to itself.

At binary node creation, left and right nodes should get a new parent and the parent should reference itself. With any initialization order there are states when the new binary node should be used to initialize either its own field or field `parent` of its left or right nodes before it is completely initialized. Therefore, arbitrary accesses to this node should be protected like in the previous case.

3. Overview

3.1. Language conventions and terminology

Bertrand Meyer [10] pointed out that language rules can simplify or make it more difficult to achieve null safety guarantees. *E.g.*, in Java or C# a superclass constructor has to be called before the subclass constructor. Hence, non-null fields of the subclass cannot be initialized before calling superclass constructor. Without such restrictions, field initialization can be carried out in any suitable order that allows for fixing examples I and II without any new types.

I use Eiffel as an implementation testbed. The language specifies two type marks – **attached** (the default) and **detachable** – to denote non-null and maybe-null types respectively. Current object (`this` in Java and C#) is named **Current** and constructors are called *creation procedures*. They can also be used as regular routines, and are checked twice: as creation procedures for safe object initialization, and as regular procedures. Data members of a class are called *attributes*.

The language standard [6] introduces a notion of a *properly set* variable. For object initialization this means that all attributes of attached types should reference existing objects. By default, a field of a reference type does not reference an existing object, or is **Void**. If **Void** is used as a target of a call, the run-time raises an exception *"Access on void target"*. A compile-time guarantee that a system never causes such an exception is called *Void safety*.

3.2. Solution outline

All examples from the previous section can be divided into 2 major groups:

(A) Examples I to IV: – Can the code be reordered so that all fields are initialized before use?

(B) Examples V and VI: – Can compile-time rules ensure an object with recursive references to itself is not used as a completely initialized one?

The issue in group (A) arises because **Current** object is passed before all attributes of this object are properly set. The simplest rule would be to forbid using **Current** until all attributes are properly set:

Validity rule 1 (Creation procedure, strong)**.** *An expression* ***Current*** *is valid in a creation procedure or in an unqualified feature it (directly or indirectly) calls if all attributes of the current class are properly set at the execution point of the expression.*

The rule is sufficient to deal with group (A) by reordering initialization instructions.

But the rule is too strong for group (B). Of course, if a reference to an incompletely initialized object is leaked, the task to identify such an object becomes almost intractable not only in theory, but also due to complexity of implementing alias analysis correctly [2]. Explicit type annotations [3; 4; 12; 16] move detection of incompletely initialized objects from static analysis methods to the type system. I avoid performing alias analysis and extending the type system by preventing use of incompletely initialized objects in the first place.

The key source of obscurity in an object-oriented environment is polymorphism. Creation procedures are associated with specific classes, hence, no polymorphism is involved here. Even unqualified features they call can be checked for creation validity. The checks will make sure that class fields are not accessed before they are set and **Current** is completely initialized. But qualified calls are still an issue:

- a call on an incompletely initialized object cannot assume all attributes are properly set;
- a qualified call does not allow seeing what operations on an incompletely initialized object are performed.

The solution is to disallow qualified calls when some objects are incompletely initialized:

Validity rule 2 (Creation procedure, weak)**.** *A creation procedure is valid if any of the following is false at the same execution point:*

- ***Current*** *is used before all attributes have been properly set and not all attributes are properly set after that.*
- *The expression at the execution point is one of*
  - *a qualified feature call;*
  - *a creation expression that makes a qualified call.*

Unlike validity rule 1, the weak version assumes there is information, whether creation procedures of other classes make direct or indirect qualified calls. It could be explicitly or implicitly specified in creation procedure signatures, or inferred from code.

4. Related work

Raw types ***(solve examples I and IV with*** 2+ ***annotations).*** Manuel Fähndrich and K. Rustan M. Leino [3] denote attached types with T^- and detachable types with T^+ and propose to add raw types T^{raw-} to be used for partially initialized objects. If class C has an attribute of type T and some entity has type C^{raw-} then a qualified call to this attribute has type T^+

regardless of original attachment status of that attribute. An assignment to an entity of a raw type accepts only a source expression of a non-raw non-null type to ensure that if an object becomes fully initialized, it cannot be uninitialized. Also, by the end of every constructor, every non-null field should be assigned.

Then raw types are refined with class frames corresponding to superclasses. Inside a constructor of a class C, the special entity `this` has type C^{raw-}, and when the constructor finishes, the type becomes C^{-}. In a constructor of a super-class A the type of `this` is $C^{raw(A)-}$. The authors also specify conformance rules in this type system. Unfortunately, rules for super-class constructors, *e.g.*, for $T^{raw(R)-}$, are not directly applicable to languages with multiple class inheritance like Eiffel. And raw types do not support creation of circular references.

An implementation demonstrated that further extensions are required for real code, *e.g.*, to access fields that have been initialized and to indicate that a method initializes certain fields.

Masked types ***(solve examples I to VI with many annotations).*** Xin Qi and Andrew C. Myers [12] address the complete object life cycle. They instrument the type system with so called "masks" representing sets of fields that are not currently initialized. For example, the notation `Node\parent!\Node.sub[l.parent] -> *[this.parent]` for an argument `l` tells that it has a type `Node` and on entry requires that its field `parent` is not set and at the same time fields declared in subclasses of `Node` are not set unless `l.parent` is initialized. On exit the actual argument conforms to the type `Node\*[this.parent]` that indicates that the node object will be completely initialized as soon as its field `parent` is set.

The notation is very powerful and goes far beyond void safety, but even with its complexity authors complain that it is not sufficient for real programs. For information hiding they propose abstract masks updated in descendant classes as required. The idea looks similar to the data groups approach proposed by Rustan Leino in [9]. For modular processing of abstract masks, subclass masks and mask constraints are introduced with union and difference operations.

Like with masked types, validity rule 2 depends on whether class attributes are properly set and a reference to **Current** object escapes before that. Flow-sensitive type analysis is performed without special annotations too. However, with masked types the results are checked against provided specifications, while in my approach they are used to check validity rule conditions.

Free and committed types ***(solve examples I and IV to VI with*** 1+ ***annotations).*** Alexander J. Summers and Peter Müller distinguish [16] just two object states: under initial-

ization and completely initialized. A newly allocated object has a so called "free" type. When an object is deeply initialized, *i.e.*, all its fields are set to deeply initialized objects, it is said to have a "committed" type. The commitment point logically changes the type of an object from free to committed and is defined as the end of a constructor that takes only committed arguments. Possible aliasing between free and committed types is prevented by not having a subtyping relation between them. This differs from the convention for raw types [3].

Validity rule 2 is very close in spirit to the idea of free and committed types. But it relies on a flow-sensitive analysis and ceases free type status when all attributes are set. This allows for handling cyclic data structures without explicit annotations.

A variant of committed and free types is implemented in the Checker Framework with annotations `@UnknownInitialization` and `@UnderInitialization` supporting type frames `@UnderInitialization (A.class)` to tell that all fields specified in a (super)class `A` have been initialized. Authors of the Checker Framework claim that `this` cannot be used in a class constructor as `@Initialized`. This rules out examples II and III.

Other approaches *(solve examples I to VI with* 0 *annotations, non-modular).* Additional annotations are avoided by Bertrand Meyer in [10] using so called "targeted expressions" and creation-involved features. The analysis is somewhat similar to the abstract interpretation approach used by Fausto Spoto [13] and should be applied to the system as a whole, thus sacrificing modularity. This makes it difficult to develop self-contained libraries. The advantage of the approach is in selective detection of variables that are not completely initialized.

5. Formalization

Formalization of validity rules and proofs of their properties are done using the Isabelle/HOL proof assistant to avoid any inconsistencies and omissions. The theories code verified by *Isabelle2016*[3] is available at `https://bitbucket.org/kwaxer/void_safety/` (tag `1.2.5`).

Initialization state Validity rules are formalized using a simplified version of an Eiffel-like abstract syntax. The transfer function $\cdot \gg \cdot$ takes 2 arguments – an expression and a set of attributes V that may be unattached before the expression – and returns a set of attributes

[3]Isabelle2016. 01/16/2016. URL: `http://isabelle.in.tum.de/website-Isabelle2016/` (visited on 2017-05-07).

that may be unattached after the expression. At the beginning of a creation procedure the set of unattached attributes is a set of all current class attributes of attached reference types.

A validity predicate $V \vdash e \surd_c'$ tells if an expression e satisfies validity rule 1 in the context with unset attributes V.

Safe uses of Current If **Current** is never referenced in a creation procedure, there is no issue because the incompletely initialized object is not passed anywhere. If **Current** is referenced when all attributes are set, there is no issue as well: once an object is completely initialized, it remains completely initialized and can be freely used. Finally, if **Current** is referenced when not all attributes of the current class are set, but can escape only at the current execution point (*i.e.*, all previous expressions do not make any qualified calls, thus excluding the possibility to access this incompletely initialized object), it is possible that all attributes are set now and therefore the object is completely initialized regardless of its status when the reference to it escaped. These properties are captured by a function *safe*.

Detection of qualified feature calls For telling if a feature makes a qualified feature call, it is sufficient to analyze the corresponding abstract syntax tree. The function also takes care about qualified feature calls present in the features that are called from a current creation procedure using an unqualified feature call.

Another function is used to compute a set of creation procedures that can be called by the current one. Because the set of classes is known at compile time and is bounded, all creation procedures recursively reachable from the current one can be computed as a least fixed point.

Together with the function that tells whether a creation procedure has immediate qualified calls the function *has_qualified* tells if a creation procedure can lead to a qualified call.

Validity predicate A formal predicate $S,\ V \vdash e \surd_c$ for validity rule 2 is defined using functions *safe* and *has_qualified*. S stands for the current system to retrieve dependencies between creation procedures. The predicate is true as soon as $V \vdash e \surd_c'$ is true, i.e. validity rule 2 is more permissive than validity rule 1.

The predicate is monotone, so it is sufficient to analyze loops and unqualified feature calls just once, because any subsequent iterations or recursive feature calls would be analyzed with a larger set of properly-set attributes.

The soundness proof for object initialization is similar to the one given by Alexander J. Summers and Peter Müller [15] with two major differences. Firstly, the *free* status of a current object does not last until the end of a creation procedure, but only up to the point when all attributes are set, with the reservation that the creation procedure is not called by another one with an incompletely initialized **Current**. Secondly, annotations are replaced with the requirement to avoid qualified feature calls in the context with incompletely initialized objects.

For initialization of **Current** two situations are possible. In the first case all attributes of the current class are set and there are no incompletely initialized objects in the current context. Then the current object is deeply initialized and can be freely used before the creation procedure finishes. In the second case either some attributes of the current class are not properly set or the context has references to objects that are not completely initialized. Because qualified calls are disallowed in these conditions, the uninitialized attributes cannot be accessed and access on void target is impossible. Due to the requirement to set all attributes at the end of a creation procedure, all these objects will have all attributes set, and, taking into account that the only reachable objects are either previously fully initialized or are new with all attributes pointing to the old or new objects, *i.e.*, also fully initialized, all objects become fully initialized in the context where all attributes of the current class are set and no callers passed an uninitialized **Current**.

6. Practical results

Although validity rule 1 looks pretty restrictive, 4254 classes of public libraries have been successfully converted relying on this rule. This comprises 822487 lines of code and 3194 explicit creation procedures. 59% of these creation procedures (1894 in absolute numbers) perform regular direct or indirect qualified calls and might be in danger if not all attributes were set before **Current** was used. However, it was possible to refactor all the classes to satisfy the rule.

On average, 60% of creation procedures make qualified calls. Remaining 40% do not use any qualified calls and set attributes using supplied arguments or by creating new objects. They could be unconditionally marked with annotations as safe for use with incompletely initialized objects.

In contrast to this, just a tiny fraction of all creation procedures – 77 creation procedures from two libraries, or less than 2% – do pass uninitialized objects and take advantage of the

weaker validity rule 2. In other words, if specific annotations were used, at most 5% of them would be useful, the rest would just clutter the code.

The validity rule checks for creation procedures are pretty light. The libraries were compiled with and without checks for validity rule 2 on a machine with 64-bit *Windows 10 Pro*, *Intel® Core™ i7-3720QM*, 16GB of RAM and SSD hard drive using *EiffelStudio 16.11 rev.99675*. For all libraries the slowdown was just 0.7% that seems to be more than acceptable.

7. Conclusion

Proposed solutions for the object initialization issue have the following benefits:

No annotations. Validity rules do not require any other type annotations in addition to attachment marks.

Flexibility. Creation of objects mutually referencing other objects is possible.

Simplicity. The analyses require only tracking for attributes that are not properly set, for use of **Current** and for checking whether certain conditions are satisfied when (direct or indirect) qualified feature calls are performed.

Coverage. It was possible to refactor all libraries to meed the requirements of the rules without changing design decisions. The rules solve all examples from the motivation section.

Modularity. Validity rule 2 depends on properties of creation procedures from other classes. Because these creation procedures are known at compile time, the checks do not depend on classes that are not directly reachable from the one being checked. Therefore, a library can be checked as a standalone entity without the need to recheck it after inclusion in some other project.

Performance. Experiments demonstrate very moderate increase of total compilation time, below 1% on sample libraries with more than 2 millions lines of code.

Incrementality. Fast recompilation is supported if information about reachable creation procedures and whether they perform qualified calls is recorded for every class.

Main drawbacks of the rules are:

Certain coding pattern. Certain initialization order have to be followed.

Disallowing legitimate qualified calls. Lack of special annotations prevents from distinguishing between legitimate and non-legitimate qualified calls. To preserve soundness all qualified calls are considered as potentially risky.

Special convention for formal generics. If a target type of a creation expression is a formal generic parameter, special convention should be used to indicate whether a creation procedure of an actual generic parameter satisfies the validity rule requirements.

References

1. Design Patterns: Elements of Reusable Object-oriented Software / E. Gamma [et al.]. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN 0-201-63361-2.
2. Effective Dynamic Detection of Alias Analysis Errors / J. Wu [et al.] // Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. — Saint Petersburg, Russia : ACM, 2013. — Pp. 279–289. — (ESEC/FSE 2013). — ISBN 978-1-4503-2237-9. — DOI: 10.1145/2491411.2491439.
3. *Fähndrich M.*, *Leino K. R. M.* Declaring and Checking Non-null Types in an Object-oriented Language // Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. — Anaheim, California, USA : ACM, 2003. — Pp. 302–312. — (OOPSLA '03). — ISBN 1-58113-712-5. — DOI: 10.1145/949305.949332.
4. *Fähndrich M.*, *Xia S.* Establishing Object Invariants with Delayed Types // Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. — Montreal, Quebec, Canada : ACM, 2007. — Pp. 337–350. — (OOPSLA '07). — ISBN 978-1-59593-786-5. — DOI: 10.1145/1297027.1297052.
5. *Hoare T.* Null references: The billion dollar mistake // Presentation at QCon London. — 2009.
6. *ISO.* ISO/IEC 25436:2006(E): Information technology — Eiffel: Analysis, Design and Programming Language. — 1st. — Geneva, Switzerland : ISO (International Organization for Standardization), IEC (International Electrotechnical Commission), 12/01/2006.
7. *JetBrains.* Kotlin Language Specification. — 01/31/2017. — URL: https://jetbrains.github.io/kotlin-spec/kotlin-spec.pdf (visited on 2017-01-31).
8. *Kogtenkov A.* Mechanically Proved Practical Local Null Safety // Proceedings of the Institute for System Programming of the RAS. — Moscow, Russia, 2016. — Dec. — Vol. 28, no. 5. —

Pp. 27–54. — ISSN 2079-8156 (Print), 2220-6426 (Online). — DOI: 10.15514/ISPRAS-2016-28(5)-2.

9. *Leino K. R. M.* Data Groups: Specifying the Modification of Extended State // Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. — Vancouver, British Columbia, Canada : ACM, 1998. — Pp. 144–153. — (OOPSLA '98). — ISBN 1-58113-005-8. — DOI: 10.1145/286936.286953.
10. *Meyer B.* Targeted expressions: safe object creation with void safety. — 07/30/2012. — URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).
11. *Pearce D. J.* On Flow-Sensitive Types in Whiley. — 09/22/2010. — URL: http://whiley.org/2010/09/22/on-flow-sensitive-types-in-whiley/ (visited on 2017-05-07).
12. *Qi X.*, *Myers A. C.* Masked Types for Sound Object Initialization // Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Savannah, GA, USA : ACM, 2009. — Pp. 53–65. — (POPL '09). — ISBN 978-1-60558-379-2. — DOI: 10.1145/1480881.1480890.
13. *Spoto F.* Precise null-pointer analysis // Software & Systems Modeling. — 2011. — Vol. 10, no. 2. — Pp. 219–252. — ISSN 1619-1366. — DOI: 10.1007/s10270-009-0132-5.
14. *Stata R.* ESCJ 2: Improving the safety of Java. — 12/02/1995. — URL: http://kindsoftware.com/products/opensource/ESCJava2/ESCTools/docs/design-notes/escj02.html (visited on 2017-04-27).
15. *Summers A. J.*, *Müller P.* Freedom before commitment: simple flexible initialisation for non-full types: tech. rep. / ETH Zurich, Department of Computer Science. — Zurich, Switzerland, 2010. — No. 716. — DOI: 10.3929/ethz-a-006904372.
16. *Summers A. J.*, *Müller P.* Freedom Before Commitment: A Lightweight Type System for Object Initialisation // Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. — Portland, Oregon, USA : ACM, 2011. — Pp. 1013–1032. — (OOPSLA '11). — ISBN 978-1-4503-0940-0. — DOI: 10.1145/2048066.2048142.
17. *Volpano D.*, *Irvine C.*, *Smith G.* A Sound Type System for Secure Flow Analysis // Journal of Computer Security. — Amsterdam, The Netherlands, The Netherlands, 1996. — Jan. — Vol. 4, no. 2/3. — Pp. 167–187. — ISSN 0926-227X. — URL: http://dl.acm.org/citation.cfm?id=353629.353648.

UDC 004.052.42

Verification of Definite Iteration over Arrays with a Loop Exit in C Programs

Maryasov I. V. (A. P. Ershov Institute of Informatics Systems SB RAS)

Nepomniaschy V. A. (A. P. Ershov Institute of Informatics Systems SB RAS)

Kondratyev D. A. (A. P. Ershov Institute of Informatics Systems SB RAS)

This work represents the further development of the method for definite iteration verification [6]. It extends the mixed axiomatic semantics method [1] suggested for C-light program verification. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system. To prove generated verification conditions an induction is applied which is a challenge for SMT-solvers. At proof stage the SMT-solver Z3 is used in our verification system. To overcome mentioned difficulty a rewriting strategy for verification conditions is suggested. It allows to verify the definite iteration automatically using Z3. Also the paper describes the application of the theorem prover PVS for automatic proving of such verification conditions. An example, which illustrates the application of these methods, is considered.

This research is partially supported by RFBR grant 15-01-05974.

Keywords: C-light, loop invariants, mixed axiomatic semantics, definite iteration, arrays, Z3, PVS, specification, verification, Hoare logic

1. Introduction

C program verification is an important task nowadays. A lot of projects (e. g. [2, 3]) propose different solutions. But none of the mentioned above suggests any methods for loop verification without invariants whose construction is a challenge. Therefore, the user has to provide these invariants. In many cases it is a difficult task. Tuerk [12] suggested to use pre- and post-conditions for while-loops but the user still has to construct them himself.

Our method describes a class of loops, which can be verified automatically without any invariants or loop pre- and post-conditions. It deals with a definite iteration of a special form. We extend our mixed axiomatic semantics of the C-light language [1] with a new rule

for verification of such iterations. This extension includes a verification method for definite iteration over unchangeable arrays with a loop exit in C-light programs. The method includes an inference rule for the iteration without invariants, which uses a special function that expresses loop body. This rule was implemented in verification conditions generator, which is the part of our C-light verification system.

At the proof stage, the SMT-solver Z3 [9] and the proof assistant PVS [11] are used. A rewriting strategy for the induction based verification conditions was suggested to prove them in Z3.

2. Definite Iteration and Replacement Operation

The method of loop invariants elimination for definite iteration was suggested in [10]. It includes four cases:

1. Definite iteration over unchangeable data structures without loop exits.
2. Definite iteration over unchangeable data structures with a loop exit.
3. Definite iteration over changeable data structures with a loop exit.
4. Definite iteration over hierarchical data structures with a loop exit.

The first case was considered in [6]. Our paper deals with the second case.

Let us remind the notion of data structures, which contain a finite number of elements. Let $memb(S)$ be the multiset of elements of the structure S and $|memb(S)|$ be the power of the multiset $memb(S)$. For the structure S the following operations are defined:

1. $empty(S) = true$ iff $|memb(S)| = 0$.
2. $choo(S)$ returns an element of $memb(S)$ if $\neg empty(S)$.
3. $rest(S) = S'$, where S' is a structure of the type of S and $memb(S') = memb(S) \setminus \{choo(S)\}$ if $\neg empty(S)$.

Sets, sequences, lists, strings, arrays, files, and trees are typical examples of the data structures.

Let $\neg empty(S)$, then $vec(S) = [s_1, s_2, \ldots, s_n]$ where $memb(S) = \{s_1, s_2, \ldots, s_n\}$ and $s_i = choo(rest^{i-1}(S))$ for $i = 1, 2, \ldots, n$.

Consider the statement

$$\textbf{for x in S do v} := \textbf{body}(\mathbf{v}, \mathbf{x})\ \textbf{end}$$

where S is a structure, x is the variable of the type "an element S", v is a vector of loop variables which does not contain x and *body* represents the loop body computation, which

does not modify x and S, and which terminates for each $x \in memb(S)$. The loop body can contain only the assignment statements, the **if** statements and the **break** statements. Such **for** statement is named a definite iteration.

The operational semantics of such statement is defined as follows. Let v_0 be the vector of initial values of variables from v. If $empty(S)$ then the result of the iteration $v = v_0$. Otherwise, if $vec(S) = [s_1, s_2, \ldots, s_n]$, then the loop body iterates sequentially for x taking the values $s_1, s_2, \ldots, s_n$.

To express the effect of the iteration let us define a replacement operation $rep(v, S, body, n)$, where $rep(v, S, body, 0) = v$, $rep(v, S, body, i) = body(rep(v, S, body, i-1), s_i)$ for all $i = 1, 2, \ldots, n$ if $\neg empty(S)$.

A number of theorems, which express important properties of the replacement operation, were proved in [10].

The inference rule for definite iterations has the form:

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v')\ \wedge v = rep(v', S, body)\}\ \mathbf{A};\{Q\}}{E, SP \vdash \{P\}\ \mathbf{for\ x\ in\ S\ do\ v} := \mathbf{body(v, x)\ end\ A};\{Q\}}$$

Here A are program statements after the loop. We use forward tracing: we move from the program beginning to its end and eliminate the leftmost operator (at the top level) applying the corresponding rule of the mixed axiomatic semantics [1] of C-light. E is the environment which contains an information about current function (its identifier, type and body) which is verified, an information about current block and label identifier if **goto** statement occurred earlier. SP is program specification which includes all preconditions, postconditions, and invariants of loops and labeled statements.

Let S be a one-dimensional array of n elements. Consider the special case of definite iteration

$$\mathbf{for\ (i = 0; i < n; i++)\ v} := \mathbf{body(v, i)\ end}$$

where $\mathbf{v} := \mathbf{body(v, i)}$ consists of assignment statements, **if** statements and **break** statements.

In order to generate verification conditions we have to determine v, $body(v, i)$, and the function rep.

Let the loop body has the form

$$\{\mathbf{x_1} = \mathbf{expr_1}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});$$
$$\mathbf{x_2} = \mathbf{expr_2}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});$$
$$\ldots$$
$$\mathbf{x_k} = \mathbf{expr_k}(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_k});\}$$

where $expr_j (j = 1, 2, \ldots k)$ are some C-light expressions.

The vector v of loop variable consists of all variables from left parts of assignment statements: $v = (x_1, x_2, \ldots, x_k)$. From the statements before the loop, we can get the initial value of v and obtain the first axiom for *rep*:

$$rep(v, S, body, 0) = (x_{1_0}, x_{2_0}, \ldots, x_{k_0})$$

where $x_{j_0}, j = 1, 2, \ldots, k$ are the initial values of x_j before the loop execution.

At the next step, we make consecutive substitutions

$$x_1 = expr_1(x_1, x_2, \ldots, x_k);$$

$$x_2 = expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k);$$

$$\ldots$$

$$x_k = expr_k(expr_1(x_1, x_2, \ldots, x_k), expr_2(expr_1(x_1, x_2, \ldots, x_k), x_2, \ldots, x_k), \ldots, x_k);$$

And then in the right parts $rep((x_1, x_2, \ldots, x_k), S, body, i-1)$ is substituted for x_j.

For each **if** statement of the form $\mathbf{if}\,(\mathbf{e(i, x_1, x_2, \ldots, x_k)})\{\mathbf{A};\}\,\mathbf{else}\,\{\mathbf{B};\}$, where **A** and **B** are compound statements consisting of assignment statements, two axioms are added to the output of verification conditions generator:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow A^*$$

$$\forall x_1 \forall x_2 \ldots \forall x_k \; \neg e(i, x_1, x_2, \ldots, x_k) \Rightarrow B^*$$

where A^* and B^* are obtained by consecutive substitutions as described above.

The **break** statement could appear at the top level of the loop or in the **if** statement. The first case is obvious, it means that the loop iterates no more than once and all the statements after **break** in the loop body are ignored. Therefore, the function *rep* is defined for $i = 0, 1$.

The second case means that for some j such that $0 < j \leq n$ a loop exit occurs and such j is defined by the condition of the **if** statement. Therefore, for all i such that $j \leq i \leq n$

$$rep((x_1, x_2, \ldots, x_k), S, body, i) = rep((x_1, x_2, \ldots, x_k), S, body, j)$$

In this case the following axiom is added:

$$\forall x_1 \forall x_2 \ldots \forall x_k \; e(i, x_1, x_2, \ldots, x_k) \Rightarrow (A^* \wedge (\forall l \, i < l \Rightarrow A^*))$$

For the case when the **break** statement is located in the **else** statement, the negation of e is used.

3. Example

Let us demonstrate the application of our method. Consider the following function **search**. For a given integer *key* it finds its first occurrence in the given array of integers *arr* consisting of *length* elements. If *key* does not occur in *arr* the function returns -1.

The annotated (in SMT-LIB v2 syntax of Z3) C-light program has the form:

```
/* (assert (> length 0)) */
int search(int* arr, int length)
{
    auto int result = -1;
    for (i = 0; i < length; i++)
     if (key == arr[i])
     {
      result = i;
      break;
     };
    return result;
}
/* (assert (or
            (forall ((i Int))
             (implies (and (< -1 i) (< i length))
                      (and (not (= key (select arr i))) (= result -1))))
            (exists ((r Int))
             (implies (and (< -1 r) (< r length))
                      (forall ((i Int))
                       (implies (and (< -1 i) (< i r))
                                (and (not (= key (select arr i)))
                                     (= key (select arr r))
                                     (= result r)))))))) */
```

In this function $v = (result)$ and its initial value before the iteration is -1. Also note that in Z3 all functions must be total. Therefore, the first axiom is:

```
(declare-fun rep (Int Int (Array Int Int) Int) Int)
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies (< i 1)
                  (= (rep result key arr i) -1))))
```

According to our method described in Sec. 2, the second and the third axioms are:

```
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies (and (< 0 i) (= (select arr (- i 1)) key))
```

```
                (and (= (rep result key arr i) (- i 1))
                     (forall ((j Int))
                      (implies (< i j)
                               (= (rep result key arr j) (- i 1))))))))
(assert (forall ((i Int) (key Int) (arr (Array Int Int)) (result Int))
         (implies
                (and (< 0 i)
                     (not (= (select arr (- i 1)) key)))
                (= (rep result key arr i) (rep result key arr (- i 1))))))
```

Z3 is the SMT-solver but our task is to check a validity of verification conditions, not satisfiability. Therefore, the verification conditions generator produces the negation of the verification condition:

```
(assert (not
         (forall ((result Int) (key Int) (length Int) (arr (Array Int Int)))
          (implies (and
                    (> length 0)
                    (= result (rep result key arr length)))
                   (or
                    (forall ((i Int))
                     (implies (and (< -1 i) (< i length))
                              (and (not (= key (select arr i)))
                                   (= result -1))))
                    (exists ((r Int))
                     (implies (and (< -1 r) (< r length))
                              (forall ((i Int))
                               (implies (and (< -1 i) (< i r))
                                        (and (not (= key (select arr i)))
                                             (= key (select arr r))
                                             (= result r))))))))))))
```

And then we expect the answer “unsat” which means that the negation is unsatisfiable therefore the verification condition is true.

However, Z3 does not support proofs by induction, which appears inevitably in our veri-

fication method. In this example we get the answer "unknown" which means that Z3 is not able to determine whether the formula is satisfiable or not. Rustan Leino suggested a rewriting strategy and a heuristic for when to apply it to verify simple inductive theorems [5].

4. Working with Induction in Z3

To prove by induction some proposition $\forall n\ P(n)$ we try to prove the equivalent formula $\forall n\ (\forall k\ k < n \Rightarrow P(k)) \Rightarrow P(n)$. In this way we rewrite the verification condition for Z3 be able to prove it. We should add an extra axiom (induction hypothesis) of the form $\forall n\ \forall k\ k < n \Rightarrow P(k)$ and modify the verification condition by adding a base case of induction $P(1)$. In our case of definite iteration over unchangeable one-dimensional arrays the inductive variable is always the length of array. Therefore, the verification conditions generator is able to rewrite the verification condition which contains a *rep* function automatically.

In the example from Sec. 3 this extra axiom has the form:

```
(assert (forall ((result Int) (key Int) (length Int) (len Int)
                 (arr (Array Int Int)))
  (implies (and
            (> len 0)
            (> length len)
            (= result (rep result key arr len)))
           (or
            (forall ((i Int))
             (implies (and (<= 0 i) (< i len))
                      (and (not (= key (select arr i))) (= result -1))))
            (exists ((r Int))
             (implies (and (<= 0 r) (< r len))
                      (forall ((i Int))
                       (implies (and (<= 0 i) (< i r))
                                (and (not (= key (select arr i)))
                                     (= key (select arr r))
                                     (= result r)))))))))))
```

And the base case is when $length = 1$. Therefore we add the second verification condition which is obtained from the one in Sec. 3 by replacing the first conjunct `(> length 0)` with `(=`

```
length 1).
```

After adding this axiom and the second verification condition, Z3 immediately returns the answer "unsat" which means that the verification condition is true. Thus, the program is partially correct.

5. Using PVS to Prove Inductive Verification Conditions

Another approach to the implementation of our method is based on the meta verification condition generation (MetaVCG) [8] for building verification conditions and using PVS for its proving. Detailed information about the MetaVCG can be found in [4].

The PVS language allows us to define the theory, which is the input argument of PVS theorem prover. The PVS language is a functional programming language supplemented with constructs for defining higher-order logic sentences. Thus, a PVS theory contains type definitions, functions, and formulas. The PVS theorem prover can be applied to a certain formula of the theory or to all formulas of the theory.

Let us consider the process of verification of the example from Sec. 3. The following theory was generated:

```
search: THEORY
BEGIN
  rep(result:int, key:int, arr:ARRAY[nat->int], i:nat) : RECURSIVE int =
     IF (i < 1) THEN result
     ELSE (IF ((0 < i) AND (arr(i-1) = key)) THEN rep(i-1, key, arr, i-1)
           ELSE rep(result, key, arr, i-1) ENDIF) ENDIF
  MEASURE i
  vc: LEMMA
      FORALL (result:int, key:int, length:nat, arr:ARRAY[nat -> int]):
             ((length > 0) AND (result = rep(-1, key, arr, length)))
      IMPLIES
      ((FORALL (i:int): ((0 <= i) AND (i < length) AND (not (key = arr(i))))
       IMPLIES
       (result = -1))
      OR
      (EXISTS (r:int): ((0 <= r) AND (r < length))
```

```
        IMPLIES
        (FORALL (i:int): ((0 <= i) AND (i < r) AND (not (key = arr(i)))
                          AND (key = arr(r))))
         IMPLIES (result = r)))
END search
```

Note that it is necessary to provide a measure for the recursive function *rep*. The measure is a well-founded relation. As mentioned in Sec. 4 the length of array is the appropriate measure for the definite iteration over unchangeable arrays with a loop exit, therefore the measure could be provided to PVS automatically.

PVS has special inference rules, which allow to use induction. In the case under consideration the construct `(induct-and-simplify "length")` is used. It tells the prover to apply the induction with the variable *length*. This leads to successful automatic proving of two formulas: the base case and the induction step.

6. Conclusion

This paper represents an extension of our system [7] for C-light program verification. In the case of definite iteration over unchangeable arrays with a loop exit, this extension allows us to generate verification conditions without loop invariants.

This generation is based on the described inference rule for the C-light **for** statement which introduces the replacement operation. It expresses definite iteration in the special form described in the paper.

The suggested rewriting strategy for the induction allowed us to prove obtained verification conditions in Z3. Also they are automatically proved in PVS.

The rewriting strategy allowed Z3 to prove automatically the partial correctness of the example from [6]. It iterates over an array of integers and for a given integer computes the number of its occurrences in this array. Also, PVS was able to prove automatically the partial correctness of this example.

We plan to improve the suggested algorithm of *rep* function determination for the case of nested **if** statements and to prove its correctness. Also, we will consider the case of loop elimination for changeable data structures and verify automatically classical array sorting programs.

References

1. Anureev I. S., Maryasov I. V., Nepomniaschy V. A. C-programs Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2011. — Vol. 45 — Issue 7. — P. 485–500.
2. Cohen E., Dahlweid M., Hillebrand M., Leinenbach D., Moskal M., Santen T., Schulte W., Tobies S. VCC: A Practical System for Verifying Concurrent C // Proc. of 22nd Int. Conf. TPHOLs. — LNCS. — 2009. — Vol. 5674. — P. 23–42.
3. Filliâtre J.-C., Marché C. Multi-prover Verification of C Programs // Proc. of 6th ICFEM. — LNCS. — 2004. — Vol. 3308. — P. 15–29.
4. Kondratyev D. A. The Extension of the MetaVCG Approach by Semantic Mark-up Concept // Proc. of the Int. Workshop-conf. "Tools & Methods of Program Analysis". — St. Petersburg, 2015. — P. 107–118.
5. Leino K. R. M. Automating Induction with an SMT Solver // Proc. of 13th Int. Conf. VMCAI. — LNCS. — 2012. — Vol. 7148. — P. 315–331.
6. Maryasov I. V., Nepomniaschy V. A. Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs // Modeling and Analysis of Information Systems. — 2015. Vol. 22 — Issue 6. — P. 773–782.
7. Maryasov I. V., Nepomniaschy V. A., Promsky A. V., Kondratyev D. A. Automatic C Program Verification Based on Mixed Axiomatic Semantics // Automatic Control and Computer Sciences. — 2014. — Vol. 48 — Issue 7. — P. 407–414.
8. Moriconi M., Schwarts R. L. Automatic Construction of Verification Condition Generators From Hoare Logics // Automata, Languages and Programming. — LNCS. — 1981. — Vol. 115. — P. 363–377.
9. Moura L. de, Bjørner N. Z3: An Efficient SMT Solver // Proc. of 14th Int. Conf. TACAS 2008. — LNCS. — 2008. — Vol. 4963. — P. 337–340.
10. Nepomniaschy V. A. Verification of Definite Iteration over Hierarchical Data structures // Proc. of FASE/ ETAPS 1999. — LNCS. — 1999. — Vol. 1577. — P. 176–187.
11. Owre S., Rushby J. M., Shankar N. PVS: A Prototype Verification System // 11th Int. Conf. CADE 1992. — LNAI. — 1992. — Vol. 607. — P. 748–752.
12. Tuerk T. Local Reasoning about While-Loops // VSTTE 2010. Workshop Proceedings. — 2010. — P. 29–39.

УДК 004.416.3+004.4'242

Automated Semantics-Driven Source Code Migration: a Pilot Prototype

Artyom Aleksyuk (Peter the Great St. Petersburg Polytechnic University, Russia; JetBrains Research)

Vladimir Itsykson (Peter the Great St. Petersburg Polytechnic University, Russia; JetBrains Research)

The purpose of the study is to demonstrate the feasibility of automated code migration to a new set of programming libraries. Code migration is a common task in modern software projects. For example, it may arise when a project should be ported to a new library or to a new platform. The developed method and tool are based on the previously created by the authors formalism for describing libraries semantics. The formalism specifies a library behavior using a system of extended finite state machines (EFSM). The mentioned EFSMs are a foundation of the code migration method.

This paper outlines the metamodel designed to specify library descriptions and proposes easy to use domain-specific language (DSL), which can be used to define models for particular libraries. The mentioned metamodel directly forms the code migration method which is also described in the paper. A process of migration splits into five steps, and for each step the algorithm was developed.

Models and algorithms were implemented in the prototype of an automated code migration tool. The prototype was tested on both artificial code examples and several real-world open source projects. Results of the experiment indicate that code migration can be successfully automated with developed tool acting as the proof of concept. Models and methods designed form a basis for more powerful migration methods and full-featured automated code migration tools.

Keywords: software library, code migration, behavioral description, program transformation

1. Introduction

This study is devoted to automated program migration, which is an actual software engineering task. Usually developers are faced with this task when they are going to upgrade their project to a new version of library or port their projects to a more effective, more secure or more functional library. Usually the source project has been properly tested and the developer

wants to be sure that the ported project has an equal level of quality. A manual migration does not guarantee the expected quality level. Developers have to test the migrated project from scratch as a new one.

The aims of our research are

- to develop the new migration method based on formal library specifications which is fully automated and preserves the quality of the project;
- to prove the feasibility of our approach.

The proposed approach is based on previously created by the authors formalism for library specification description. The mentioned formalism represents libraries as a set of extended finite state machines (EFSM). Each EFSM reflects the lifecycle of the whole library or its entities. Examples of such entities are statically or dynamically created files, sockets, semaphores, streams, mutexes, threads, etc. States of EFSM are states of objects, vertices represent library calls. Full specification of the formalism can be found in [1].

One of the main aims of this work is to develop an experimental tool which can be used for proving feasibility of automated model-based migration. To do this we designed a simple DSL for specifying libraries' behavior. These specifications are used as an input for the multistep method which transforms source project into the new one according to models of both the old and the new libraries.

To evaluate our approach, we have conducted a set of experiments with artificial and real-world open source projects. All examined projects were ported correctly by our tool. These results are a proof of applicability of the proposed approach.

The rest of the paper organized as follows. The first section contains the description of the state of the art. In the second section the proposed approach to migration is introduced. The third section is devoted to the implementation of the tool prototype that proves feasibility of our approach. In the fourth section developed tool is evaluated on the artificial and real-world projects. In the conclusion the obtained results are analysed and possible directions of a future research are discussed.

2. State of the Art

Firstly, we need to outline the migration task. Suppose we have a software system using a specific set of libraries. The task is to port a program to another set of libraries without rewriting the system from the scratch. Below we consider several approaches to an automated

solution of this task:

- migration with a help of manually written wrapper libraries;
- migration by running a syntax-level porting tool;
- migration by applying a semantics-level porting tool.

One of the most common ways to migrate a program to another library is to use manually written wrappers [2]. Wrapper is a small library which has the same interface as the source library and uses the destination library to actually perform a work.

Complexity of wrappers may vary a lot. The simplest ones just redirect function calls. This approach is sufficient if a destination library has functions with the same set of arguments and has the same semantics. More complex wrappers may also apply data format transformations.

There are several downsides of the wrapper approach:

- wrappers are usually written for specific source and destination libraries;
- complex wrappers can severely affect the performance;
- wrappers usually restrict access to destination library's structures and/or methods;
- if some parts of the program directly use a destination library and other parts use it through a wrapper, it makes harder to exchange data structures between these parts.

Despite all the limitations listed above the wrapper approach can be utilized for some libraries. Several known wrapper projects are listed below:

- ANGLE is an OpenGL ES implementation on top of the Direct3D [3];
- SLF4J (Simple Logging Facade for Java) is a unified logging framework which can use several logging implementations as a backend [4].

Migration by running a syntax-level porting tool is another available approach. It differs from the wrapper approach in that this one actually ports software by changing its source code. These tools primarily rely on the information about a syntax of programming language and are based on template replacement or term rewriting.

One example of usage of the template replacement method is IntelliJ IDEA [5] with its built-in feature called Structural Search and Replace. This feature is very similar to a familiar search and replace tool found in most text editors, however it is aware of a programming language syntax. For example, it is able to find code which has another formatting style. Also, it can match code with small insignificant differences such as class fields declaration order.

An example of a term rewriting software is TXL tool [6]. It relies on its own functional language for specifying rewriting rules. To define a programming language syntax it uses

the extended Backus–Naur form. The development of a term rewriting approach leads to appearance of rewriting strategy concept implemented in Stratego/XT [7] and DMS [8] tools.

An example of a practical usage of this approach to migrate program is described in [9]. The authors of this paper wrote a set of rewriting rules to help migrate programs from Qt 3 to Qt 4. During the evaluation phase, they managed to partially migrate kdelibs project to a newer version of Qt library.

The main drawback of the syntax-level migration approach is an ability to perform only the simplest replacements like method name change or call arguments reorder [10]. More complex replacements require to write large templates or are totally impossible with this approach.

The approaches described above have a limited power to automatically migrate source code in case of a significant difference between a source and a destination libraries. An additional information such as a library semantics is needed. In this paper, we use a more powerful and universal semantics-based approach.

One of the recent articles [11] extends the pattern-based approach with semantics information to better match program elements which needs to be migrated. However, the replacement process is still controlled by the set of templates and so is not capable to perform complex changes to the source code.

3. The Proposed Approach

The key part of the proposed migration approach is a library metamodel. The design of the metamodel is mostly influenced by the following criteria:

- the complexity of the analysis;
- the ability to express libraries' semantics;
- the ease of library model construction.

In this paper, we use the previously created by the authors formalism [1] as a base for the metamodel. The mentioned formalism was created to be used in a wide range of areas, including the defect detection and software mining[1].

The metamodel is a set of EFSMs (extended finite state machines). Each EFSM represents a semantic entity which library can process or use for its operations. For example, a File I/O library will possibly have entities named like "File", "Filename", "Stream", etc. Usually each class in a library has a corresponding EFSM, but this is not a strict rule. EFSMs are defined

[1]It should be noted that the formalism is still being developed and should not be treated as a final work

by tuple $< Q, Q_0, X, V, C, C^A, C^D, U, F, T >$.

Each library's EFSM has a set of states Q. For example, the "File" automaton can have states like "Opened" and "Closed". Also, each automaton declares a set of transition relations T and the non-empty set of initial states Q_0. X is the set of finish states. An example is a *Closed* state for a *File* entity. C is the set of function calls, constructor calls and other code elements acting as stimuli for state transitions.

Some code elements return new entities after execution. C_i^D is the set of child automatons created when C_i code element is being activated.

Most complex libraries cannot be fully described with just states and transitions, so we decided to extend the library metamodel with properties V and semantic actions C^A. Each EFSM may have a set of properties identified by name and containing an arbitrary value (strings, integers, etc.). Properties may be modified by update functions U. Transition between states is only possible if guard condition predicate $F(V)$ is true.

Metamodel actions define semantically significant events which cannot be expressed by a property change or a state transition. Actions are implemented as a transition attribute. Also, they may have arbitrary parameters just like properties. For example, a change of library settings may be defined by an action. More details about actions and properties may be found in the article [1].

4. Method

The proposed migration method consists of five steps.

The first step is the **trace extraction**. A trace contains a list of code elements placed in the order of activation. Any suitable approach can be used to do this task.

The second step is the **trace mapping**. This step includes fetching the program trace and mapping it onto the source library model.

The third step named **equivalent trace calculation** is the most interesting one. Equivalent trace calculation is an immediate process of searching a replacement for a source model trace. The resulting model trace should use a destination library and must be semantically equivalent. One of the most important advantages of the proposed method is that a transformation is done in the metamodel context.

The replacement search is driven by two factors. Firstly, if a source trace contains entity creation, a resulting trace must also have it. For example, if a source (migrated) program fetches

a file length, i.e. creates a FileLength entity, a resulting trace should also create this entity. This is necessary because a source program may pass the created entity to another function or store in a class field, and migrated program must do the same. Secondly, the resulting trace must contain the same set of actions as the source one.

The algorithm currently used by the tool is based on a breadth-first search (BFS). The library model is treated as a graph and solution of the shortest path problem can be viewed as a replacement sequence, where the sought-for vertex is a required state of the EFSM. The search is simultaneously started from several vertices (all library entities available in the scope). If the equivalent path is not found, a user help is needed.

At first, the library model is being converted into the graph representation. Each extended finite state machine is transformed into the separate graph, and then all graphs are merged into the single one. The resulting graph contains several vertex clusters which correspond to each EFSM.

Actions and properties from the library model put additional restrictions for the acceptable solution. To take these restrictions into account, we define a new graph G' where each vertex of G' is a possible state of a traversal (after visiting some number of edges), and where there is an edge $u \to v$ if starting from state u it is possible to add one edge to the path to reach state v.

Traditional BFS algorithm keeps two queues: a visited queue which contains vertices which were already visited and a pending queue which contains vertices going to be processed. Our algorithm has a third queue named "have missing requirements" which contains models with inaccessible dependencies. If the visited queue receives a model which has a needed dependency in it's context, these two models are merged and placed in the pending queue.

The pending queue is sorted so the models with a shortest path have a highest priority. As metamodel edges are unweighted, the found path is minimal.

As we mentioned above, the next edge in the path does not necessarily starts from the end vertex of the previous edge. In this case a following step is applied. For each vertex in the context an additional model is generated which has a current vertex set to a fetched one. All generated models are placed in the pending queue. This step allows to fallback to any entity available in the context. In some cases, a number of generated models may be huge so this step is applied only in exceptional cases, for example, if there is no other solution available.

Summing up, results of this step are:

- the abstract syntax tree (AST) node which should be replaced;
- the edge from the source library matching the replaced node;
- the sequence of edges from the destination library's model which forms the replacement.

The fourth step is the **mapping of a new trace back into AST nodes**. This task can be easily done using information from the library model.

The fifth step is the **program transformation**. A set of AST nodes received from the previous step are placed in the syntax tree.

5. Tool Development

To demonstrate the feasibility of automated code migration and test the metamodel, the proof-of-concept tool for program migration was developed[2].

In this work, we have decided to use Java programming language for analyzing and processing. The main reason is an absence of "hard to analyze" constructions like macros, class templates, operator overloading, etc. Processing of these construction does not make sense for the proof-of-concept tool.

We have reviewed several approaches to fetch a program trace:

- using JVM application programming interfaces (APIs) to instrument program execution;
- interacting with existing Java debuggers;
- using aspect-oriented programming (AOP) to weave instrumentation code in the program.

The developed tool uses aspect-oriented programming because there are several widespread and well-tested AOP implementations which allow to instrument code with minimal efforts [12]. We have chosen *AspectJ* implementation as the most popular one.

We use *JavaParser* library to parse Java source code. It provides a high-performance and easy to use tool which is able to transform Java code into the AST model. It can also transform AST back into a code, preserving comments and formatting, which is very helpful for transformation tasks. Preserving formatting is necessary if the source is managed by a version control system like Git.

The developed tool relies on a custom domain-specific language (DSL) for library model description. The DSL is based on *Kotlin* programming language [13] and allows to write models without a deep knowledge of tool architecture.

[2]Source code of the proof-of-concept migration tool is available at https://github.com/h31/LibraryMigration

The developed tool contains the module which allows to visualize library models. It helps a lot to debug models and to communicate with persons who have an expertise in a library but does not know much about DSL.

6. Evaluation

First, it was necessary to choose a set of libraries all of which solve the same problem. Our choice was HTTP client implementations. We made models for the following libraries:

- *Apache HttpClient* from *HttpComponents* project;
- Java Class Library built-in client (*java.net.HttpURLConnection* and related classes);
- *OkHttp* client.

All of these libraries have a set of common entities (URL address, HTTP header, response content) and a set of library-specific entities. For example, Java built-in client has the class named *HttpURLConnection* which combines both the request and the response. *Apache HttpClient* and *OkHttp* contain the dedicated request and response classes, through slightly different. Also, these libraries have the Client object which should be instanced to make an HTTP request. Java built-in client does not have such a class.

After the first pilot version of tool was finished, we have tested it on the set of simple artificial examples each less than one hundred lines of source code. All of them (after some debug) were successfully migrated to new libraries. We tested reverse migration and it was successful too.

To prove feasibility of proposed approach we decide to evaluate the developed tool on real-world project. One of the important requirements for an evaluated project was an availability of test cases which are necessary for dynamic trace extraction. We have chosen a project named *instagram-java-scraper* which uses *OkHttp* client. This project was successfully migrated too. All project tests were passing and the code review had shown that migrated code is correct.

The migration tool has an automatic test suite which checks the correctness of the migration. The test suite also performs the reverse migration tests.

7. Conclusion

In this paper, we present the results of our research in the area of a software project migration. We have developed the migration tool which uses formal library specifications for automated porting Java programs to usage of new library. We have evaluated our tool on set of artificially constructed programs and the real-world open source project. All of artificial

programs and the real-world one were successfully migrated. Based on this we conclude that our approach has the right to exist.

We are planning to improve our approach and migration tool. The key directions of future research are:

- refinement of library specification formalism;
- development and extension of library model specification language;
- increasing possibilities of user control on the migration process;
- development of a more reliable and feature-rich migration tool.

References

1. Itsykson V.M. The Formalism and Language Tools for Semantics Specification of Software Libraries // Modeling and Analysis of Information Systems. — 2016. — Vol. 23, no. 6. — P. 754–766. — (In Russ.).
2. Marosi A. C., Balaton Z., Kacsuk P. GenWrapper: A generic wrapper for running legacy applications on desktop grids // 2009 IEEE International Symposium on Parallel Distributed Processing. — 2009. — May. — P. 1–6.
3. Google. A conformant OpenGL ES implementation for Windows, Mac and Linux. — 2017. — URL: https://github.com/google/angle (online; accessed: 18.05.2017).
4. QOS.ch. Simple Logging Facade for Java (SLF4J). — 2017. — URL: https://www.slf4j.org/ (online; accessed: 18.05.2017).
5. Jemerov Dmitry. Implementing refactorings in IntelliJ IDEA // Proceedings of the 2nd Workshop on Refactoring Tools / ACM. — 2008. — P. 13.
6. Cordy James R. The TXL source transformation language // Science of Computer Programming. — 2006. — Vol. 61, no. 3. — P. 190–210.
7. Stratego/XT 0.17. A language and toolset for program transformation / Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser // Science of computer programming. — 2008. — Vol. 72, no. 1. — P. 52–70.
8. Baxter Ira D, Pidgeon Christopher, Mehlich Michael. DMS/spl reg: program transformations for practical scalable software evolution // Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on / IEEE. — 2004. — P. 625–634.
9. Broeksema Bertjan, Telea Alexandru. Visual support for porting large code bases // Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on / IEEE. — 2011. — P. 1–8.
10. Christoph Alexander, Müller Matthias M. GREAT: UML transformation tool for porting middleware applications // International Conference on the Unified Modeling Language / Springer. — 2003. — P. 18–30.
11. Transforming Code with Compositional Mappings for API-Library Switching / L. Wu, Q. Wu,

G. Liang et al. // 2015 IEEE 39th Annual Computer Software and Applications Conference. — Vol. 2. — 2015. — P. 316–325.

12. Filman Robert E, Havelund Klaus. Source-code instrumentation and quantification of events // Foundation of Aspect-Oriented Languages. — 2002. — P. 45–49.
13. JetBrains. Statically typed programming language for the JVM, Android and the browser. — 2017. — URL: https://kotlinlang.org/ (online; accessed: 18.05.2017).
14. Eisenbarth Thomas, Koschke Rainer, Vogel Gunther. Static trace extraction // Reverse Engineering, 2002. Proceedings. Ninth Working Conference on / IEEE. — 2002. — P. 128–137.

UDC 004.052.42, 004.432

Towards Static Type-checking for Jolie

Bogdan Mingela (Innopolis University)

Nikolay Troshkov (Innopolis University)

Manuel Mazzara (Innopolis University)

Larisa Safina (Innopolis University, University of Southern Denmark)

Alexander Tchitchigin (Typeable.io LLC)

Daniel de Carvalho (Innopolis University)

Static verification of source code correctness is a major milestone towards software reliability. The dynamic type system of the Jolie programming language, at the moment, allows avoidable run-time errors. A static type system for the language has been exhaustively and formally defined on paper, but still lacks an implementation. In this paper, we describe our steps toward a prototypical implementation of a static type checker for Jolie, which employs a technique based on a SMT solver.

Keywords: Jolie, type-checking, verification, microservices

1. Introduction

Static type checking is generally desirable for programming languages improving software quality, lowering the number of bugs and preventing avoidable errors. The idea is to allow compilers to identify as many issues as possible before actually run the program, and therefore avoid a vast number of trivial bugs, catching them at a very early stage. Despite the fact that, in the general case interesting properties of programs are undecidable [19], static type checking, within its limits, is an effective and well established technique of program verification. If a compiler can prove that a program is well-typed, then it does not need to perform dynamic safety checks, allowing the resulting compiled binary to run faster.

Jolie [15] is the only language natively supporting microservice architectures and, currently, has dynamic type checking only. A static type system for the language has been exhaustively and formally defined on paper, but still lacks an implementation. The obstacles of programming in a language without a static type analyzer have been witnessed by Jolie developers, especially by newcomers. However, implementing such system is a non trivial task due to technical

challenges both of general nature and specific to the language. In this paper, we introduce and describe ongoing work on a static type checker for the Jolie programming language [15]. Our approach follows the formal specification rules as defined in [17]. The project is built as a Java implementation of source code processing and verification via Z3 SMT solver [8] and it has to be intended as our community contribution to the Jolie programming language [5].

Section 2 recalls the basic of Jolie and section 3 discusses related work. The description of the static type-checking and the system architecture can be found in Section 4, while Section 5 draws conclusive remarks and discusses open issues.

2. Background

Microservices [9] is an architectural style evolved from Service-Oriented Architectures [11]. According to this approach, applications are composed by small independent building blocks that communicate via message passing. These composing parts are indeed called microservices. This paradigm has seen a dramatic growth in popularity in recent years [16]. Microservices are not limited to a specific technology. Systems can be built using a wide range of technologies and still fit the approach. In this paper, however, we support the idea that a paradigm-based language would bring benefit to development in terms of simplicity and development cost.

Jolie is the first programming language constructed above the paradigm of microservices: each component is autonomous service that can be deployed separately and operated by running in parallel processes. Jolie comprises formally-specified semantics, inspired by process calculi such as CCS [13] and the π-calculus [14]. As for practical side, Jolie is inspired by standards for Service-Oriented Computing such as WS-BPEL [2]. The composition of both theoretical and practical aspects allows Jolie to be the preferred candidate for the application of modern research methodologies, e.g. runtime adaptation, process-aware web applications, or correctness-by-construction of concurrent software.

The basic abstraction unit of Jolie is the microservice [9]. It is based on a recursive model where every microservice can be easily reused and composed for obtaining, in turn, other microservices. Such approach allows distributed architecture and guarantees simple management of all components, which reduces maintenance and development effort. Microservices communicate and work together by sending messages to each other. In Jolie, messages are represented in tree structure. A variable in Jolie is a path in a data tree and the type of a data tree is a tree itself. Equality of types must therefore be handled with that in mind. Variables are not

declared, wherefore the manipulation of the program state must be inferred. Communications are type checked at runtime, when messages are sent or received. Type checking of incoming messages is especially relevant, since it could moderate the consequences of errors.

The Jolie language is constructed in three layers: The behavioural layer operates with the internal actions of a process and the communication it performs seen from the process point of view, the service layer deals with the underlying architectural instructions and the network layer deals with connecting communicating services.

Other workflow languages are capable of expressing orchestration of (micro)services the same way Jolie can do, for example WS-BPEL [2]. WS-BPEL allows developers to describe workflows of services and other communication aspects (such as ports and interfaces), and it has been also shown how dynamic workflow reconfiguration can be expressed [12]. However, WS-BPEL has been designed for high-level orchestration, while programming the internal logic of a single micro-service requires fine-grained procedural constructs. Here it is were Jolie works better.

3. Related work

The implementation of a static type checker for Jolie is part of a broader attempt to enhance the language for practical use. Previous work on the type system has been done, however focusing mostly on dynamic type checking. Safina extended the dynamic type system as described in [21], where type choices have been added in order to move computation from a process-driven to a data-driven approach.

The idea to integrate dynamic and static type checking with the introduction of refinement types, verified via SMT solver, has been explored in [22]. The integration of the two approaches allows a scenario where the static verification of internal services and the dynamic verification of (potentially malicious) external services cooperates in order to reduce testing effort and enhancing security.

The idea of using SMT Solvers for static analysis, in particular in combination with other techniques, has been successfully adopted before for other programming languages, for example LiquidHaskell and F*. LiquidHaskell [10][1] is a notable example of implementation of Liquid Types (*Logically Qualified Data Types*) [20]. It is a static verification technique combining *automated deduction* (SMT solvers), *model checking* (Predicate Abstraction), and *type systems* (Hindley-Milner inference). Liquid Types have been implemented for several other program-

[1]Online demo at http://goto. ucsd.edu/ rjhala/liquid/haskell/demo/

ming languages. The original paper presented an OCaml implementation. F* [1] instead an ML-like functional programming language specifically designed for program verification.The F* type-checker uses a combination of SMT solving and manual proofs to guarantee correctness

Another direction in developing static type checking for Jolie is creating the verified type checker [3] by means of proof assistant instead of SMT solver. Proof assistant is a software tool needed to assist with the development of formal proofs by human-machine collaboration and helps to ascertain the correctness of them. The type checker is expressed as well-typed program with dependent types in Agda [18]. If the types are well formed, all required invariants and properties are described and expressed in the types of the program meaning that the program is correct. This work is currently in progress and evolves in parallel with ours.

4. Static type-checking implementation

This paper builds on top of Julie Meinicke Nielsen's work at the Technical University of Denmark (*"A Type System for the Jolie Language"* [17]) implementing the type checker specification. The thesis represents the theoretical foundation for the type checking of the core fragment of the Jolie language, which excludes recursive types, arrays, subtyping of basic types, faults and deployment instructions such as architectural primitives. The work of Nielsen presents the first attempt at formalizing a static type checker for the core fragment of Jolie, and the typing rules expressed there are the core theory behind our static checker.

The implementation of the type checker consists of two system components. Firstly, a Java program accepts the source code of a Jolie program, builds an abstract syntax tree (AST), visits it and produces a set of logical theorems written in Z3 [8]. At the second phase, the generated theorems feed a Z3 solver of which they represent the input. The basic idea is to implement, for each Jolie node[2], methods containing statements expressed in the SMT-LIB [6] syntax. These statements can then be processed via a solver. In Figure 1 the overall process is pictorially represented.

The concept of SMT solvers is closely related to logical theorems. Logic, especially in the field of proof theory, considers theorems as statements of a formal language. Existence of such logical expressions allows to formulate a set of axioms and inference rules to formalize the typing rules for each of Jolie nodes and then perform the validation of the nodes using constructed theorems. Consequently, the Jolie typing rules are the specific cases of logical theorems, that

[2]Any syntax unit is considered a node. It can be a logical or arithmetic expression, an assignment; a condition; a loop etc. Those nodes comprise the abstract syntax tree.

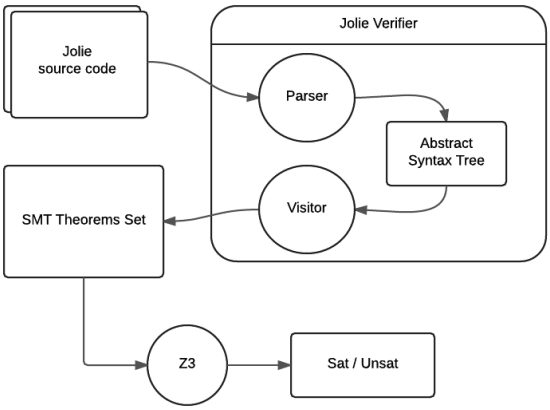

Fig. 1. Process of Type Checking in the Jolie Verifier

are used in the project. The concept is implied from software verification fundamentals [7].

Since Jolie program may contain complex expressions with function calls, it is also necessary to consider data structures representing a match between names and expressions, in order to be able to avoid inconsistency and redundancy, that are likely to cause conflicts during type-checking. The project implementation considers using a stack during the recursive checking of the nodes during a traversal mostly so far.

4.1. Jolie verifier

The Java program reuses an existing structure of a Visitor pattern that was used in a previous project for formatting Jolie source code [4]. It accepts processed Jolie program source code in the form of AST and performs traversing. For each kind of node the system creates one or more logical formulas written using SMT-LIB [6] syntax, which are then stored into a file on disk. At the current implementation state the theorems are collected in a single data element.The verifier targets assignments, conditions, and other cases of variables usage where type consistency can be violated.

4.2. SMT Solver

Z3 carries out the main functionality of program verification. Z3 is an SMT solver from Microsoft Research [8]. It is targeted at solving problems that arise in software verification and software analysis. Given a set of formulas that was previously created by the verifier in Java, Z3 processes it and returns whether this set is satisfiable or not. In case of any contradiction in the set, the solver will signal that the overall theorem is not satisfiable, therefore alerting that

the input program is not consistent in terms of types usage.

4.3. Typing rules to SMT translation

Our objective is to accurately translate Jolie typing rules into SMT statements, therefore allowing static type checking [3]. The foreground activity so far is producing the set of statements for the construct of the behavioural layer of Jolie. The layer describes the internal actions of a process and the communications it performs seen from the processвЂ™ point of view. The layer is chosen for the first phase of the development because of being the foundation of the syntactical structures of Jolie. Also there is a similarity of the layer with common programming languages in a sense of the abstraction level. So these facts make the behavioural level to be the first entry in the world of Jolie language capabilities.

Here we will illustrate an example of the translation in order to understand the procedure in detail. All statements at the behavioural layer of Jolie are called behaviours. We write $\Gamma \vdash_B B \triangleright \Gamma'$ to indicate a behaviour B, typed with respect to an environment Γ, which updates Γ to Γ' during type checking [17].

The conditional typing statement is the following:

$$\frac{\Gamma \vdash e : bool \quad \Gamma \vdash_B B_1 \triangleright \Gamma' \quad \Gamma \vdash_B B_2 \triangleright \Gamma'}{\Gamma \vdash_B if(e)\ B_1\ else\ B_2 \triangleright \Gamma'}$$

The typing rule of the *if* statement does not contradict intuition. The statement is typeable when its condition expression is boolean, and the execution of both its branches brings the same updates to the environment. This means that the set of matches between expressions and variables with their types remains the same with no difference from a branch choice. This is necessary since it is not possible to predict what branch will be executed at runtime. [4]

The full implementation is available on github [5]. Here we show the Java fragment which builds the corresponding SMT statement.

```
1 public void visit(IfStatement n) {
2         for (Pair<OLSyntaxNode, OLSyntaxNode> statement :
                      n.children()) {
3             OLSyntaxNode condition = statement.key();
4             OLSyntaxNode body = statement.value();
5             check(condition);
```

[3] Please note that, at the moment, not all the rules in [17] have been implemented.

[4] The *else* part may also be omitted and B_2 may be replaced by an empty behavior.

[5] https://github.com/innopolis-jolie-smt-typechecker/jolie

```
6               TermReference conditionTerm = usedTerms.pop();
7               writer.writeLine("(assert (hasType " +
                    conditionTerm.id + " bool))");
8               if (body != null) {
9                   body.accept(this);
10              }
11          }
12          if (n.elseProcess() != null) {
13              n.elseProcess().accept(this);
14          }
15      }
```

The code structure represents basic steps to achieve a record with corresponding SMT statements of the block as a result. Firstly, a condition of the *if* statement is separated from the body. Then the condition is sent to be checked using the same visitor class. Eventually after the last 'recursion' step the condition is put in the stack of terms, which contains any terms (expressions, variables etc.) processed during the checking. So the term corresponding to the condition is expected to be on top of the stack. Then an assertion that says the condition term is boolean is written. Afterwards the body is processed using one of the other overloads of the visitor. These steps can be repeated in case of existence of nested conditional statements. In the end of the method the *else* branch body of the very first *if* is processed if it is present. There is also an important note is that the conditional statement does not impose any other direct type restrictions besides the condition term that is confirmed by the mentioned typing rule. Other implemented nodes can be seen in the source mentioned above.

The Jolie verifier takes some input for processing. Let us consider the following simple piece of Jolie code with a conditional statement:

```
1 a = 2;
2 b = 3;
3 if ( a > b ) {
4  println@Console( a + b )()
5 }else{
6  println@Console( "Hello, world!" )()
```

```
7 }
```

In the case everything works, none of the typing rules is violated. Z3 agrees with the opinion and results in 'sat', that means the program state is satisfiable. There is the SMT statements representing the condition processing:

```
1 (declare-const $$__term_id_10 Term)
2 (assert (hasType $$__term_id_10 bool))
3
4 (assert (hasType $$__term_id_10 bool))
```

The first assertion is made based on an expression type determination: the expression $a > b$ is boolean. The second one is imposed by the typing rule: the condition expression must be boolean. In the case there is no contradiction between these two assertions.

If the condition would be replaced with some other type expression the typing rule may be violated. There is the corresponding example case with a replacement of $a > b$:

```
1 a = 2;
2 b = 3;
3 if ( 5 ) {
4   println@Console( a + b )()
5 }else{
6   println@Console( "Hello, world!" )()
7 }
```

And the constructed SMT statements for the condition expression are following:

```
1 (declare-const $$__term_id_10 Term)
2 (assert (hasType $$__term_id_10 int))
3
4 (assert (hasType $$__term_id_10 bool))
```

Now the contradiction between the assertions is notable. The parser decided the expression to be an integer, which is correct. But the restriction on a condition type from the typing rule simply contradict with the actual type. Consequently Z3 results in 'unsat'. This means that the program state representing the assertion unsatisfiable and incorrect in terms of the considered static type checking analysis.

5. Conclusions and future works

Static type checking allows compilers to identify programming mistakes (for what concerns types) at compile time, i.e. before actually running the program. Therefore a vast number of trivial bugs can be avoided being caught and fixed at a very early stage of the software life-cycle. In this paper, we tackled the problem of static type checking for the Jolie programming language, which natively supports microservice. A static type system for the language has been exhaustively and formally defined on paper, but so far still lacked a full implementation. We introduced our ongoing work on a static type checker and presented some details of the implementation. The type checker prototype, at the moment, consists in a set of rules for the type system expressed in Z3. The actual implementation covers operations such as assignments,logical statements, conditions, primitive terms usage and comparison.

The type checker is already able to validate programs, as it has been shown in this paper. However, it works with certain assumptions. The main assumption is that programs do not contain implicit type casts. The Jolie language allows implicit type casts, however, their behavior is very complex. Handling such situations is an open issue left for future development and future versions. Two other major issues have not been addressed.

Variable types can be changed at runtime. This strictly depends on the approach that has been chosen. Generally, static typing guarantees that a variable has a type that cannot be changed after declaration or assignment. However, Jolie allows this operation. We need to determine which behavior we expect from the type checker, thus deciding how to process type changes.

Implicit type casts in Jolie are ambiguous. This is a major problem, and further research is required in order to find a solution. While Jolie allows implicit type casts, sometimes the result of a cast is not obvious. For example, casting a negative Integer to Boolean will result in a False. This is an unexpected behavior when compared to other programming languages. There may be a solid rationale for this, however, we need to investigate all cases and make sure that the type checker works accordingly to the Jolie actual behavior, and not to the expected one.

References

1. F*. https://www.fstar-lang.org/.
2. WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.
3. Eugenii Akentev. Verified type checker for Jolie programming language. https://github.com/ak3n/jolie.
4. Nikita Alekseev. Jolie Code Formatter. https://github.com/nickaleks/jolie.
5. Alexey Bandura, Nikita Kurilenko, Manuel Mazzara, Victor Rivera, Larisa Safina, and Alexander Tchitchigin. Jolie community on the rise. In *9th IEEE International Conference on Service-Oriented Computing and Applications, SOCA*, 2016.
6. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard. Version 2.0 , 2010.
7. Hoare C.A.R. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
8. Leonardo de Moura and Nikolaj BjГërner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
9. Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In Bertrand Meyer and Manuel Mazzara, editors, *Present and Ulterior Software Engineering*. Springer, 2017.
10. Ranjit Jhala. Liquid Haskell. http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/about/.
11. M.C. MacKenzie et al. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006.
12. M. Mazzara, F. Abouzaid, N. Dragoni, and A. Bhattacharyya. Design, modelling and analysis of a workflow reconfiguration. In *Proceedings of the International Workshop on Petri Nets and Software Engineering, Newcastle upon Tyne, UK, June 20-21, 2011*, pages 10–24, 2011.
13. Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., 1995.
14. Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.
15. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
16. S. Newman. *Building microservices*. O'Reilly Media, Inc., 2015.
17. Julie Meinicke Nielsen. A Type System for the Jolie Language. Master's thesis, Technical University of Denmark, 2013.
18. Chalmers University of Technology. Accessed December 2016. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php.
19. Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Trans.*

Amer. Math. Soc., 74:358–366, 1953.

20. Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.
21. Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.
22. Alexander Tchitchigin, Larisa Safina, Manuel Mazzara, Mohamed Elwakil, Fabrizio Montesi, and Victor Rivera. Refinement types in jolie. In *Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE*, 2016.