UDK 004

Static Memory Consistency Constraints Checking

Andrianov P.S. (Ivannikov Institute for System Programming of RAS)

Mutilin V.S. (Ivannikov Institute for System Programming of RAS)

Memory model describes the memory consistency requirements in a multithreading system. Compiler optimizations may violate the consistency requirements due to bugs, and the program behavior will differ from the required one. The bugs in compiler optimizations, like incorrect instruction reordering, are very difficult to detect, because they may occur with a very low chance in real execution on a hardware.

There are different approaches of formal verification for memory consistency requirements, but the challenge is that the approaches are not scalable for industrial software. In the paper we present the MCC tool that was evaluated on the industrial virtual machine ARK VM and was able to find a real bug in a compiler optimization.

The MCC is a static tool, which allows to check all possible executions of a particular test, not relying on a hardware execution. The approach also includes test suite generation and specification of memory consistency properties.

Keywords: static analysis, memory model, compiler optimizations

1. Introduction

Nowadays processors have one or more layers of cache memory, which improves performance. However, their use also throws up new challenges. *Hardware memory model* defines necessary and sufficient conditions to guarantee that memory writes by other processors will be visible to the current processor, and that the current processor's writes will be visible to other processors. There exists a strong memory model, which guarantees that all processors always see exactly the same values for any given memory location. An opposite case is a weaker memory model.

There is different software, which can be executed over different hardware architectures. So, languages, like C++ or Java, specify its own memory models, which describe software behaviour independently from hardware. Java Memory Model (JMM) [1] provides strong guarantees, for example, if Java program does not have data races the programmer may assume sequentially consistent behavior.

ARK VM[1] is a general virtual machine, which is a part of a unified operating system Har-

[1]https://gitee.com/openharmony-sig/arkcompiler_runtime_core.git

Shared variable: *var.x*; local variables: *sum*, *r*

```
...                         ...
                            r = var.x
loop:                       loop:
  wait()                      wait()
  sum += var.x                sum += r
...                         ...
```

Before optimization   After optimization

Fig. 1. Example of a program in pseudocode before and after optimization

monyOS [2]. Like the other virtual machines ARK VM supports running multithreaded programs. Such programs consist of threads concurrently executing VM instructions, e.g. writing to or reading from the memory. Some instructions have a special memory consistency semantics providing guarantees for a programmer when the other threads will see memory updates. For example, for Java there are strong guarantees that threads will see the results of writing to and reading from volatile variables in a sequentially consistent order. This memory consistency guarantees are specified in JMM. ARK VM has its own memory model (MM).

ARK VM should satisfy the requirements of MM, i.e. both ARK interpreter and ARK compiler. In our method we focused on the latter, i.e. verifying that the ARK compiler fulfills memory consistency requirements. More specifically we are interested in correctness of optimization transformations performed in Just-in-Time (JIT)/ Ahead-of-Time (AoT) of the ARK VM compiler. The method checks that ARK compiler optimizations generate code in target hardware instruction set architecture (HW ISA) which does not break memory consistency requirements of the input program.

The main contribution is an approach to detect memory consistency violations in the industrial software. The approach is more general and may be applied to other VMs or compilers.

2. Motivating example

Consider the following example (in pseudocode) in Fig. 1.

Here in the loop there are two actions: *wait* and access to a shared memory *var.x*. A compiler optimization transforms the code and extracts the read of the shared variable from the loop to the local variable r. The transformation modifies the program logic. Imagine, for example, a second thread, which updates *var.x* and then wakes the initial thread. Then the

[2]https://www.harmonyos.com/en/

code after optimization will be able to read shared *var.x* before wake, but it is not possible before optimization.

The task of proving that the left program is not equivalent to the right, is very complicated, especially due to the presence of a loop. Also, dynamic tools likely may not reproduce the problem, because it requires both a specific interleaving sequence between threads and specific cache updates in a hardware. Thus, we have a task to develop an approach to detect such kind of bugs.

3. Method Overview

The MCC approach breaks into three major parts:

- MCC-props - specifications of consistency properties;
- MCC-testgen - a test generator (OTK tool [2]);
- MCC-core - a core verification engine.

The main workflow is presented in Fig. 2. One has to specify constraints, or invariants, which should be hold on the all program executions. MCC-testgen provides a test suite for a specified property. When the test is compiled, the compiler optimizations are triggered. MCC-core verifies, that the specified property holds on each compiled test (in native code).

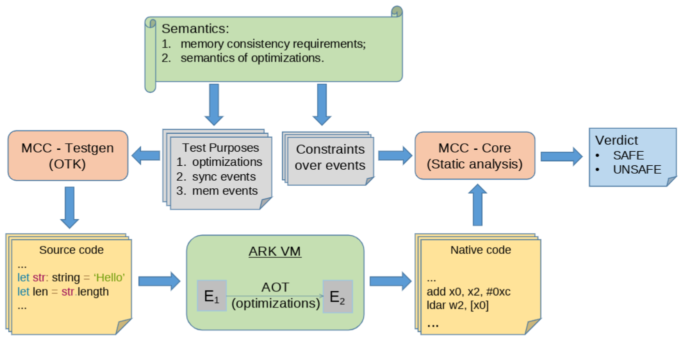

Fig. 2. MCC approach overview.

4. MCC properties

As it was already written, in the approach we have two representations of program: before compiler optimizations and after it. The first one is in the form of source code with the corresponding VM MM. The second one is in the form of native code with corresponding HW MM.

On each representation we can define a set of possible executions including sequence of events defined by the program order and the relation between written and read values, i.e. which writes are seen by which reads. In general, we want to check that any execution in the native code should be possible in the source code.

MM defines a set of synchronization events which significantly restricts the relation between reads and writes. In many cases MM forbids to transfer reads and writes through synchronization events. In our work we rely on this fact and specify our properties in terms of the sequence of events along the execution.

In general, we formulate our properties in terms of sequence of events with modalities referring to time, similar to Linear Temporal Logic, but for the sake of effectiveness we introduced a simple language in the form of source code model comments presented later. The language is used to specify MM properties of ARK VM, namely *ordering* and *existence* specifications.

4.1. Ordering specification

There are different kinds of events in program: synchronization events (*sync*), non-synchronization events (*unsync*) and others. Synchronization events used for synchronization between threads, they provide some guarantees for user, for example, access to Java volatile variable or acquire of a lock. Thus, synchronization events cannot be reordered. Non-synchronization events are accesses to the shared data, they depends from synchronization events, but may not depend to each other. Thus, non-synchronization events can be reordered with each other, but cannot be reordered with synchronization events.

So, this is ordering specification:

- synchronization events cannot be reordered;
- non-synchronization events cannot be reordered with synchronization events.

As it was already written, operations with volatile variables are an example of synchronization events in JMM. In ARK VM there are intrinsic calls. The intrinsics have different flags, related to optimizations. And a specific set of flags means, that the intrinsic call cannot be optimized and can be expressed as a synchronization event. Reads and writes of a memory cell are non-synchronization events. For ARK VM they are expressed as load/store into memory. Thus, we have a set of non-synchronization events, each of them can not be reordered through any synchronization event.

An example of specification is presented in Fig. 3. It contains a call of *wait* - a *sync* event and a read of shared data - an *unsync* event.

```
...
loop:
  // MODEL: sync
  wait()
  // MODEL: unsync
  sum += var.x
...
```

Fig. 3. Example of a program with a specification inside

So, intrinsics with the flags are synchronization event, and can not be optimized. In the opposite case, it is possible to optimize and even remove the call.

4.2. Existence specification

One more generic property is that synchronization event cannot be totally removed by optimization. Actually, this is very strong condition and dead code optimization can remove the synchronization events. However, the check is rather important and in our experiments we found a bug with missing flag using this property.

5. MCC test generator

MCC-testgen is built upon a method for test generation using extended ISA model (OTK) [2]. We extend this method by supporting *model of situations* with data types related to MCC-props. We base on the *model of situations* which are being developed for generating tests covering logic of optimizations in ARK compiler. We extend them with situations related to MCC-props. For example, MCC-testgen *model of situations* introduces synchronization data types which are combined with general models. Thus, MCC-testgen generates tests for covering both general logic of optimization and a memory consistency property.

Currently, MCC-testgen provides *model of situations* for five optimizations: loop-invariant code motion (LICM), LICM for conditions (LICMCond), loop peeling (LP), redundant loop elimination (RLE), loop unrolling (LU). It means, that MCC-testgen generates tests specifically to trigger the corresponding optimizations. Then, it combines the test with memory actions.

Every test is a sequence of actions, which trigger the optimization. For example, it may have loops with different number of iterations, depth and so on. Between the actions, related to optimization, there are actions, related to memory model events.

Every memory action consists from three parts: pre-action, target action and post-action. Target action may be a write to a local register, a write to a shared object or an empty action. Pre-action and post-action can contain sync events, read/write to a shared object or an empty action. There may be several memory action triples in a single test.

The testgen randomly iterates different kinds of memory actions and generate a specified number of tests. For each test generated with the MCC-testgen the MCC-core verifies that ARK compiler optimizations satisfy memory consistency requirements stated in MCC-props.

6. MCC core

MCC-core is a verifier of MCC properties on a VM program. It may be an any program, but we used tests, provided by MCC-testgen. Actually, MCC needs not only an HW assembly program with a specification, which operations are sync and unsync events. The relation is extracted from a compiler dump with debug information.

The MCC-props specification is written inside a VM program using model comments. We just need to mark synchronization and non-synchronization events inside source code of a test. Then MCC-core checks that every specification constraint in terms of VM instructions is fulfilled in the HW program. For every model event in terms of VM instructions, MCC-core finds its HW assembler representation. And then it checks that the specified relation is the same. For example, assembler instructions for event A are before assembler instructions for event B.

MCC-core result is a verdict saying, whether it has detected a scenario of execution of HW variant which does not satisfy a specification property.

MCC-core takes a VM program (or directory) and configuration as input. Internally, it performs the following steps:

- *Model extraction* – extracts a model of events: synchronization and non-synchronization with relation to the origin source code.
- *Preparation* – obtains an assembly dump from the compiler.
- *Assembler parser* – obtains a relation of assembler instructions to the origin source code, that is performed using the compiler dump information.
- *Constraint checker* – checks, that every synchronization event is not reordered or removed in the assembler code.

Model extraction is performed just by model comments: if a line is marked as *sync*, it is considered as a synchronization event. And *unsync* means non-synchronization event. There is also a possibility to specify constraints directly, for example, $A > B$, meaning the event A must be after B. In general case the model comments should be set manually. However, as we use a test generator, it performs all the work. So, the model comments are a part of OTK test templates.

Preparation stage takes a source code, produces a bytecode file and then runs compiler to get a compiler dump in Intermediate Representation (IR). A compiler dump file contains different information, but we use it to extract relation from IR assembler operations to origin source lines.

To extract the relation, we search pattern:

$$< IRoperation > ...(< filename >:< sourceline >)$$

However, not all IR operations has the information, thus it may be not complete. As source lines are marked with *sync* and *unsync* events, using the relation, we may obtain the same information about the assembler operations.

The constraint checker is implemented as a model checker based on Configurable Program Analysis (CPA) [3, 4]. We implemented *Sync CPA* which checks MCC ordering and existence properties. Its limitation still is false alarms due to dead code.

7. Discussion on method completeness

MCC approach performs static checks, and allows to consider all executions (see soundness of CPA approach). So, for a particular test and a particular property the MCC approach is able to be sound. For dynamic methods executing HW assembly on the processor the most tricky part is reproducing bugs, because they have a low probability and require rare cache coherence protocol states to occur.

The sources of incompleteness come from the techniques used in the components.

First, since we are basing on MCC-testgen and rely on the concrete set of generated tests. The tests reach some coverage of *model of situations* and we can measure progress related to the consistency model. However, we are dealing with tests, hence 100% coverage does not give 100% guarantee of covering all situations in the implementation.

Second, the set MCC properties elaborated from a memory model may not be complete. Here we rely on our expert knowledge.

8. Evaluation

To evaluate the MCC tool we used a machine with Ubuntu 22.04, Intel Core i5-6600K CPU @ 3.50GHz × 4 and 32 Gb RAM. We generated 100 tests for the LICM optimization and then checked MCC properties with MCC core. We run the tests on the origin ARK VM and on the ARK VM with bug introduced. The bug is related to incorrect intrinsic flag, which potentially may enable a compiler optimization in forbidden cases. The results are presented in Table 1.

Stage	Wall Time	CPU Time	Passed	Failed
Test generation	8 m 14 s	21 m 31 s	-	-
Normal run				
MCC-core	45 s	45 s	100	0
A bug introduced				
MCC-core	42 s	43 s	86	14

Table 1

Evaluation

Current version of the test suite for one LICM optimization achieves about 42% of line coverage in *compiler* directory. Anyway, we are mostly interested in coverage of target code, which works with memory events. Usually the target code is represented as conditions checking the properties of instructions, i.e. whether it is a sync event or not. It not a big number in line of code, but important for memory consistency. We manually inspected the collected coverage and found that we covered such conditions.

With the MCC tool there was found a real bug. It has been existed for a long time in optimization of intrinsic calls[3]. Some operations can be reordered with the call, which may lead to unexpected behaviours.

9. Related work

For checking JMM requirements there exists benchmark sets, like *JCStress*[4], which targets on concurrency bugs, including memory model violations. The benchmark set allows to reproduce MM bugs indeed, however, the chance is very low. For example, an introduced bug in volatile processing was fixed after several months[5], and test failure rate was about 0,0001%.

[3]https://gitee.com/openharmony-sig/arkcompiler_runtime_core/pulls/1389

[4]https://github.com/openjdk/jcstress

[5]https://gitee.com/openharmony-sig/arkcompiler_runtime_core/pulls/993

One more approach for memory consistency verification is formal prove with Alloy [5]. The approach requires to describe memory models of language and hardware, and also a mapping from one to another (a model of compiler). Then, it is possible to prove with Alloy model checker, if there is no forbidden execution found after “compilation”. The main problem of the approach is performance, as with all formal methods. Authors used a rather powerful machine with four 16-core 2.1 GHz AMD Opteron processors and 128 GB of RAM. However, proving that compilation from C11 to x86 takes about 3 hours for an execution with 5 synchronization events. Need to say, that comparison of memory models (without “compilation” task) takes less time, especially, if a counterexample was found.

DoItYourself tool [6] allows to generate litmus tests for a specific hardware memory model. Power and x86 are supported. Memory model specification is written on Coq, and then the tool generates the test with a potential forbidden execution. If the forbidden execution is observed on a real test run, it means, that the hardware does not fulfill its specification. The approach suits for checking hardware memory model in case we have a complete memory model specification. However, it does not consider compilation level and, moreover, compiler optimizations. Thus, it solves another task.

10. Conclusion

We presented an approach for practical detection of memory consistency violations. It was applied to an industrial virtual machine and found a complicated bug in compiler optimization.

References

1. J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2013) Java memory model, §17.4. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se19/html/jls-17.html
2. S. Zelenov and S. Zelenova, “Model-based testing of optimizing compilers,” in *Testing of Software and Communicating Systems*, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 365–377.
3. D. Beyer, T. A. Henzinger, and G. Théoduloz, “Configurable software verification: concretizing the convergence of model checking and program analysis,” in *Proceedings of CAV*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 504–518.
4. D. Beyer and M. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, 2011, vol. 6806, pp. 184–190.

5. J. Wickerson, M. Batty, T. Sorensen, and G. Constantinides, “Automatically comparing memory consistency models,” 01 2017, pp. 190–204.
6. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “Fences in weak memory models,” in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272.

UDK 004.415.52

Requirements patterns in deductive verification of process-oriented programs and examples of their use

Chernenko I. M. (Institute of Automation and Electrometry SB RAS)

Process-oriented programming is a promising approach to the development of control software. Control software often has high reliability requirements. Formal verification methods, in particular deductive verification, are used to prove the correctness of such programs regarding the requirements. Previously, a temporal requirements language [2] was developed to specify temporal requirements for deductive verification of process-oriented programs. It was also shown that a significant part of the requirements falls into a small number of classes. Requirements patterns was developed for these classes. In this paper, we present a collection of process-oriented programs and requirements for them. Requirements are formalized in the temporal requirements language and classified according the set of patterns. We also define a new requirement pattern. These results can be used in the research of formal verification methods for process-oriented programs, in particular in the research of methods of proving verification conditions.

Keywords: deductive verification, temporal requirements, control software, process-oriented programs

1. Introduction

Formal verification is a crucial part in the development of safety-critical software, in particular, control software. Deductive verification is one of formal verification methods in which program properties are formalized in the form of logical formulas expressing relations between program variables (preconditions, postconditions and invariants) and added to the programs as annotations. Then for this annotated program, the generation and proving of verification conditions are performed.

Process-oriented programming [11] is one of approaches to control software development. A process-oriented program is defined as a sequence of interacting processes which is executed in the control loop. Each process is represented by a set of named executable codes called process states.

Control software often has temporal requirements. Previously, we developed a deductive verification approach [1] for process-oriented programs with temporal requirements. This approach entails utilizing control loop invariants and storing the history of changes in variable values to specify temporal requirements. In our work [2], we introduced the DV-TRL annotation language oriented to deductive verification for formalizing requirements to process-oriented programs. We also formalized a set of 45 requirements for 10 case studies and discovered that a considerable portion of these requirements can be classified into a few distinct classes. Based on these findings, we established four requirement patterns.

To verify temporal requirements, model checking is a commonly used method where temporal properties are specified using temporal logic formulas (e.g., LTL or CTL). Model checking tools then automatically check if the program model satisfies the specified requirements. However, one limitation of model checking is the state explosion problem. Hence, deductive verification is also employed to verify temporal requirements.

Deductive verification is typically used to verify functional requirements [5, 6]. In deductive verification, requirements are described by assertions at certain points in the program linking the current values of variables at these points [4]. Nevertheless, deductive verification is also used to verify temporal requirements. For example, in STeP [8], verification rules are used to reduce the proof of correctness of a program in the SPL language with respect to a temporal formula to a set of verification conditions that are formulas of first-order logic. In [7] the deductive verification of control software with temporal properties specified using timing charts is presented. The Why3 system is used for verification. Events and stable states of the timing chart are represented in Why3 by loops, the body of which corresponds to the iteration of the control loop, and the guard specifies the stable state of the timing chart. In our work, we define temporal properties in the form of invariants of the control loop. Unlike STeP, we do not develop special verification rules for temporal formulas, but use the rules of axiomatic semantics, similar to the rules of Hoare logic.

Temporal specification of requirements plays a crucial role in the development of critical control software. However, this process is often burdensome and prone to errors. To address these challenges, research is being undertaken to simplify the temporal specification. For instance, in [9], the authors introduce a classification encompassing commonly encountered temporal requirements in specification tasks. Similarly, we have developed our own classification and established patterns. While our classification may be incomplete, it incorporates requirement classes identified in real control system specifications.

This paper aims to expand the set of temporal requirements and establish control programs in the process-oriented language poST [12]. Additionally, we enhance the collection of requirement patterns to align with the extended set of requirements.

The rest of the paper has the following structure. We describe the temporal requirements language in Section 2 and requirement classification in Section 3. In Section 4, we present the set of case studies.

This work is supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

2. Temporal Requirements Language DV-TRL

The temporal requirement language DV-TRL introduced in [2] is based on *state* date type storing the history of all changes of program variables and the set of specialized functions over this data type. The *state* data type is defined by the following set of constructors:

- $emptyState : state$ the initial empty state of the control system;
- $toEnv : state \to state$ inputs to the environment
- $setVar : state \times variable \times value \to state$ sets values to variables;
- $setPstate : setPstate : state \times process \times pstate \to state$ sets process states;
- $reset : state \times process \to state$ resets local clocks of processes.

A corresponding constructor is defined for each type of changes in programs.

Following domain-specific functions over the *state* data type are used in program annotations:

- $getVar : state \times variable \to value$ returns values of variables;
- $getPstate : state \times process \to pstate$ returns current states of processes;
- $substate : state \times state \to bool$ checks that the first state is a substate of the second state. A state s' is a substate of s if $s' = s$, or there exist a state s', a constructor c, and values v_1, ..., v_n such that $s' = c(s'', v_1, \dots, v_n)$, and s'' is a substate of s;
- $toEnvNum : state \times state \to nat$ returns a number of application of constructor *toEnv* for getting the second state from the first substate. The score starts anew if the reset constructor meets;
- $toEnvP : state \to bool$ checks whether the state has the form $toEnv(s)$ for some s;

These functions allow describing the requirements for process-oriented programs in a natural way.

Here the *value* type is a union of types *bool*, *nat* and *int*. Types *bool*, *nat* and *int* describe sets of logical constants (true and false), natural numbers and integers numbers, respectively.

The *variable*, *process*, and *pstate* types are used to encode the names of variables, processes, and process states of a process-oriented program respectively.

3. Requirements Patterns

In this section, we will be discussing 5 temporal requirements patterns. These patterns are formulated as parametric formulas in the DV-TRL language. The initial 4 patterns were previously introduced in [2]. We add one more pattern and present the formula only for it.

The first pattern establishes requirements that specify that event E_2 should occur no later than τ after event E_1. An example of this is the first requirement for the turnstile control program, which we will discuss further below.

The second pattern describes requirements that state that a specific event will occur after one iteration of the loop. An example of this is the first requirement for the thermopot control program, which we will presented on later.

The third pattern describes requirements that assert that events should occur at one specific point within the control loop. An example of this is the sixth requirement for the revolving door control program, which we will examine in more detail below.

The fourth pattern for requirements combines the first and second patterns. If event E_1 occurs, event E_2 must occur within a time interval of τ. In this scenario, the formula that describes E_1 imposes restrictions on the variable values in two different states, separated by one iteration of the control loop. An example of this class is the first requirement for the pedestrian crossing light control program that will be discussed later.

The fifth pattern describes requirements that state that if event E_1 occurs, event E_2 can only occur after a minimum time interval of τ. The formula that describes event E_1 imposes restrictions on variable values in two distinct states, with the time between them being one iteration of the control loop. The requirements pattern for this class can be expressed as follows:

E_1 has happened, then event E_2 does not happen for at least τ.

$$\begin{aligned}
&p_5(s, \tau, vc_1, vc_2) \equiv \\
&\text{toEnvP}\, s \wedge \\
&(\forall s_1 s_2 s_3.\, \text{substate}\, s_1\, s_2 \wedge \text{substate}\, s_2\, s_3 \wedge \text{substate}\, s_3\, s \wedge \text{toEnvP}\, s_1 \wedge \text{toEnvP}\, s_2 \wedge \\
&\text{toEnvP}\, s_3 \wedge \text{toEnvNum}\, s_1\, s_2 = 1 \wedge \text{toEnvNum}\, s_2\, s_3 < \tau \wedge vc_1(s_1, s_2) \longrightarrow vc_2(s_3),
\end{aligned}$$

where variable constraint vc_1 describes the event E_1, vc_2 describes the event E_2. This class includes, for example, the fourth requirement for the fridge control program considered below.

4. Case Studies

This section presents the set of case studies. We provide descriptions of the case studies and requirements for them as well as classify these requirements according to the patterns and provide examples of their formalization. The poST programs can be found on GitHub [13].

4.1. Turnstile

In this case study, we examine a turnstile as a controlled device. The turnstile is equipped with a coin acceptor, doors operated by a signal (open), an LED (enter) indicating the possibility of passage, and two sensors to detect the presence of a user (PdOut) and the opening of the doors (opened). The doors remain locked until a payment signal (paid) is received from the coin acceptor, after which they open. If the user does not pass through within 10 seconds after the doors open, they will automatically close. After a successful payment, the coin acceptor is locked. The coin acceptor is unlocked by a reset signal once the turnstile is closed.

For the turnstile control program, we propose the following 7 requirements:

1. The open signal should remain true for a maximum of 10 seconds.
2. If the turnstile has been closed and the payment has not been made, it will not open until the payment is made.
3. After receiving the paid signal from the coin acceptor, the open signal should be given immediately.
4. After receiving the PdOut signal indicating the passage of a user, the turnstile must be closed within a maximum of 1 second.
5. The open signal should remain true for a minimum duration of 1 second.
6. After the opened signal appears and until it is reset, the enter LED should be lit.
7. After the turnstile is closed, the signal reset should be given to unlock the coin acceptor.

The requirement 1 belongs to the 1st class. The requirements 2, 3 and 7 belong to the 2nd class. The requirement 4 belongs to the 4th class. The requirement 5 belong to the 5th class. The requirement 6 belong to the 3rd class. For example, the first requirement is formalized as the following annotation:

$$\text{toEnvP}\, s \wedge$$
$$(\forall s1.\, \text{substate}\, s1\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvNum}\, s1\, s \geq 100 \wedge \text{getVarBool}\, s1\, \mathit{open}' \longrightarrow$$

$(\exists s3.\, \text{toEnvP}\, s3 \wedge \text{substate}\, s1\, s3 \wedge \text{substate}\, s3\, s \wedge$
$\text{toEnvNum}\, s1\, s3 \leq 100 \wedge \neg\, \text{getVarBool}\, s3\, \mathit{open}' \wedge$
$(\forall s2.\, \text{toEnvP}\, s2 \wedge \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s3 \wedge s2 \neq s3 \longrightarrow \text{getVarBool}\, s2\, \mathit{open}')))$

4.2. Pedestrian Crossing Light

In this case study, a pedestrian crossing light with a button installed at the crossing is considered as a controlled device. Its regular state is "red". When a pedestrian appears at the crossing, he presses the button. If the green signal of the pedestrian crossing light was on more than 10 seconds ago, then the green signal will turn on 5 seconds after pressing the button. If the green light was on no more than 10 seconds ago, then the green will turn on after the timeout between transitions equal to 15 seconds. If green is turned on, it will be on for 30 seconds, then red turns on.

Thus, there is one input signal ("the button is pressed") and one control signal ("green is on"). The program gets the input signal and, depending on it, controls the traffic light signal.

This program should satisfy the following requirements:

1. If the red light is on and the button is pressed, the green light will be activated within a maximum of Tr seconds.
2. If the green light just has turned on, then the green light will be on for at least Tg seconds.
3. Once the green light has just been activated, it will transition to red within a maximum of Tg seconds.
4. If the red light just has turned on, then the red light will be on for at least Tr seconds.

Here Tr is the maximum time during which a pedestrian waits for the green signal of the traffic light to turn on after pressing the button, equal to 15 seconds, Tg is the duration of the green signal of the traffic light, equal to 30 seconds.

The requirements 1 and 3 belong to the 4th class. The requirements 2 and 4 belong to the 5th class. For example, the first requirement is formalized as the following annotation:

$\text{toEnvP}\, s \wedge$
$(\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge$
$\text{toEnvNum}\, s2\, s \geq Tr \wedge \text{getVarBool}\, s1\, trafficLight = RED \wedge$
$\text{getVarBool}\, s1\, requestButton = NOT\_PRESSED \wedge$
$\text{getVarBool}\, s2\, requestButton = PRESSED \longrightarrow$
$(\exists s4.\, \text{toEnvP}\, s4 \wedge \text{substate}\, s2\, s4 \wedge \text{substate}\, s4\, s \wedge \text{toEnvNum}\, s2\, s4 \leq Tr \wedge$

$$\text{getVarBool}\, s4\, trafficLight = GREEN \wedge$$
$$(\forall s3.\, \text{toEnvP}\, s3 \wedge \text{substate}\, s2\, s3 \wedge \text{substate}\, s3\, s4 \wedge s3 \neq s4 \longrightarrow$$
$$\text{getVarBool}\, s3\, trafficLight = RED)))$$

4.3. Revolving Door

In this case study, a revolving door is considered as a controlled device. Revolving doors are installed at the entrances to buildings with a large flow of visitors. The device consists of a three- or four-section door rotating around a vertical axis, a motor and a brake for instant stop of the door.

In the absence of users, the door is stationary, and when the user approaches, it begins to rotate. The rotation continues while the user is inside the rotation space. The approach of the user and his presence inside the rotation space is registered by the motion sensor (user). If the user leaves the rotation space, then after a certain time the rotation stops.

The pressure sensor registers the pressure on the sectional partitions. Rotation is suspended for a short time when pressure is exerted to the partitions.

We offer the following requirements for this program:

1. When a user enters, the door starts to rotate if pressure is not detected.
2. Rotation continues while the user is inside the rotation space if pressure is not detected.
3. If the user has left the rotation space, then the rotation should stop after no more than 1 second if users do not reappear during this time.
4. If pressure is detected, then rotation should be suspended for at least 1 second.
5. If pressure is no longer detected then rotation should resume no more than 1 second.
6. Simultaneous appearance of rotation and brake signals is prohibited.

Here the requirements 1 and 2 belong to the 2nd class. The requirements 3 and 5 belong to the 4th class. The requirement 4 belongs to the 5th class. The requirement 6 belongs to the 3rd class. For example, the sixth requirement is formalized as the following annotation:

$$\text{toEnvP}\, s \wedge$$
$$(\forall s1.\, \text{substate}\, s1\, s \wedge \text{toEnvP}\, s1 \wedge$$
$$\text{getVarBool}\, s1\, brake = True \longrightarrow \text{getVarBool}\, s2\, rotation = False)$$

4.4. Fridge

In this case study, a fridge is considered as a controlled device. The fridge consists of a fridge and a freezer and has two compresses. The temperature in the fridge is registered by the fridgeTempGreaterMin and fridgeTempGreaterMax sensors, showing whether the temperature exceeds the minimum and maximum values, respectively. To control the temperature in the freezer, the device has sensors freezerTempGreaterMin and freezerTempGreaterMax. The fridge maintains the temperature in the range between the minimum and maximum values. When the temperature in the fridge is exceeded, the compressor (fridgeCompressor) turns on, which turns off when the temperature reaches the minimum value. The freezer Compressor is used for the freezer. When the fridge door is opened, the lighting turns on, which turns off when it is closed. If the fridge door is open for more than 30 seconds, a sound signal (dorsignal) is given.

We offer the following requirements for this program:

1. When the fridge door is opened, the lighting turns on.
2. When the fridge door is closed, the lighting turns off.
3. If the fridge door is open, the signal is given after no more than 30 seconds if the user does not close the door during this time.
4. The sound signal is not given spontaneously. The signal is given only if the door is open for at least 30 seconds.
5. If the temperature in the fridge exceeds the maximum, the compressor turns on.

Here the requirements 1, 2 and 5 belong to the 2nd class. The requirement 3 belongs to the 1st class. The requirement 4 belongs to the 5th class. For example, the fourth requirement is formalized as the following annotation:

$\mathrm{toEnvP}\ s\wedge$
$(\forall s1 s2 s3.\ \mathrm{substate}\ s1\ s2 \wedge \mathrm{substate}\ s2\ s3 \wedge \mathrm{substate}\ s3\ s \wedge \mathrm{toEnvP}\ s1 \wedge \mathrm{toEnvP}\ s2\wedge$
$\mathrm{toEnvP}\ s3 \wedge \mathrm{toEnvNum}\ s1\ s2 = 1 \wedge \mathrm{toEnvNum}\ s2\ s3 < OPEN\_DOOR\_TIME\_LIMIT$
$\mathrm{getVarBool}\ s1\ fridgeDoor = CLOSED' \wedge \mathrm{getVarBool}\ s2\ fridgeDoor = OPEN \longrightarrow$
$\neg\ \mathrm{getVarBool}\ s3\ doorSignal)$

4.5. Thermopot

In this task, a thermopot is considered as a controlled device. A thermopot is the device that combines the functions of a kettle and a thermos. The thermopot has three temperature modes. It heats the water to the temperature corresponding to the selected temperature mode (selectedTemp), and maintains this temperature. The device contains a housing with a sealed flask, which allows it to maintain the required temperature for a long time, a lid and a heating element (heater). There is a control panel with three buttons (button1, button2, button3) on the lid that allow you to select the desired temperature mode. The boiling button (boilingButton) is used to turn on the boiling. During boiling, the lid is locked. Output signal lid controls the lid locking. After boiling, the thermopot switches to the temperature maintenance mode. In the temperature maintenance mode, the heating element turns on when the water temperature becomes more than 5 degrees less than the set temperature. The boilingMode and maintainingMode indicators show whether the thermopot is in the boiling and temperature maintenance mode, respectively.

We propose the following requirements for this program:

1. Until the required temperature is reached, the lid is locked.
2. When the set temperature is reached, the heating element turns off.
3. The heating element turns on when the water temperature becomes more than 5 degrees less than the set temperature.
4. When one of the temperature mode selection buttons is pressed, the corresponding required temperature is set.

   item If the boiling button is not pressed, the heating will not turn on.

Here the requirements 1, 2, 3 and 5 belong to the 2nd class. The requirement 4 belongs to the 3rd class. For example, the first requirement is formalized as the following annotation:

$$\begin{aligned}
&\text{toEnvP}\, s \wedge \\
&(\forall s1 s2.\, \text{substate}\, s1\, s2 \wedge \text{substate}\, s2\, s \wedge \text{toEnvP}\, s1 \wedge \text{toEnvP}\, s2 \wedge \text{toEnvNum}\, s1\, s2 = 1 \wedge \\
&\text{getVarBool}\, s1\, \mathit{boilingMode}' \wedge \text{getVarInt}\, s2\, \mathit{temperature}' < BOILING_POINT' \longrightarrow \\
&\text{getVarBool}\, s2\, \mathit{lid}' = LOCKED')
\end{aligned}$$

5. Conclusion

In this paper, we proposed a collection of control programs in process-oriented poST language. We formulated a set of temporal requirements for each program. These requirements have been classified according previously developed requirements patterns. This classification confirmed that most of the requirements belong to previously introduced classes, but some requirements form a new class. We defined the pattern for this requirements class. The developed collection of process-oriented programs can be used in the research of methods of formal verification of such programs, in particular, in the development of a methodology for proving verification conditions.

In the future, we plan to investigate the possibility of automation of proving verification conditions for requirements belonging to developed classes.

References

1. Anureev I., Garanina N., Liakh T., Rozov A., Zyubin V., Gorlatch S. Two-step deductive verification of control software using Reflex. // International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer. 2019. P. 50–63.
2. Chernenko I.M., Anureev I.S., Garanina N.O., Staroletov S.M. A Temporal Requirements Language for Deductive Verification of Process-Oriented Programs // 2022 IEEE 23rd International Conference of Young Professionals in Electron Devices and Materials (EDM). 2022. P. 657-662.
3. Dwyer M.B., Avrunin G.S., Corbett J.C. Patterns in property specifications for finite-state verification // Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). 1999. P. 411-420.
4. Fillitre J.C. Deductive software verification // Int J Softw Tools Technol Transfer. 2011. Vol. 13. P. 397-403.
5. Gurov D., Herber P., Schaefer I. Automated Verification of Embedded Control Software // International Symposium on Leveraging Applications of Formal Methods / Springer. 2020. P. 235-239.
6. Gurov D., Lidström C., Nyberg M., Westman J. Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report // Critical Systems: Formal Methods and Automated Verification. Cham : Springer International Publishing, 2017. P. 3-18.
7. Lourenço C.B., Cousineau D., Faissole F. et al. Automated Verification of Temporal Properties of Ladder Programs // Formal Methods for Industrial Critical Systems / Ed. by Lluch Lafuente A., Mavridou A. Cham : Springer International Publishing, 2021. P. 21-38.
8. Manna Z., Bjørner N.S., Browne A. et al. An Update on STeP: Deductive-Algorithmic Verification of Reactive Systems // Tool Support for System Specification, Development and Verification / Ed. by Berghammer R., Lakhnech Y. Vienna: Springer Vienna, 1999. P. 174-188.
9. Mekki A., Ghazel M., Toguyeni A. A. K. Assisting Temporal Requirement Specification // Computer Technology and Application. 2012. Vol. 3. P. 47-55.
10. Sin C. O., Kim Y.S. TimeLine Depiction: an approach to graphical notation for supporting temporal property specification // Innovations in Systems and Software Engineering. 2023. P. 319-335.
11. Zyubin V. E. Hyper-automaton: a Model of Control Algorithms // 2007 Siberian Conference on Control and Communications. 2007. P. 51-57.
12. Zyubin V.E., Rozov A.S., Anureev I.S. et al. PoST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language // IEEE Access. 2022.
13. Chernenko I. PoST verification condition generator. URL: https://github.com/ivchernenko/post_vcgenerator (online; accessed: 21.10.2023)

UDK 004.4'42

Model Checking Programs in Process-Oriented IEC 61131-3 Structured Text

Garanina N.O. (Institute of Automation and Electrometry SB RAS)

Staroletov S.M. (Institute of Automation and Electrometry SB RAS)

Zyubin V.E. (Institute of Automation and Electrometry SB RAS)

Anureev I.S. (Institute of Automation and Electrometry SB RAS)

The process-oriented programming is a paradigm based on the process concept where each process is a concurrent finite state machine inside. The paradigm is intended for PLC (programmable logic controllers) developers to write Industry 4.0-enabled software. The poST language is a promising process-oriented extension of the IEC 61131-3 Structured Text (ST) language designed to provide a conceptual consistency of the PLC source code with technological description of the process under control. This language combines the advantages of FSM-based programming with the standard syntax of the ST language. We propose transformational semantics of poST providing rules for translation of poST language statements to Promela – the input language of the SPIN model checker. Following these semantic rules, our Xtext-based translator outputs a Promela model for the poST program. Our contribution is a method for automatic generation of the Promela code from poST control programs. The resulting Promela program is ready to be verified with SPIN model checker against linear temporal logic requirements to the source poST program.

Keywords: control software, model checking, process-oriented programming, LTL, SPIN, Structured Text

1. Introduction

With the significant progress in formal methods for proving program models, the question of their applicability to real software remains. It can be stated that the application of such methods to general programs does not seem appropriate due to the laboriousness of translating language constructs with ambiguous semantics, a large state space, and the uncertainty or inexpressibility of requirements.

In the field of control software, the situation is twofold: on the one hand, the cost of an error is high, and control programs need formalization and proving correctness with respect to formalized requirements, on the other hand, programs for PLC (Programmable Logic Controllers) designed to implement industrial process control algorithms are concise, the languages themselves contain a small number of constructions, the requirements are expressible as there are few key control variables. These features of PLC languages makes applicability of formal methods more promising.

Currently, the languages of the IEC 61131-3 standard [11] are used for programmable industrial controllers. They differ in program representation: in text, ladder diagrams, function block diagrams and instruction lists. In this work, we focus on a textual representation in Structured Text (ST) language. This language is Pascal-like and has a simple syntax with some features (e.g. timers, intervals) specific to PLC programs [10, 11]. However, most control programs are cyclical and state dependent, resulting in a large number of switch statements.

Exploiting this feature and presenting developers with convenient means of operating with states and transitions, we introduce the concept of process as the main logical entity of the program. We call programming according to this methodology process-oriented, and the language that adds such syntactic sugar is called process-oriented programming language. Note, that it is reasonable to extend standardized ST language, since the development, simulation and firmware creation environments for real controllers are already adapted for this language. The poST (process-oriented Structured Text) language is one of such extensions, which is compiled into the ST language. The source-to-source compiling is also called transpiling, and nowadays it is a convenient means to switch between various languages [8].

The use of formal verification methods for PLC programs is justified primarily by the fact that such programs can work with expensive equipment of some plant and the incorrect behavior of the program can lead to both financial losses and serious consequences for the environment and plant personnel. Therefore, it is necessary to provide as many means as possible for static checks of such programs in the early stages of their production.

In this paper, we present an automatic translation of poST programs into programs in the input language of a program verifier which use the model checking method. A review of the practices of applying the model checking for such systems is presented in the paper [9].

In our series of previous works, we used the SPIN verification tool [7] coming with the Promela input language (Protocol Meta-Language), which corresponds to the CSP approach [6], that is, it has the ability to describe systems as interacting processes. This is close to process-oriented, but not quite the same: the language does not work with process states by language means, although states, processes and switching between them can be represented in the form of conditional structures, additional variables, and passing the progress through channels.

In this paper, we describe, the code transformation from poST to Promela as well as the issues of formalizing and implementing such transpiling. The implementation is a solution based on the Eclipse Xtext parser tool in Java and Xtend.

The rest of the paper has the following structure. In Section 2, we consider features of the poST and Promela languages. Section 3 defines rules for the transformational semantics for the poST language and the conclusion is given in Section 4.

This work is supported by the Russian Ministry of Education and Science, project no. 122031600173-8.

2. The poST and Promela languages

The poST language is a novel process-oriented extension of the IEC 61131-3 Structured Text (ST) language which provides conceptual consistency of the PLC source code with technological description of the process under control. The language combines the advantages of FSM based programming with the conventional syntax of the ST language which would facilitate its adoption.

Inspired by a general PLC scan cycle, a poST program includes processes whose activity is orchestrated into a cycle in order of their appearance in the program code. This scheme expects PLC models that abstract from a scan cycle time [5]. Each process is specified by an ordered set of process states. To describe process states, we use standard ST constructs (variable declarations, control-flow statements, etc.) and specific process-oriented features: process states statements (`START`/`STOP PROCESS`, `SET NEXT` and other) and timeout statements. The semantics of the poST language assumes an automatic implementation of low-level constructions for mapping I/O signals to program variables, process states, timeout statements, and cyclic time-triggered control. The grammar of the poST language in the Xtext format is available in the repository [15].

Promela language [13] is used to describe parallel communicating processes based on the CSP formalism [6]. Promela program consists of parallel processes communicating through channels or shared variables. The execution of a set of Promela parallel processes exploits the interleaving semantics. Interleaving can be bounded by `atomic` and `d_step` statements, which permit interruption of specified sequence of process actions. The Promela language includes blocking control-flow statements `if` and `do`, unlike standard non-blocking poST control-flow statements. Promela model can be verified by model checker SPIN [7] against LTL requirements, hence it assumes only finite types for model variables. This causes the main difficulty of translation poST programs to Promela models because poST types for real numbers cannot be directly translated to Promela types.

The poST language specifies control algorithms which interact with some environment. Hence, we suppose that the source poST code may contain the control program and the controlled object program. Translating this code, we construct the Promela model that corresponds to the order of processes activation, the structure of process states, timeout management, and variable types up to abstracting from real types.

3. poST Language Transformation Rules

In this section, we describe representative translations from poST language to Promela language. We start from common constructs as types, variables, standard operations and control-flow statements (Sections A and B). Further, we present special process-oriented features, namely, a scan cycle, process states, timeouts, and special LTL forms for requirements. The complete formal transformational semantics for poST is located in the repository [14].

3.1. Naming, Types, Declarations, and Operations

poST has global, program, and process namespaces, while Promela only has global and process namespaces. To avoid name collisions, we form the full entity names in the Promela model of a poST program taking care of (1) the entity name; (2) the entity kind (process, variable, etc.); and (3) the name of the entity owner. To improve program readability, we use the default naming mode which takes into account just entity kinds enriched with counters if there are several equal names in the global Promela namespace.

Variable types are translated trivially in most cases. We consider poST programs without real variables, since Promela does not have real types, and data type abstraction is beyond the scope of this paper. Translation of the `TIME` type is discussed below when describing the translation of the `TIMEOUT` expression.

Following this naming and type policy, all renamed poST variables are declared in Promela model as global variables, and poST constants are trivially translated using Promela directive `#define`.

Promela includes the same operations as poST except exponentiation which can be modeled bit-wise shift Promela operations.

3.2. Control-Flow Statements

In Table 1 we give translation two kinds of poST control-flow statements to Promela code. Let *code′* be the Promela image of the poST *code* made by our translation algorithm. We use the Promela `else` branch and the `skip` statement in the translation of the poST `IF` statement because the Promela `if` statement is blocking and the process cannot proceed further if condition *cond* is false. Following the Promela semantics, `else` branch is chosen when no other condition in `if` statement is satisfiable. The `skip` statement in this branch just does nothing. The poST `CASE` statement is translated in the similar way. The Promela `do` statement is also blocking, hence in modeling poST `DO` statement we need the `else` branch with the `break` statement to model the termination of a loop in Promela.

Table 1

Control-Flow statements

poST	**Promela**	**poST**	**Promela**
`IF` *cond* `THEN`	`if ::` *cond′* `-> {` *body′* `}`	`WHILE` *cond*	`do ::` *cond′* `-> {` *body′* `}`
body	`:: else -> skip;`	`DO` *body*	`:: else -> break;`
`END_IF`	`fi;`	`END_WHILE`	`od;`

3.3. Process-oriented features

A control system is specified in poST by a set of poST programs. Hence, we should translate several poST programs into a single Promela model. A particular poST program consists of processes which activated cyclically one after another. Several poST programs run in a joint scan cycle in the order of their appearance in the source code. In Promela, we model this cyclic activation sequentially passing *a move message* from a process to a process through a Promela channel in the order in which poST processes appear in the source programs. Promela channels support blocking reads and writes. We use the channel `turn` of capacity 1 to pass nicknames of processes in move messages. Every process is initially blocked until reading its nickname from this channel. After execution of its body, a process pushes the nickname for the next process into the channel to pass the move. The Promela process for the last poST process turns the move to the service process `Gremlin` which represents the non-deterministic environment generating input values. This process is the first process of the resulting Promela model. Due to sequential activation semantics of poST programs, we use Promela `atomic` statement for the body of every translated poST process. This statement permits interleaving execution of other processes and significantly simplifies verification. The left block of Table 2 gives translation of the upper process structure of poST programs to Promela.

Table 2

Process-oriented features

poST	**Promela**	**poST**	**Promela**
`PROGRAM` $prog_1$	`mtype : P =`	`PROCESS` id	`mtype:S_`id' `=`
`PROCESS` id_{11}	`{ p_`id'_{11}`, ... }`	`STATE` s_1	`{s_`s'_1`,...,s_`$Error'$`}`
$body_{11}$	`chan turn=[1]of{mtype:P}`	$body_1$	`mtype:S_`id'
`END_PROCESS`		`END_STATE`	`c_`id' `= s_`$Stop'$`;`
`...`	`active proctype` id'_{11}`(){`	`...`	`active proctype` id'`(){`
`END_PROGRAM`	`do :: turn ? p_`id'_{11} `->`	`STATE` s_n	`do :: turn ? p_`id'`->`
`...`	`atomic {`	$body_n$	`atomic {`
`PROGRAM` $prog_m$	$body'_{11}$`;`	`END_STATE`	`if`
`PROCESS` id_{1m}	`turn ! p_`id'_{21}`;}`	`END_PROCESS`	`:: c_`id'`==s_`s'_1`->`
$body_{1m}$	`od;`		`{` $body'_1$ `}`
`END_PROCESS`	`}`		`...`
`...`	`...`		`:: else ->skip;`
`END_PROGRAM`	`active proctype` id'_{nm}`(){`		`fi;`
	`...`		`turn !` nx_pr`;}`
	`}`		`od;`
			`}`

A body of a poST process consists of *states*, including special states of inactivity `STOP` and `ERROR`. At every iteration of the scan cycle, the poST process executes the code corresponding to some of its states except states `STOP` and `ERROR` when it does nothing. In Promela, we use a special state counter for every translated process to keep the name of the current state. At the start of a poST program, its first declared process is in its first state and all other processes are in state STOP. The right block of Table 2 gives translation to Promela for a particular poST process.

poST processes can check an activity status of other processes with statement `ACTIVE` and `INACTIVE`. Also each process can force itself or anther process to change its state with statements `RESTART`, `STOP`, `START PROCESS`, `STOP PROCESS`, and others. These poST statements are translated trivially to Promela if the goal state do not include `TIMEOUT` statement. Please, see examples in the right block of Table 3.

poST process states can have a `TIMEOUT` block as the last state block. Instructions of this block is executed after the time specified in the timeout has elapsed since the process entered this state. To model this behaviour in Promela, we introduce a counting time variable — one per each process that contains states with TIMEOUT.

To reduce the size of the Promela model, we provide the following optimisation for model time counters. First, we use the value of poST scan cycle (INTERVAL) to reduce all timeout values to the nearest multiple of this interval. Second, we divide all timeout values by their greatest common divisor. In addition, we choose the minimum sufficient size *nb* of the unsigned type for the time counters. For example, we add one time counter with a size of 4 bits if a resulting Promela process has two states with timeouts that count 5 (101b) and 9 (1001b) units of time.

At every scan cycle, if a process in a state with a timeout, its time counter is incremented. A process counter sets to zero when (1) the process resets the timeout; (2) the process moves to other state; and (3) timeout happens. Following poST semantics, we use the > sign in the Promela timeout if statement because the execution of the timeout block begins at the next cycle after the timeout time has passed. We give the representative model constructs for timeouts in the left block of Table 3.

Table 3

State and Timeout Statements

poST	**Promela**	**poST**	**Promela**
IF (PROCESS id	if	PROCESS id_1	unsigned t_id'_1 : nb
INACTIVE)	:: c_id' == s_$Stop'$ ||	STATE s_1	active proctype id'_1()
THEN $body$	c_id' == s_Err' ->	$body_1$	{ ...
END_IF	{ $body'$ }	TIMEOUT T#tt	:: c_id'_1 == s_s'_1 ->{
PROCESS id	:: else -> skip; fi;	THEN $body_t$	$body'_1$
STATE s_1	active proctype id'(){	END_TIMEOUT	if
$body_1$	do :: turn ? p_id'->	END_STATE	:: t_id'_1 > tt' ->
SET NEXT	atomic {	...	$body'_t$
END_STATE	if	END_PROCESS	:: else ->t_id'_1++;
STATE s_2	:: c_id'==s_s'_1 ->		fi;}
$body_2$	{ $body'_1$		... }
END_STATE	c_id' = s_s'_2; }	PROCESS id_2	active proctype id'_2()
...	:: c_id'==s_s'_2 ->	STATE s_2	{ ...
END_PROCESS	{ $body'_2$ }	$body_2$	:: c_id'_2 == s_s'_2 ->{
	...	START	$body'_2$
	:: else -> skip;	PROCESS id_1	c_id'_1 = s'_1;
	fi;	END_STATE	t_id'_1 = 0; }
	turn ! $next$;}	...	...
	od; }	END_PROCESS	}

Table 4

The Promela model for poST programs

poST	**Promela**
`PROGRAM` $prog_1$	$Var\_Declaration'_1$
$Var\_Declaration_1$	`...`
`PROCESS` $name_{11}$	$Var\_Declaration'_m$
`...`	$Service\_Declarations$
`PROCESS` $name_{n1}$	`init{ turn !  p_Gremlin; }`
`END_PROGRAM`	`active proctype Gremlin(){...}`
`...`	`active proctype OutInput(){...}`
`PROGRAM` $prog_m$	`active proctype BOC(){...}`
$Var\_Declaration_m$	`active proctype` $name'_{11}$`(){...}`
`PROCESS` $name_{1m}$	`...`
`...`	`active proctype` $name'_{nm}$`(){`
`PROCESS` $name_{nm}$	`do ::  turn ?  p_`$name'_{nm}$ `->`
`END_PROGRAM`	`atomic { ...`
	`turn !  p_Gremlin; }`
	`od;`
	`}`

3.4. Overall Promela model for poST programs

In general, our translation algorithm takes as input several poST program that describe a control system in one file. This control system may include the control algorithm and its environment: controlled and uncontrolled objects. In Table 4, we give the resulting Promela model, which includes three service processes and processes corresponding the source poST processes. Non-deterministic service process `Gremling` models uncontrolled object. Service process `OutInput` cares about proper corresponding of inputs and outputs of programs composing the source program. Service process `BOC` captures the beginning of the scan cycle for checking requirements. Activity of these processes forms a scan cycle by passing move messages between them starting from the Gremling process modeling initial inputs from uncontrolled object and ending with the last source poST process which pass move message to Gremlin again. Inside the cycle, the processes activity is ordered as described in Table 2.

4. Discussion and Conclusion

In this paper, we have considered approaches to formalization and implementation of the source-to-source compiling (transpiling) process. For PLC programming languages with a small number of statements, such a process is justified, since all constructs of the input language can be converted into language constructs for which methods of formal program verification have already been well developed (in this case, we use the model checking method and SPIN verification system). This allows us to make examples of PLC programs intended for verification and not to write repetitive code constructs in Promela related to the semantics of process switching, transitioning through states, exchanging variable values and generating an impulse for checking program properties w.r.t. scan cycles. The project is publicly available in our repository [12]. Among the shortcomings of the current approach, we note the incomplete support for data types, in particular, real variables are not supported. This can be eliminated by implementing libraries of both fixed-point and floating-point arithmetic, however, this will entail a huge number of states and the verification of resulting programs cannot be done without manual abstraction methods. Also, library functions are not implemented.

As a result, with the development of the considered transpiler, we are approaching the development of a toolchain for writing verifiable programs in the process-oriented style.

References

1. Zyubin V. E., Rozov A. S., Anureev I. S., Garanina N. O. and Vyatkin V. poST: A Process-Oriented Extension of the IEC 61131-3 Structured Text Language // IEEE Access 2022. Vol. 10, P. 35238–35250.
2. Ponomarenko A. A., Garanina N. O., Staroletov S. M., and Zyubin V. E. Towards the Translation of Reflex Programs to Promela: Model Checking Wheelchair Lift Software // Proc. of IEEE 22nd Intern. Conf. of Young Professionals in Electron Devices and Materials (EDM). 2021. P. 493–498.
3. Anureev I. S., Garanina N. O., Liakh T. and Rozov A. S., and Schulte H. and ZyubinV. E. Towards safe cyber-physical systems: the Reflex language and its transformational semantics // Proc. of 2019 Intern. Siberian Conf. on Control and Communications (SIBCON). 2019. P. 1–6.
4. Clarke E.M., Henzinger T. A., Veith H., and Bloem R. Handbook of model checking. Springer, 2018. 1210 p.
5. Mader A. A Classification of PLC Models and Applications // Discrete Event Systems 2000. Vol. 596. P. 239–246.
6. Hoare C. A. R. Communicating sequential processes. Prentice-Hall: 1985.
7. Holzmann G. J. The Spin Model Checker, Primer and Reference Manual. Addison-Wesley: 2003.
8. Schneider L. and Schultes D. Evaluating Swift-to-Kotlin and Kotlin-to-Swift transpilers //

Proc. of the 9th IEEE/ACM Int. Conf. on Mobile Software Engineering and Systems. 2022. P. 102–106.

9. Ovatman T. An overview of model checking practices on verification of PLC software // Software & Systems Modeling. 2016. Vol 4, No 15. P. 937–960.
10. Antonsen T. M. PLC Controls with Structured Text (ST), V3: IEC 61131-3 and best practice ST programming. BoD–Books on Demand, 2020.
11. IEC 61131-3:2013. Programmable controllers - Part 3: Programming languages. 2013. URL: https://webstore.iec.ch/publication/4552
12. Translator-poST-Promela. 2023. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev
13. Promela grammar. URL: http://spinroot.com/spin/Man/grammar.html
14. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/semantics/transformation.pdf, 2023.
15. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/poST_grammar.xtext, 2022.
16. URL: https://github.com/SergeyStaroletov/poST_to_Promela_compiler_dev/samples/, 2023.
17. Xtext is a framework for development of programming languages and domain-specific languages. 2022. URL: http://eclipse.org/Xtext
18. Xtend is a flexible and expressive dialect of Java. 2021. URL: https://www.eclipse.org/xtend/

UDK 004.4’418

SSA Algebras

Sokolov P.P. (Higher School of Economics)

SSA (static single-assignment) form is an intermediate representation for compiling imperative programs where every variable is assigned to only once. Properly defined, SSA form gives rise to the family of purely syntactic categories with some nice properties. We hope this lays out the groundwork for categorical approach to compiler optimization.

Keywords: formalisms for program semantics, category theory, program synthesis and transformation

1. Introduction

In compilers of imperative languages, a program is usually represented as a control flow graph (CFG) where every block is a sequence of assignments and, notably, reassignments. SSA (static single-assignment) is a special form of CFG where every variable is assigned to exactly once. SSA form is useful because it opens up many possibilities for optimization of programs: pruning, constant folding, register allocation, language-dependent optimizations etc.

Here we argue that it is also a suitable basis for a categorical framework to judge about compiler optimizations, too. First, we define a base category SSA; then we try to define a product in SSA and arrive at an important equivalence relation for programs. Finally, we provide categorical definitions of optimization and compilation. **Main results** are presented as theorems in bold.

2. Related work

SSA is a hot and thoroughly studied research topic; while there already is a comprehensive book on SSA-based compiler design [1], new papers keep coming out [2, 3]. This paper, however, is not in the tradition of previous SSA works because we focus on the formal, mathematical, categorical side of things.

Applying category theory to programming languages has a long history; it is extensively used to model execution of programs either inside Kleisli category [4] or via Curry-Howard-Lambek correspondence [5]. Unfortunately, it is impossible to express the notion of optimization in these categories; **our approach allows one to judge about optimization inside a categorical framework**.

3. The categories of SSA programs

For the language of expressions of an SSA program, let us use the set of expressions $E = \Sigma[\mathbb{N}]$ in arbitrary algebraic signature Σ where variables are taken from the set of natural numbers starting from 0. The following definition is useful:

Definition 1. *An expression $e \in E$ is said to be **bound by** n iff $n \in \mathbb{N}$ and, for all variables $x \in \mathbb{N}$ mentioned in e, $x < n$.*

Now, we consider SSA programs to be a sequence of assignments where, for new binding x_i, one can refer to one of m *inputs* and to one of i previous assignments using variables $\{0, \dots, m-1\}$ and $\{m, \dots, m+i-1\}$, respectively.

Definition 2. *Let $P_{m \to n} \subseteq E^* \times \mathbb{N}^n$ be the set of valid SSA programs for language Σ with m inputs and n outputs, that is, a set of pairs (O, R) where, for every $r \in R$, $r < m + |O|$ and, for every $i < |O|$, O_i is bound by $m + i$.*

Lists of operations O can be concatenated as lists with $(+\!\!+)$; same goes for output tuples R. To turn concatenated lists back into valid programs, we use *substitution* which is applied to natural numbers and substitutes them according to the first matching predicate. An example of how substitution works on an expression in the language with binary $(+)$:

$$(x_1 + x_3 + x_0)[i < 2 \mapsto i + 1 \mid\mid i \mapsto i - 1] = x_2 + x_2 + x_1$$

Notable examples of SSA programs include:

Definition 3. *For every natural n, there is the **identity** SSA program with no operations:*

$$\mathrm{id}_n = (\varepsilon, (0, \dots, n-1)) \in P_{n \to n}$$

Definition 4. *For every permutation $\sigma \in S_n$, there is an SSA program which reorders the inputs according to σ:*

$$p_\sigma = (\varepsilon, (\sigma_0, \dots, \sigma_{n-1})) \in P_{n \to n}$$

Definition 5. *Given SSA programs $f : P_{m \to n}$ and $g : P_{k \to m}$, **composition** $f \circ g : P_{k \to n}$ is a program which feeds outputs of g into f, that is,*

$$O^{f \circ g} = O^g +\!\!+ O^f[i < m \mapsto R^g_i \mid\mid i \mapsto i - m + k + |O^g|] \tag{1}$$

$$R^{f \circ g} = R^f[i < m \mapsto R^g_i \mid\mid i \mapsto i - m + k + |O^g|] \tag{2}$$

Our **first main result** is that this turns out to be a category:

Theorem 1. *There is a category* $\mathrm{SSA}(\Sigma)$ *of SSA programs in language* Σ *with* $\mathrm{Ob} = \mathbb{N}$ *and* $\mathrm{Hom}(m, n) = P_{m \to n}$.

The proof goes as follows: id and $f \circ g$ are given above. Their properties are proven by obvious induction on the lengths of programs.

Now, the problem with this category is that SSA programs might contain expressions they never even reuse. It might be useful if the compiled language has side-effects; but we can model effectful programs by threading some extra variables through IO operations, as is done in languages with linear typing like Clean [6]. To get rid of truly unused expressions, we (informally) define

Definition 6. *Let* $\mathrm{prune} : P_{m \to n} \to P_{m \to n}$ *be a function which, for every SSA program, removes operations not reachable from the output and patches indices accordingly.*

Statement 1. *There is a category* PrunedSSA *of pruned SSA programs with* $\mathrm{Hom}(m, n) = \{\mathrm{prune}(p) \mid p \in P_{m \to n}\}$.

The proof goes as follows: let $\mathrm{id}(n) = \mathrm{id}_{\mathrm{SSA}}(n)$ and $f \circ g = \mathrm{prune}(f \circ_{\mathrm{SSA}} g)$. The following are obvious:

$$\mathrm{prune}(\mathrm{id}_{\mathrm{SSA}}(n)) = \mathrm{id}_{\mathrm{SSA}}(n); \tag{3}$$

$$\mathrm{prune}(f \circ \mathrm{prune}(g \circ h)) = \mathrm{prune}(f \circ g \circ h) = \mathrm{prune}(\mathrm{prune}(f \circ g) \circ h). \tag{4}$$

Statement 2. *In* PrunedSSA, *0 is a terminal object with* $! = (\varepsilon, ())$.

To get into products, we first have to define another operation on programs.

Definition 7. *Given SSA programs* $f : k \to m$ *and* $g : k \to n$*, their* ***parallel composition*** $f|g : k \to (m + n)$ *is a program which computes both* f *and* g *from the same inputs, that is,*

$$O^{f|g} = O^f +\!\!+ O^g[i < k \mapsto i \mid\mid i \mapsto i + |O^f|] \tag{5}$$

$$R^{f|g} = R^f +\!\!+ R^g[i < k \mapsto i \mid\mid i \mapsto i + |O^f|] \tag{6}$$

Statement 3. *In* PrunedSSA, *for every* $f : k \to m$ *and* $g : k \to n$, $f|g$ *commutes with* f, g, $\pi_1 = (\varepsilon, (0, \ldots, m - 1))$ *and* $\pi_2 = (\varepsilon, (m, \ldots, m + n - 1))$ *as a product.*

Note that $f|g$ is not unique as a product arrow: operations from both f and g can be freely intertwined and the result would still satisfy the product property. Furthermore, both f and g might contain the same expressions which can be reused in a product arrow, which, actually, is a well-known CSE (common subexpression elimination) problem [7]. Finally, composing effectful programs like this is slightly wrong. The following equivalence helps with the first problem:

Definition 8. *Two SSA programs $p, q : P_{m \to n}$ are called equivalent (or $p \sim q$) iff $l = |O^p| = |O^q|$ and there exists a permutation $\sigma \in S_l$ such that the following holds:*

$$O^p_j = O^q_{\sigma(j)}[i < m \mapsto i \mid\mid i \mapsto \sigma(i - m) + m] \tag{7}$$

$$R^p = (\sigma(R^q_1), \ldots, \sigma(R^q_n)) \tag{8}$$

Theorem 2. $p \sim q$ ***is an equivalence relation;*** $f \circ g$ ***and*** $f|g$ ***respect it.***

The proof goes as follows: equivalence follows from properties of permutations; respect of $f \circ g$ and $f|g$ can be proven by taking permutations that act on parts of $f \circ g$ and $f|g$.

4. Optimization, compilation and semantic functors

In this framework, optimization and compilation procedures become functors of SSA which act on morphisms in the semantic-preserving way. Reserving study of semantics of effectful programs for further work, here we outline the definitions and properties of such functors via **semantic functors**:

Definition 9. *Given a structure S of signature Σ, semantic category* $\mathrm{Sem}(S)$ *is a category with* $\mathrm{Ob} = \mathbb{N}$ *and* $\mathrm{Hom}(m, n) = (S^m \to S)^n$.

Definition 10. *A semantic functor $F_S : \mathrm{SSA}(\Sigma) \to \mathrm{Sem}(S)$ is a functor with $F_S(n) = n$ which interprets SSA programs as functions on vectors over S.*

In terms of semantic functors, optimization and compilation can be defined like this:

Definition 11. *An endofunctor $Q : \mathrm{SSA}(\Sigma) \to \mathrm{SSA}(\Sigma)$ is an optimization with respect to S iff $F_S \circ Q \simeq F_S$.*

Definition 12. *Given a functor $G : \mathrm{Sem}(S) \to \mathrm{Sem}(T)$, where S is a structure of Σ and T is a structure of Ξ, a functor $C : \mathrm{SSA}(\Sigma) \to \mathrm{SSA}(\Xi)$ is a compilation with respect to G iff $F_T \circ C \simeq G \circ F_S$.*

Note that, in the case of compilation, G (and, consequently, C) might not keep the objects (sizes of inputs/outputs) intact. This is reserved for the case where a single $s \in S$ is represented by a vector of values from T.

Study of the properties of optimization and compilation functors is reserved for further work.

References

1. Rastello F. SSA-Based Compiler Design. Springer Publishing Company, Incorporated, 2016.
2. Buchwald S., Lohner D., Ullrich S. Verified Construction of Static Single Assignment Form // Proceedings of the 25th International Conference on Compiler Construction. Barcelona, Spain, 2016. P. 67–76.
3. Bhat S., Grosser T. Lambda the Ultimate SSA: Optimizing Functional Programs in SSA. [2022]. URL: https://arxiv.org/abs/2201.07272 (access date: 28.10.2023).
4. Moggi E. Notions of computation and monads // Information and Computation. 1991. Vol. 93, № 1. P. 55–92.
5. Lambek J., Scott P.J. Introduction to higher-order categorical logic. Cambridge University Press, 1988.
6. Plasmeijer R., Eekelen M. Keep It Clean: A Unique Approach to Functional Programming // SIGPLAN Not. New York, NY, USA: Association for Computing Machinery, June 1999. Vol. 34, № 6. P. 23–31.
7. Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997. P. 378–396

UDK 004.415.53

Metamorphic testing for generative artificial intelligence systems

Iakusheva S.F. (Moscow Institute of Physics and Technology)

Khritankov A.S. (Moscow Institute of Physics and Technology, Higher School of Economics University)

Harbachonak D.I. (Higher School of Economics University)

Generative artificial intelligence systems should be carefully tested because they can generate low-quality content, but the testing challenges the test oracle problem. Metamorphic testing helps to test programs without test oracles. In this study, we consider an AI system that generates personalized stickers. Personalized sticker is a thematic picture with a person's face, some decorative elements and phrases. This system is stochastic, complex and consists of many parts. We formulate requirements for the whole system and check them with metamorphic relations. The requirements focus on dependency on input images quality and resulting quality of generated stickers. Our testing methodology helps us to identify major issues in the system under test.

Keywords: metamorphic testing, generative artificial intelligence systems, quality control

1. Introduction

Nowadays artificial intelligence (AI) systems are commonly used for content generation, for example, Midjourney project and other diffusion neural networks [12]. Generative AI systems affect the users, and such content may have inappropriate quality or even be harmful. So, verification of the requirements for these systems should be precise.

One of the challenges is the test oracle problem [2]: difficulty or impossibility to compute if the program's output is correct. Generative AI can produce a lot of different items for one generation request, and every item may be called correct, so we only can check some metrics or use human supervision. Thus, verification of the quality requirements is complicated.

There are some methods to verify programs without test oracles, one of them is metamorphic testing [3]. In this study, we apply metamorphic testing to a generative AI system for picture generation. We consider an image-generation system [7], formulate two requirements, propose two metamorphic relations, test the system and find major issues.

The article has the following structure. Section 2 provides the background on the problems of testing AI systems. Section 3 describes the system under test. Section 4 gives metamorphic relations for the system. Then Section 5 describes the experiment. Section 6 concludes our work.

2. Related work

Many methods for image generation exist [6], for example: stylization, style transfer and animation, image composition, generating images from text (Adversarial Networks (GANs), Transformers and Diffusion Models). All these methods have their specific advantages and limitations.

There are many validation approaches for AI [13]: classification-based [8], model-based, learning-based [9], rule-based [4] and non-oracle (metamorphic [15]). Metamorphic testing [3] is one of the testing methods for programs with the test oracle problem. It allows us to easily generate test inputs and automate the test process. This method uses so-called metamorphic relations instead of verifying every test output or calculating metrics.

The metamorphic relation for a program can be represented as a function

$$R(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \longrightarrow \{0, 1\}, \qquad (1)$$

where $n \geq 2$ is the total number of test inputs, x_i as a i-th program input and $f(x_i)$ is a i-th output.

Algorithm of metamorphic testing is quite simple. The system is run on several test inputs, for which some relation holds. Then, we check the presence of the corresponding relation between the test outputs. Thus, we consider the evolution in the system response when we change the input data. There is no need for checking every single test output for correctness.

Metamorphic testing is used for testing different stochastic AI systems like machine translators [10], web systems [1], image classifiers [5], image recognition systems [16], etc. Metamorphic testing is also used for testing stochastic systems. In this case, statistical methods are used for verifying the relations (for example, criteria [1], correlation [17], ANOVA [10]).

3. System under test

We consider a sticker-generating system (SGS) [7] as a system under test (SUT). We call a "sticker" a thematic picture with a person's face, some decorative elements and maybe praising phrases on it (Fig. 1). For example, a sticker can be presented to a user after finishing an online course. However, some input images cannot be transformed into stickers because they lack some parts of the face, have poor quality, etc. In this situation SGS does not generate anything and raises exceptions.

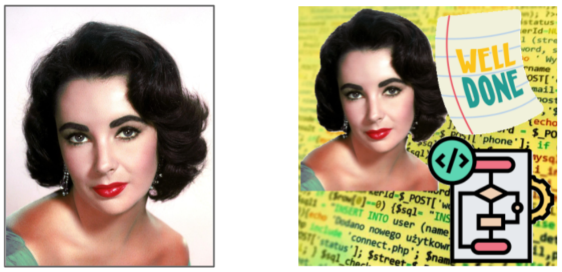

Fig. 1. Original image and a "programming" thematic sticker.

Previously [7], this system was compared with the simple basic solution. The users decided that generated stickers were more interesting and aesthetically pleasing. Unfortunately, this method is not automatic.

SGS contains many components: a neural network (NN) for face segmentation (MediaPipe), a NN for determining the topic of the sticker description (SpaCy), and auxiliary tools for blending, adding glasses, multification (AnimeGAN), style transfer and quality control.

Of course, NN components of SGS can be tested with metamorphic testing individually. But the whole system is much more complex, and good quality of the components does not guarantee the good quality of the system. Our purpose is to verify requirements for the whole system. So, we consider it as a "black box" and do not use its internal structure.

4. Proposed method

4.1. Requirements and Metamorphic relations

Let us consider SUT as a stochastic function $F(x, t)$ where x denotes an input image and t denotes a text description.

We choose two requirements from different areas to test the SUT.

The first requirement relates to the user's needs. Any user should recognize his or her face on the sticker. Also, this sticker can be used as a profile photo. So, SUT should preserve the face in such a way that the face on the sticker should look almost the same as the face on the original image. So, if we use the sticker as an input for generating a new sticker, this new sticker should be generated. We interpret this as a

Requirement 1: *every sticker is an ideal input and new stickers can be produced from every sticker.*

So, we formulate metamorphic relation:

MR1: $count(F(x_i, t)) = count(F(F(x_i, t), t))$, $1 \leq i \leq n$

which means that the number of successfully generated stickers is not reduced after repeated generation.

The second requirement origins in mathematical models of SUT components. Segmentation NN should work worse on images with the poor quality (e.x. with noise and low contrast). We can decrease the quality of the image until SUT rejects it. And if we decrease the quality even more, this image also will be rejected by SUT. So, image quality should affect the number of generated stickers.

Requirement 2: *the worse the image quality, the less the number of successfully generated stickers.*

If we sequentially degrade the input images' quality, the number of successfully generated stickers should decrease monotonously.

MR2: $count(F(x_{i,0}, t)) \geq count(F(x_{i,1}, t) \geq \cdots \geq count(F(x_{i,m}, t))$, $1 \leq i \leq n$.

Due to the stochasticity, the monotonicity may be violated. So, we use a statistical criterion to verify it. We check MR2 using B-spline interpolation method [14]. We slightly modify it (https://gitlab.com/mlrep/mldev-metamorphic).

4.2. Experiment

We conduct an experiment in order to check MR1 and MR2.

For the MR1 we use 36 open-source portrait images of different ages and races. We try to create 100 stickers from every image and then try to create a sticker from every generated sticker. The number of stickers is determined automatically as the number of generated files. After that, we compare the number of generated stickers and number of stickers generated from stickers.

For the MR2 we use the same 36 images. We repeatedly apply gaussian noise to get a sequence of 30 gradually blurred images. Then, we try to generate 30 stickers from each image. We apply a criterion of monotonicity for each sequence to check if it decrease monotonously.

4.3. Results

Table 1 represents the result. MR1 fails completely and MR2 fails in a small percent of cases. This means that the system does not preserve faces well, but almost corresponds it's mathematical model.

Table 1

Results for SUT.

MR	Proportion of fulfilled MRs	Proportion of failed stickers
MR1	0.07	0.16
MR2	0.97	-

The experiment shows that SUT has major problems with face preserving and some minor issues with dependency on image quality. We can assume that NN components of the system work well, but the system in general has quality problems. So, proposed metamorphic relations are useful for testing and detecting problems.

4.4. Other observations

Our tests show that SUT changes input images a lot. So, if we use the output sticker as input, the second output will look less like the original image. If we repeat this many times, we will get an image with an unrecognizable face on it. Example of this transformation is provided on Figure 2.

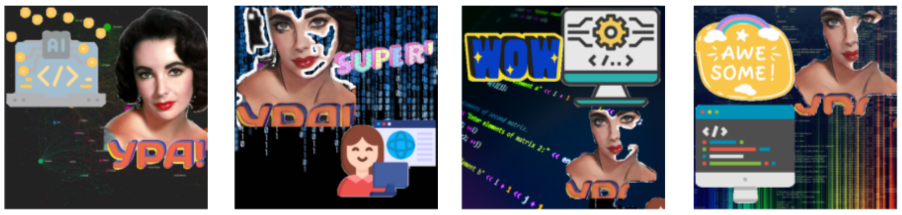

Fig. 2. 4 steps to get an image with unrecognizable face

Table 2 provides information about a small experiment with an image on Figure 1. We take the original image and lower its quality with three different methods. Then, we repeat sticker generation until possible.

This experiment shows that after 7 or 8 iterations all the stickers contain unrecognizable faces. So, we can estimate the number of steps to get such images. If we decrease image

Table 2

Number of stickers which are successfully generated from stickers

Number of repetitions	1	2	3	4	5	6	7	8	9
Original	100	79	39	21	7	1	0		
Darkened	100	82	37	16	6	2	1	0	
Low-contrast	100	81	36	15	5	2	0		
White-black	100	81	37	16	7	2	1	1	0

quality (blur, change the contrast, add color filter, etc.), the number of steps will decrease. So, potentially such series can be used for metamorphic testing too.

5. Conclusion

In this paper, we apply metamorphic testing to a generative AI system. We study the problem of AI testing, choose a generative system, propose two metamorphic relations, test the system and detect some major issues. Our metamorphic relations are derived from different groups of requirements: user and technical. Thus, we show that a metamorphic testing method is useful for verifying the requirements and testing such systems.

In this paper, we consider a relatively simple sticker generation system. Application to more complex systems could be a direction of future work.

References

1. Ahlgren J. et al. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) //IEEE, 140š149. – 2021.
2. Barr E. T. et al. The oracle problem in software testing: A survey //IEEE transactions on software engineering. 2014. Vol. 41, №. 5. P. 507–525.
3. Chen T. Y. et al. Metamorphic testing: A review of challenges and opportunities //ACM Computing Surveys (CSUR). 2018. Vol. 51, №. 1. P. 1–27.
4. Deason W. H. et al. A rule-based software test data generator //IEEE transactions on Knowledge and Data Engineering. 1991. Vol. 3, №. 1. P. 108–117.
5. Dwarakanath A. et al. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing //Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. 2018. P. 118–128.
6. Harbachonak, D.: Issledovaniye metodov mashinnogo obucheniya dlya sozdaniya personalizirovan-

nyh stikerov (Study of Machine Learning Methods for Personalized Stickers Creation). Proceedings of 65th MIPT Conference, Applied Mathematics and Informatics. 2023.

7. Harbachonak, D.: Issledovaniye metodov mashinnogo obucheniya dlya sozdaniya personalizirovannyh stikerov (Study of Machine Learning Methods for Personalized Stickers Creation). B.Sc. thesis, HSE University, Moscow (2023).
8. Last M., Friedman M., Kandel A. The data mining approach to automated software testing //Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. 2003. P. 388–396.
9. Meinke K., Niu F. A learning-based approach to unit testing of numerical software //IFIP International Conference on Testing Software and Systems. Berlin, Heidelberg : Springer Berlin Heidelberg. 2010. P. 221–235.
10. Pesu D. et al. A Monte Carlo method for metamorphic testing of machine translation services //Proceedings of the 3rd International Workshop on Metamorphic Testing. 2018. P. 38–45.
11. ur Rehman F., Izurieta C. Statistical Metamorphic Testing of Neural Network Based Intrusion Detection Systems //2021 IEEE International Conference on Cyber Security and Resilience (CSR). IEEE, 2021. P. 20–26.
12. Rombach R. et al. High-resolution image synthesis with latent diffusion models //Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2022. P. 10684–10695.
13. Tao C., Gao J., Wang T. Testing and quality validation for ai software–perspectives, issues, and practices //IEEE Access. 2019. Vol. 7. P. 120164–120175.
14. Wang J. C., Meyer M. C. Testing the monotonicity or convexity of a function using regression splines //Canadian Journal of Statistics. 2011. Vol. 39, №. 1. P. 89–107.
15. Xie X. et al. Testing and validating machine learning classifiers by metamorphic testing //Journal of Systems and Software. 2011. Vol. 84, №. 4. P. 544–558.
16. Zhang Z. et al. Deepbackground: Metamorphic testing for deep-learning-driven image recognition systems accompanied by background-relevance //Information and Software Technology. 2021. Vol. 140. P. 106701.
17. Zhou Z. Q., Tse T. H., Witheridge M. Metamorphic robustness testing: Exposing hidden defects in citation statistics and journal impact factors //IEEE Transactions on Software Engineering. 2019. Vol. 47, №. 6. P. 1164–1183.

UDK 004.4’232

Implementing a Language Server for the Rzk Proof Assistant

Abounegm A. (Innopolis University)

Kudasov N.D (Innopolis University)

The Riehl-Shulman type theory for synthetic ∞-categories is a new theory building on Homotopy Type Theory (HoTT). The experimental proof assistant Rzk offers an automated proof checker for this theory. With the goal of making this theory more accessible to mathematicians and computer scientists, we present in this paper a work-in-progress on a collection of command line and interactive tools for Rzk. These tools comprise a language server, an accompanying Visual Studio Code (VS Code) extension, and a list of smaller satellite tools offering minor conveniences. Although we focus on the support of VS Code, the language server is also compatible with other popular editors that support the Language Server Protocol (LSP), such as Emacs and Vim.

Keywords: Language Server Protocol, Visual Studio Code extension, proof assistant, homotopy type theory, category theory

1. Introduction

Proof assistants, also known as interactive theorem provers (ITPs), are software tools used in mathematics, computer science, and formal methods to assist in the development and verification of mathematical proofs. These tools play a crucial role in ensuring the correctness and reliability of complex mathematical statements and software systems. Examples of such tools include Agda [1], Coq [2], and Lean [3].

Although similar to typechecking in programming languages, proof checking is normally seen as an interactive process between the mathematician and a proof assistant. To this end, most proof assistants provide some interactive features either through specialized IDEs (e.g. CoqIDE[1]), integrating via Proof General [4], or Visual Studio Code (via the Language Server Protocol (LSP) [5]). Most well-established proof assistants support several of these options.

Rzk [6] is a new proof assistant that is based on Riehl-Shulman’s type theory for synthetic ∞-categories [7]. The proof assistant is experimental but has been successfully used recently to formalize some fundamental results for ∞-categories, including the ∞-categorical Yoneda

[1]`https://coq.inria.fr/refman/practical-tools/coqide.html`

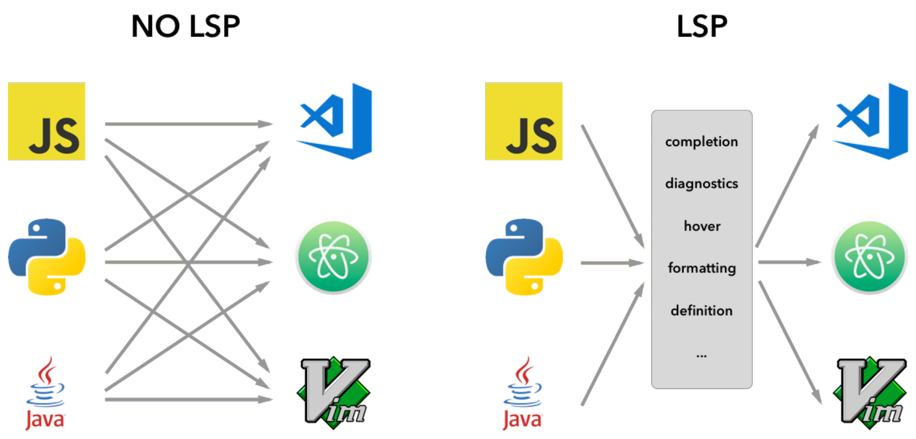

Fig. 1. The motivation behind LSP, from Microsoft's website. [2]

lemma [8].

1.1. Language Servers

A few years ago, when a new code editor was introduced, it needed to support the most popular programming languages, and depended on plugins written specifically for this editor to support languages not built into the editor. This led to a lot of repetitive work to provide useful language feature for the same language on multiple editors, and an inconsistency between the provided language features on different editors. The idea of language servers is Microsoft's attempt to solve this problem by standardizing a protocol for communication between a code editor (the client) and a background process (the server) that provides the language features for a certain language. This protocol is known as Language Server Protocol (LSP) [5]. With LSP, a language author need only implement the server once and would automatically get language support on any editor that supports LSP. Likewise, an editor that supports LSP would automatically get language support for any programming language that has an LSP server. Examples for the language features in question include providing diagnostic messages, jumping to the location an identifier is introduced, text completion, semantic syntax highlighting, and much more.

This protocol is particularly useful for interactive theorem provers since they rely on the editing experience much more than a command line interface. This is due to the fact that theorem provers generally do not need to compile to any kind of executable file and only need the type-checking stage of compilers, which can be easily performed by a language server. However, this also adds extra work on the language server since it needs to support more

[2] `https://code.visualstudio.com/api/language-extensions/language-server-extension-guide`

features than a regular compiler would, including listing variables in context along with their types, supporting Unicode symbols, and most importantly allowing for an interactive step-by-step proof walkthrough.

1.2. Related Work

Specification Language Server Protocol. While LSP has greatly improved the experience of developing language servers, it still leaves a bit to be desired. LSP was mainly designed with general purpose programming languages in mind, but theorem provers (or more generally, specification language) have slightly different requirements that are unmet by LSP. This is why [9] attempts to extend the original LSP specification with features that are especially useful for specification languages. The authors call the protocol extension *Specification Language Server Protocol* (SLSP). Simply speaking, it defines a set of new LSP requests/notification along with their payloads, and extends VS Code's interface with a view that displays proof information in a way similar to Proof General [4].

VS Code extension for Lean 4. Lean 4 [3] is a programming language and proof assistant by Microsoft Research[3] from which we draw much inspiration. In particular, the primary way to develop Lean programs is using its VS Code extension that provides a user interface for working with Lean interactively. This extension uses LSP to communicate with the Lean server and provides a lot of useful features, the most notable of which is the Info View panel that displays information about current proof state and allows interacting with it [10].

Proof General. Proof General [4] is a tool for developing interactive theorem provers that has been used for many widely-known proof assistants such as Coq and Isabelle. It is based on the Emacs editor and provides an interactive GUI with relative ease for theorem provers developed with its help.

1.3. Contribution

In this paper, we report on the work-in-progress on the implementation of utility and interactive tools around Rzk proof assistant, focusing on the language server and VS Code extension support.

[3]https://www.microsoft.com/en-us/research/project/lean/

2. Rzk Language Server and VS Code extesion

In this section, we describe design and implementation of the language server and the VS Code extension for Rzk proof assistant. The language server has direct access to the proof assistant internals, including the typechecking algorithm and the internal abstract syntax representation, and provides an interface conforming to the Language Server Protocol. The VS Code extension then acts as a buffer between the editor (VS Code) and the language server, to bring the interactive capabilities to the user.

We subject the language server to support the features specified below.

Intuitive Interface and Syntax Highlighting. The VS Code extension introduces an intuitive interface that aligns with the expectations of mathematicians and computer scientists. Users benefit from clear and accessible navigation, enabling efficient exploration of HoTT-based structures. Furthermore, the extension provides syntax and semantic highlighting, enhancing code readability, and facilitating error detection.

Code Completion and Suggestions. Rzk's VS Code extension leverages the LSP to offer intelligent code completion and context-aware suggestions. As users work with Rzk, the extension assists in writing code more efficiently by providing relevant suggestions, reducing the likelihood of syntax errors, and accelerating the development process.

Real-time Error Checking. One of the extension's notable strengths lies in its ability to perform real-time error checking. As users input and modify code, the language server continuously analyzes it, reporting back any type errors or other issues with the proof. This proactive error checking mechanism empowers users to identify and rectify issues promptly, fostering the creation of mathematically sound programs.

2.1. VS Code extension

To bring about these features to the users, a thin wrapper around the language server in the form of a VS Code extension is necessary. Additionally, the extension manages the installation of the language server itself on all major operating systems (Windows, macOS, and Ubuntu) powered by pre-built binaries attached to releases on GitHub, in addition to facilitating building the language server from source on platforms for which pre-built binaries are not available. It is also worth mentioning that the language server is designed to be compatible with any other editor that supports LSP. In particular, users have reported success in integrating it with the NeoVim editor, and it is planned to be tested with other editors as well.

2.2. Rzk Language Server

At the core of the Rzk tool suite is the language server that powers all the editor features that make it pleasant to develop proofs in Rzk. In particular, it currently supports semantic highlighting, diagnostic messages, and text completion. Additionally, progress reporting for long-running processes (such as type-checking for large projects) is currently under work.

For the first versions of the language server, it is shipped as part of the Rzk proof assistant itself under a different subcommand, but it is planned to decouple both components and have the language server depend on the core library. They are implemented in the Haskell programming language using the `lsp`[4] package.

3. Conclusion and Future Work

We have designed and implemented an initial prototype of the language server and VS Code extension for an experimental proof assistant Rzk. An intermediate result of this work has been presented at the *Interactions of Proof Assistants and Mathematics*[5] school in Germany in September of 2023. This has helped gather feedback from the users and react to it on the spot. We feel that current users are mostly satisfied with the prototype tooling, but have also provided useful suggestions for further improvements.

In the future, we plan to support displaying variable information on hover, jumping to definition, renaming symbols, and formatting Rzk code. Eventually, we also plan to support rendering topes as images, and adding an information WebView similar to the one provided by Lean 4 [10].

It is also anticipated that a Haskell library for developing language servers (especially for proof assistants) can grow out of this project.

References

1. Bove Ana, Dybjer Peter, Norell Ulf. A brief overview of Agda–a functional language with dependent types // Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22 / Springer. — 2009. — P. 73–78. — Access mode: `https://doi.org/10.1007/978-3-642-03359-9_6`.

[4]`https://hackage.haskell.org/package/lsp`

[5]`https://itp-school-2023.github.io/`

2. Bertot Yves, Castéran Pierre. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. — Springer Science & Business Media, 2013. — Access mode: `https://doi.org/10.1093/comjnl/bxh141`.
3. Moura Leonardo de, Ullrich Sebastian. The Lean 4 Theorem Prover and Programming Language // Automated Deduction – CADE 28 / ed. by Platzer André, Sutcliffe Geoff. — Cham : Springer International Publishing. — 2021. — P. 625–635.
4. Aspinall David. Proof General: A Generic Tool for Proof Development // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 2000. — P. 38–43.
5. Gunasinghe Nadeeshaan, Marcus Nipuna. Language Server Protocol and Implementation. — Apress, 2022.
6. Kudasov Nikolai. Rzk: a proof assistant for synthetic ∞-categories. — 2003. — Access mode: `https://github.com/rzk-lang/rzk`.
7. Riehl Emily, Shulman Michael. A type theory for synthetic ∞-categories // Higher Structures. — 2017.
8. Kudasov Nikolai, Riehl Emily, Weinberger Jonathan. Formalizing the ∞-categorical Yoneda lemma. — 2023. — 2309.08340.
9. The Specification Language Server Protocol: A Proposal for Standardised LSP Extensions / Rask Jonas Kjaer, Madsen Frederik Palludan, Battle Nick, Macedo Hugo Daniel, and Larsen Peter Gorm // Electronic Proceedings in Theoretical Computer Science. — 2021. — Aug. — Vol. 338. — P. 3–18.
10. Nawrocki Wojciech, Ayers Edward W., Ebner Gabriel. An Extensible User Interface for Lean 4 // 14th International Conference on Interactive Theorem Proving (ITP 2023). — 2023. — Vol. 268. — P. 24:1–24:20.