PSI Conference Series

PSI 19

12th A. P. Ershov Informatics Conference

Program Semantics, Specification and Verification:
Theory and Applications

July 1–2, 2019
Novosibirsk, Akademgorodok, Russia

Abstracts

A.P. Ershov Institute of Informatics Systems SB RAS
Ministry of Science and Higher Education of the Russian Federation
Novosibirsk State University

V. Zakharov, N. Shilov, I. Anureev (eds.)

X Workshop PSSV

Program Semantics, Specification and Verification: Theory and Applications

July 1–2, 2019, Novosibirsk, Akademgorodok, Russia

Abstracts

Organizers

A.P. Ershov Institute of Informatics Systems SB RAS
Novosibirsk State University

Sponsors

RFBR
Microsoft Research
Novosibirsk State University
STI International
STI Innsbruck
ARQA Technologies
Sibers

Novosibirsk
2019

UDK 519.68

X Workshop PSSV: Abstracts / Edited by V. Zakharov, N. Shilov, I. Anureev. — Novosibirsk: A.P. Ershov Institute of Informatics Systems, 2019. — 50 p.

ISBN 978-5-4437-0918-5

This volume comprises the papers chosen for presentation at the X Workshop PSSV to be held in Novosibirsk, Akademgorodok, Russia, on July 1–2, 2019. The main goal of the workshop is to give an overview of research directions in computer science.

В сборнике представлены аннотации пленарных, регулярных и коротких докладов, а также полные тексты стендовых докладов, включенных в программу X-го международного научно-исследовательского семинара «Программные семантики, спецификации и верификация», проводимого 1–2 июля 2019 г. в Новосибирском Академгородке. Тематика семинара охватывает следующие области: формализация программных семантик и спецификации программ, построение формальных моделей программ, применение логики для спецификации и верификации программ, автоматическое доказательство теорем, методы проверки моделей для программ, статический анализ программ, разработка формальных подходов к тестированию и валидации программ, средства для анализа и верификации программ. Полные тексты пленарных, регулярных и коротких докладов опубликованы в журнале “Системная Информатика” (Том 14, https://system-informatics.ru/issue/237).

The proceedings include invited, regular and short talks and extended abstracts of poster talks presented at the 10th International Workshop “Program Semantics, Specifications and Verification (PSSV-2019)” held in Novosibirsk Akademgorodok, Russia, on July 1–2, 2019. The Workshop covers such topics as formalisms for program semantics and specifications, logics for formal specification and verification, deductive program verification, automatic theorem proving, model checking of programs and systems, static analysis of programs, formal approach to testing and validation, program analysis and verification tools. Full texts of the invited, regular and short workshop talks are published in the journal “System Informatics” (Vol. 14, https://system-informatics.ru/issue/237).

UDK 519.68

ISBN 978-5-4437-0918-5

Steering Committee

Valery Nepomniaschy
Institute of Informatics Systems, Novosibirsk, Russia
Valery Sokolov
Yaroslavl State University, Russia

Program Committee Chairs

Nikolay Shilov
Innopolis University, Russia
Vladimir Zakharov
Moscow State University, Russia

Keynote Speakers

Alexei Lisitsa
Department of Computer Science, University of Liverpool, UK
Sergey P. Shary
Novosibirsk State University & Institute of Computational Technologies, Russia
Bin Fang
Huawei, China

Program Committee Members

Natasha Alechina, University of Nottingham, UK
Alexander Bolotov, University of Westminster, UK
Vladimir Itsykson, St. Petersburg State Polytech University, Russia
Igor Konnov, INRIA Nancy & LORIA, France
Victor Kuliamin, Institute for System Programming, Moscow, Russia
Alexei Lisitsa, University of Liverpool, UK
Irina Lomazova, Higher School of Economics, Moscow, Russia
Manuel Mazzara, Innopolis University, Russia
Valery Nepomniaschy, Institute of Informatics Systems, Novosibirsk, Russia
Valery Sokolov, Yaroslavl State University, Russia
Nina Yevtushenko, Tomsk State University & Institute for System Programming, Moscow, Russia

Additional Reviewers

Mikhail Belyaev

Author' Index

Preface

The volume contains the papers selected for presentation at the X International Workshop on Program Semantics, Specification and Verification: Theory and Applications (PSSV-2019). The Workshop took place on July 1–2, 2019, in Novosibirsk, Akademgorodok, Russia. PSSV Workshops were successfully held in Kazan (2010, 2015), St. Petersburg (2011, 2016), Nizhny Novgorod (2012), Ekaterinburg (2013), Moscow (2014, 2017) and Yaroslavl (2018). In 2010–14 and 2016 PSSV Workshops were affiliated with the International Symposium "Computer Science in Russia" (CSR); in 2015, 2017 and 2019 it was affiliated with the A.P. Ershov Informatics Conference (the PSI Conference Series). The topics of the Workshop include formal models of programs and systems, methods of formal semantics of programming languages, formal specification languages, methods of deductive program verification, model checking method, static analysis of programs, formal approaches to testing and validation, program testing, analysis and verification tools. In 2018 the Workshop was dedicated to the memory of B.A. Trakhtenbrot (1921–2016), M.I. Dekhtyar (1946–2018), and M.K. Valiev (1942–2018). Thirteen research papers have been submitted to the PSSV-2019. Program Committee accepted 3 papers as regular ones, 3 as short presentations, and 3 more papers — for the poster session. Abstracts of 3 invited talks are also included in this collection of abstracts.

The Program Committee work was done using the EasyChair conference management system.

June 2019

Vladimir Zakharov
Nikolay Shilov
Igor Anureev

Table of Contents

The JaSpe Specializer:
An Interactive Approach to Metacomputation

Igor A. Adamovich (Ailamazyan Program Systems Institute Russian Academy of Sciences, Yaroslavl region, Russia, i.a.adamovich@gmail.com)

The article presents the results of research that belongs to a scientific field usually called metacomputation, which includes program specialization as well as other deep program analysis and transformation techniques. This study develops such kind of specialization as partial evaluation in application to the widely used object-oriented Java language in the popular Eclipse development environment. The article also presents an approach to organizing an interactive human-computer dialogue between a programmer and a specialization subsystem.

The main difficulty of putting metacomputation into practice is that it does not specify a single automatic mode for optimizing programs but contains too many possibilities, which the computer cannot do the right choice of. This is true for supercompilation, partial evaluation, partial deduction and other metacomputation methods. A well-developed interactive system is required. In such a system the computer will guarantee the equivalence of program transformations and the human will provide information on how to choose transformations among the acceptable degrees of freedom.

Based on the above considerations the interactive specializer referred to as JaSpe has been designed. It uses the partial evaluation method. JaSpe is embedded in the Eclipse development environment and specializes object-oriented Java programs.

This paper describes the approach to interactive specialization implemented in JaSpe. An example of specialization of a program with objects and optimization of objects in the heap is given.

The JaSpe specializer is not yet complete. This is a work-in-progress report showing the current state of development of methods and implementation.

1. Introduction

Specialization [4] is a program optimization method based on the use of predefined information about the values of a part of variables. Let a two-argument program $f(x, y)$ be given and let the value of first argument x is equal to a. The result of the $f(x, y)$ program specialization with respect

to the known argument $x = a$ is a new program $g(y)$ of one argument, which has the following property: $f(a, y) = g(y)$ for each y.

The main difficulty of putting program specialization as well as metacomputation methods in general (supercompilation, partial evaluation, partial deduction, and other methods) into practice is that it does not specify a single automatic mode for optimizing programs. Methods of metacomputation contain too many possibilities, so that the computer cannot choose of them by means of reasonably complicated algorithmic methods. A well-developed interactive system is required. In such a system the computer will guarantee the equivalence of program transformations and the human will provide to the system some information on how to choose specific transformations from the acceptable variants.

According to these considerations, the goal was put forward to implement an interactive specializer of Java language.

Interactive specialization is the user work cycle of program optimization using a specializer. The cycle lasts until the user is satisfied (the program has been optimized reasonably well). The cycle consists of the following stages:

1. Writing a program.
2. Applying the specializer to the program.
3. Studying the residual program, comparing with the source program and obtaining various inside information about specialization process.
4. Changing the program.
5. Changing the specialization task (options of specializer).
6. Repeating from step 2.

To successfully fulfill this task, interactive support from the specializer is required. The more interactive the specializer is, the higher is level of the interactivity of the whole cycle.

Therefore, to solve the problem of introducing metacomputation into practice by the approach described above, it is required to develop and implement the features of interaction between a specializer and a human using graphical user interface (GUI).

This paper manifests the need of human-computer interaction in program specialization process by case study of implementation of the partial evaluator JaSpe for the object-oriented language Java. The specializer is embedded in the familiar Eclipse integrated development environment (IDE) [3]. The specializer presented in this paper is an extension of the specializer presented in [1]. The main areas of extension are polyvariant BTA and optimization of programs parts that manipulate objects.

In Section 2 the basic notions of partial evaluation are introduced, and high-level description of used algorithm is given. In Section 3 basic interactive features of presented specializer are specified. In Section 4 case of specialization is discussed. Section 5 contains a survey of related works in comparison with our specializer. In Section 6 we conclude.

The reported study was funded by RFBR according to the research project № 18-37-00454.

2. Partial Evaluation in JaSpe

Partial evaluation is one of program specialization techniques. Specialization by the method of partial evaluation [4] separates program constructs into *static* (depending on known data) and *dynamic* (depending on unknown data). Operations depending on static data are executed and operations depending on dynamic data are transferred to the resulting (*residual*) program. The residual program depends only on unknown (at the specialization phase) data.

The first stage of partial evaluation is a kind of static analysis called *binding time analysis* (BT-analysis, BTA). At this stage, separation of operations and data into static and dynamic is performed. The second stage is responsible for the execution of the static part of the program and the transferring of the dynamic part to a separate program. This stage is called *residual program generation* (RPG).

Terminological remark. The term *static* conflicts with the `static` modifier in Java, and the term *dynamic* can be confused with a runtime notion. Therefore, we avoid using these words in relation to partial evaluation and use the symbols S and D, for example, S-annotation, D-annotation, S-code, D-code, S-part and D-part of a program.

The outline of the partial evaluation method with monovariant BTA used in the previous version of JaSpe is given in paper [1]. This paper presents the main points and their extension to polyvariant BTA. Definitions of *polyvariance* and its types can be read in [6]. The polyvariant BTA algorithm part working with variables and operations of primitive types is similar to the previous version, the only difference is that there are no conflicts caused by different annotation of the same variables and method invocations at different points in the program. Another part of BTA algorithm that works with objects is new compared to previous version. Annotation of an object is composed of annotation of object itself (we call it *object top annotation*) and annotations of this object fields. An object with D object top annotation always has D-annotations of all of its fields. Annotations of objects is monovariant — each object has one composed annotation after BTA for entire program. Thus, BTA algorithm works as follows. First, all constants are annotated with S. Then recursively: subexpressions containing only S-parts become S; annotation of object creation expression results of creation of an S-object with S-fields; local variable declarations and assignments with S right-

hand sides become S; a conflict on several assignments to a field of an object with different annotations turns it to D; for each method invocation new version of the method body annotation is created; an if statement with the S conditional expression is annotated with S regardless of the annotation of its branches (this means that if-else will disappear while one of the branches will be residualized); other control statements are analyzed and annotated similarly. When this recursive procedure is finished, the remaining parts of code are annotated with D.

It should be noted that described BTA algorithm does not always terminate. For example, the analyzed program can loop infinitely creating new objects, while each new object can refer to the object created at the next iteration. Thus, a potential infinite chain of objects is created, each of which needs to be annotated. In this case, the annotation of the first object in the chain includes annotation of all subsequent ones. This problem has different solutions. First of all, it can be solved my methods described in [6]. Second solution is that analysis could be terminated by a user if it is executed too long. Such termination can be supported by JaSpe by IDE interface.

Another limitation of the described method is common for partial evaluation: the partial evaluator does not optimize a program when there is a lack of S-values in code.

3. Interactive features of JaSpe

First of all, it should be mentioned that JaSpe developers expect that the main users of the specializer will be ordinary Java-programmers who want to optimize their programs. However, the specializer does not always cope with the task automatically, without user interaction, often the program needs to be slightly rewritten in order to be better optimized. Highlighting of the binding time analysis results was implemented to make it clear how to rewrite the program. The latter correlates with the definition of interactive specialization given in chapter 1 of this paper.

In the JaSpe specializer the code that has been analyzed by BTA is automatically marked using the highlighting of the S- and D-parts of the code. This allows the user-programmer to better understand how the process of specialization proceeded and the reasons for the specific residual program generation.

In principle, the process of marking the code can occur on the fly synchronously with the code modifications made by the user or after the user presses the special “button”. Currently, the second option is implemented in the JaSpe specializer. The implementation of synchronous highlighting requires a lot of effort and most likely will not be considered at the research stage, but at the stage of preparing a specializer for mass usage.

In order to create a specialization task for the JaSpe specializer, the user must mark the methods that are the *entry points* of the specialization with the @Specialize Java annotation. The specialization procedure begins with such entry points.

The result of a specialization is a modified version of the source program and is called the residual program. After receiving the residual program, it is needed to place it somewhere and preferably so that the appropriate executable code can be easily obtained. The study proposes two solutions to this problem. The first solution is to save the modified files in the same project where the source files are, but in a different directory and in another Java package (package). This solution is simpler to implement, but it is less flexible. An alternative solution proposed in this study is to create a new Eclipse project where modified files are saved in packages with the original names.

4. Example of specialization by JaSpe

In this section, we consider an example of a specialization of a function `consProcessing` that applies the Cons operation three times to an integer list passed to this function as an argument. Then the function traverses the result and sums all the numbers stored in the result list. The source code with highlighting of the result of the binding time analysis is presented in Figure 1. Figure 2 shows the Cons class and Figure 3 shows the part of the residual program corresponding to the source function `consProcessing`.

When JaSpe highlights code on screen the colors are a good solution but in case of black and white paper we use other opportunities of tunable interface of JaSpe. Underscored text on Figure 1 corresponds to the S-parts of the code, text in black boxes corresponds to D-parts and gray background highlighting corresponds to mixed parts. JaSpe is a polyvariant specializer: some parts of code can have several BT-annotations. For example, the while loop on the Figure 1 at the first and second iterations has two BT-annotations containing only S-code. These annotations correspond to the objects created and stored in variables `c2` and `c1`. At the 4th and subsequent iterations the while loop has a fully D-annotation corresponding to the BT-annotation of the `argCons` argument. In more detail, definitions of *polyvariance* and its types can be read in [6].

In accordance with the classification given in Yuri Klimov's PhD thesis [6], the JaSpe specializer is polyvariant in variables, operations, methods and classes.

Highlighting the BT-annotation of the program is the most important part of the interface of the specializer, which helps the programmer to better understand why his program was optimized in this specific way and not otherwise.

```
@Specialize
public static final int consProcessing(Cons argCons) {
    Cons c0 = new Cons();
    c0.tail = argCons;
    c0.head = 1;

    Cons c1 = new Cons();
    c1.tail = c0;
    c1.head = 2;

    Cons c2 = new Cons();
    c2.tail = c1;
    c2.head = 6;

    Cons cons = c2;

    int acc = 0;
    while(cons != null) {
        acc += cons.head;
        cons = cons.tail;
    }

    return acc;
}
```

Figure 1. Source function with highlighting

```
public class Cons {
    public int head;
    public Cons tail;
}
```

Figure 2. Cons class

Figure 3 shows the part of the residual program obtained from the source function `consProcessing` presented in Figure 1. It should be noted that the source code corresponding to the S-code (underscored text) is almost completely absent. Also, the first three iterations of the while loop have been executed during specialization. An important feature of the residual code is that the first three objects of the `cons` variable, which were annotated as S, are completely absent in the runtime heap. That is, in addition to increasing the speed of the program, the memory consumption has been decreased.

It should be mentioned that discussing example could model stack machine based on list, that, for instance, interprets an AST. Some of the nodes could be known (nodes that corresponds to S-objects referenced by `c0`, `c1`, `c2`) and some nodes could be unknows (D-objects referenced by argCons).

```
public static final int consProcessing(Cons argCons) {
    Cons c0tail2;
    c0tail2 = argCons;
    Cons cons;
    int acc = 0;
    {
        {
            cons = c0tail2;
        }
        acc = 9;
        while (cons != null) {
            acc += cons.head;
            cons = cons.tail;
        }
    }
    return acc;
}
```

Figure 3. Result of specialization

Table 1. Acceleration by specialization

Size of argCons, number of items	Acceleration, times
0	32
1	10
2	8
10	2.21
100	1.08

For completeness, Table 1 shows the acceleration that were acquired as result of specialization depending on the size of the `argCons` argument. From the presented values it is clear that the specializer is able to give a very noticeable acceleration. The acceleration source is the reduction of time for memory allocation of three objects that are added to Cons. The allocation time for a constant number of objects is constant, therefore, with an increase in the number of objects in the cons, the fraction of saved time decreases, and hence the acceleration.

5. Related work

The partial evaluation method was invented more than 30 years ago. At the first period, it was developed mainly for functional languages. Book [4] summarizes this wave of research.

Then there was the development of the method of partial evaluation in the application to imperative languages. This development resulted in the creation of several specializers, the most developed of which is the Tempo specializer [2, 11].

An important innovation of Tempo from the interactivity point of view were code coloring in accordance with the result of BT-analysis. However, in Tempo the coloring was produced to a separate html-file, not in the editor included in the IDE as in JaSpe.

The rest of Tempo interface is similar to other specializers in the sense that it uses command line and configuration files to compose a specialization task.

For object-oriented languages, two important specializers have been developed. The first is JSpec, a Java language specializer developed by Ulrik Pagh Schultz in France [12, 13]. This specializer contains significant limitations, in particular, it works only with immutable objects. However, the creator proposed several interesting principles of interaction with user.

In Schultz's publications on the specialization of the Java language, it was proposed to create a task of specialization using the so-called "classes of specialization". The approach used in the JaSpe specializer differs from the Schultz approach, being no less expressive, but simpler to understand, use and implement.

Also, in Schulz's works it was proposed to compose the code obtained as a result of specialization into the so-called "aspects". Aspects according to Schulz contain specialized versions of the methods and are "woven" with the source program using the pass of the special program called weaver. Another approach to deployment of residual code proposed in our paper (creating a new project for the residual code) has no limitations compared to the Schultz variant. But our approach allows the user to better compare the residual code with the source one (since JaSpe preserves the structure of the program).

Another important specializer is the CILPE specializer, developed by Yu. A. Klimov in 2009 at Keldysh Institute of Applied Mathematics of RAS [5-10]. This specializer most deeply develops the theory of partial evaluation for object-oriented languages and applies it to the object-oriented Common Intermediate Language, the Microsoft .NET Framework bytecode. This specializer supports almost all the basic constructs of object-oriented languages, like C # or Java. From the interface point of view, the CILPE specializer basically used the command line to interact with the user. The idea of specifying the entry points of the specialization using Java annotations was borrowed by JaSpe from the specializer CILPE.

Among the recent advances in the field of partial evaluation, the GraalVM toolkit [14, 15] should be mentioned. It is developed in Oracle Labs and includes a partial evaluator. However, this partial evaluator does not stand out in the interactivity direction, but nevertheless it is an important achievement in the practical application of metacomputation.

6. Conclusion

This paper argues the need for an interactive approach to metacomputation for the widespread use of these methods of deep program transformation. The non-formal definition of interactive specialization is presented – this is the cycle of optimization of a program. A solution is proposed in the form of a the JaSpe specializer implemented in the Eclipse IDE. Short high-level description of a partial evaluation method implemented in JaSpe is given. The features of interaction between a specializer and a human using graphical user interface implemented in the JaSpe specializer are presented.

We illustrate the work of the JaSpe by applying the specializer to an example program written in the object-oriented Java language. The program obtained as a result of specialization works much faster than the original one.

We see the following directions for further development of the specializer:

- to implement support for all Java 8 constructs;

- to implement additional interactive tools for composing a specialization task and controlling the process of binding-time analysis and residual program generation;
- to implement tools to visualize the correspondence between source and residual code;
- to demonstrate that a well-developed specializer can convert well-structured high-level human-oriented code, which cannot be automatically parallelized, into code that can be parallelized by existing methods and tools.

Availability. Examples and the previous version of JaSpe with monovariant BTA and without optimizations of objects are available at ftp://ftp.botik.ru/rented/iaadamovich/Specializer/.

References

1. Adamovich I.A., Klimov And, V. An Interactive Specializer Based on Partial Evaluation for a Java Subset // The Proceedings of ISP RAS. 2018. vol. 30. № 4. pp. 29-44.
2. Consel C., Lawall J.L., and Meur A.-F.L. A tour of Tempo: a program specializer for the C language // Sci. Comput. Program. 2004. Vol. 52. № 1-3. pp. 341–370.
3. Eclipse Foundation // Eclipse Integrated Development Environment (IDE), URL: https://www.eclipse.org (last accessed 9.06.2019).
4. Jones N.D., Gomard C.K., and Sestoft P. Partial Evaluation and Automatic Program Generation // Prentice-Hall. 1993. 415 p.
5. Klimov Yu.A. [SOOL: an object-oriented stack-based language for specification and implementation of program specialization techniques], 2008. 32 p. (Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints]; № 44).
6. Klimov Yu.A. [Specialization of programs in object-oriented languages by partial evaluation]: Ph.D. thesis. Moscow: Keldysh Institute of Applied Mathematics of RAS, 2009. 183 p.
7. Klimov Yu.A. [Specializer CILPE: binding time analysis], 2009. 28 p. (Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints]; № 7).
8. Klimov Yu.A. [Specializer CILPE: examples of object-oriented program specialization], 2008. 28 p. (Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints]; № 30).
9. Klimov Yu.A.: [Specializer CILPE: partial evaluation for object-oriented languages]. // Programmny`e sistemy`: teoriia i prilozheniia [Program Systems: Theory and Applications]. 2010. № 3(3). pp. 13–36.
10. Klimov Yu.A. [Specializer CILPE: residual program generation], 2009. 26 p. (Preprinty` IPM im. M.V. Keldy`sha [Keldysh Institute Preprints]; № 8).
11. Renaud M. Program Specialization // Wiley-ISTE. 2012. 554 p.
12. Schultz U.P., Lawall J.L. and Consel C. Automatic program specialization for Java // ACM Trans. Program. Lang. Syst. 2003. vol. 25. № 4. pp. 452–499.
13. Schultz U.P. Object-Oriented Software Engineering Using Partial Evaluation: PhD thesis. Rennes: University of Rennes I, 2000. 215 p.

14. Würthinger T., Wimmer C., Humer C., Wöß A., Stadler L., Seaton C., Duboscq G., Simon D., and Grimmer M.: Practical partial evaluation for high-performance dynamic language runtimes // SIGPLAN Not. 2017. vol. 52. № 6. pp. 662–676.
15. Würthinger T., Wimmer C., Wöß A., Stadler L., Duboscq G., Humer C., Richards G., Simon D., and Wolczko M. One VM to rule them all // Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! ACM. 2013. pp. 187– 204.

Building Cyclic Data in a Functional-Like Language Extended with Monotonic Objects

Alexei I. Adamovich (Ailamazyan Program Systems Institute of RAS, lexa@adam.botik.ru),
Andrei V. Klimov (Keldysh Institute of Applied Mathematics of RAS, andrei@klimov.net)

We discuss an approach to deterministic parallel programming based on a two-level programming language. It comprises a higher-level functional-like subset for application programmers and a lower-level object-oriented Java-like language for experts in parallel programming, who develop libraries of classes. The experts guarantee determinism for the higher-level language user. We refer to these classes and objects as *monotonic* and give their definition as preserving two properties of the higher-level programs: determinism and idempotency. In this case study, we address the problem of representing cyclic data structures, which is unsolvable in purely functional languages, easy in object-oriented languages, and solvable but tricky with monotonic objects. As an introductory example, a simple monotonic class is given—a variable of a primitive type. Then we show that an analogous class declaration for a variable of a reference type is non-monotonic and reveal that building cyclic data structures in this setting is nontrivial. Finally, we present examples that demonstrate some of the subtleties of designing monotonic classes. This paper describes a work-in-progress towards a theory and a software system for deterministic parallel programming. It also poses new problems for program verification to assist in proving that class declarations are monotonic and, therefore, parallel programs developed in the proposed system are proved deterministic.

1. Introduction and Related Work

Parallel/concurrent programming and debugging are complicated because parallel/concurrent programs are nondeterministic in the general case. To simplify programming, various specific model of parallelism and concurrency have been, and are continuously being, invented. Some of them focus on the determinism of the results of computation, where all runs result with equivalent final states despite of interleaving and different ordering of operators from different parallel/concurrent threads.

Please refer to our recent paper [5] for a survey of an extensive field of research on deterministic parallel/concurrent programming. Let us list below just some of these works which are the most interesting for our study.

The most restricted models are those based on pure functional programming, where side effects are absent, threads are independent of each other, and hence the results are always the same. Not surprisingly, active research on deterministic parallelism is carried out in the community of the purely functional language Haskell [10]. Our work is in fact a transfer of the ideas of the functional paradigm to object-oriented languages, while retaining some important properties in a more general form.

In the object-oriented setting, the work on Deterministic Parallel Java (DPJ) [6,7,11] extends the Java language and the compiler by adding certain features that guarantee determinism using an analysis that checks that parallel threads interfere only in a disciplined way, which does not violate determinism.

The most interesting from our point of view are results by Lindsey Kuper *et al*. [12,13,14,15] where she suggests that variables shared between threads should change monotonically in some partially ordered set (more precisely, a semilattice) and operations on them are defined in such a way that the result of computation is deterministic. In our work, we use this idea in a more general object-oriented setting.

The rest of the paper is organized as follows. In Section 2 we introduce the idea of a two-level programming language and system. In Section 3 the notion of a monotonic class and object is defined. In Section 4 an introductory example of a monotonic class is given. Sections 2–4 are based on our previous work [4,8,9]. In Section 5 we present novel material: using some examples we discuss problems and solutions on how to build cycle data structures using monotonic objects, thereby overcoming the limitations of purely functional programming, while preserving the determinism of parallel computation.

2. A Two-Level Programming Language and System

In order to meet controversial requirements of program determinism for application programmers and diversity of deterministic parallel computation models, we suggest the use of (and develop) a two-level programming language and system:

- The higher-level language is like a functional subset of Java with simple means to initiate new threads on function calls and method invocations (often called “futures”, “promises”[1]).

[1] https://en.wikipedia.org/wiki/Futures_and_promises

Threads create objects of classes declared at the lower level and communicate only through operations on them. Any syntactically correct program here is deterministic by construction. This higher-level language is intended for use by the application developers.

- The lower-level language is full Java, or any other similar object-oriented language, which comprises a complete set of means for concurrent programming of generally nondeterministic programs. In this language, experts in parallel and concurrent programming declare lower-level classes in such a way that their use in the higher-level language guarantees determinism and preserves other valuable properties discussed below. We refer to these classes and their objects as *monotonic*.

Notice a subtlety of this definition: the declarations of monotonic classes belong to the lower level, while their monotonicity is defined through the properties of programs in the higher-level language, rather than the properties of the classes *per se* like pre- and post-conditions of their methods, object invariants, *etc*.

The development of this language and system is based on (and continues) our previous research into parallel programming systems [1,2,3,4].

3. The Notion of a Monotonic Class and Object

To formally articulate what determinism means we need the notion of equivalence of computations. We use the Leibniz notion of contextual equivalence.

Definition 1 (equivalence of values). Two values are (*contextually*) *equivalent* if they are indistinguishable programmatically in the higher-level language, that is:

- the values are of the same type, and
- in the case of a primitive type: the values are equal, and
- in the case of a reference type: any function or method with a primitive result type, when applied to these values, returns equal results, or neither terminate. □

Definition 2 (equivalence of computations). Two executions of copies of an expression (with the copies of arguments) are (*contextually*) *equivalent*, if

- they both terminate and return equivalent values, or
- they both throw exceptions (which may be different), or
- neither terminate. □

Definition 3 (monotonic). A declaration of a class and the objects of the class are called *monotonic* if any program in the higher-level language using the operation of object creation and methods on the objects of this class, satisfies the following properties:

- *Determinism* of computation (or *confluence*), that is, the results obtained in different order of a parallel/concurrent computation are equivalent.
- *Idempotency*, that is, a repeated computation of a copy of an expression is equivalent to the original computation, and the difference between the side effects of the two runs is not observable programmatically in the higher-level language.

These properties must be satisfied simultaneously for all monotonic classes when they are used together in any program in the higher-level (functional-like) language. □

One may wonder about motivation behind the idempotency property. It may seem that idempotency is implied by determinism. However, in the definition of determinism, to compare the results of different computation order, repeated runs are performed starting from the same initial state, while in the definition of idempotency, the next run uses the final state of the previous one.

```
public class IntVar {

  boolean defined = false;
  int value;

  public synchronized int get()
  {
    if (!defined) wait();
    return value;
  }

  public synchronized void set(int x)
  {
    if (!defined) {
      value = x;
      defined = true;
      notifyAll();
    }
    else if (value != x)
      throw new RuntimeException();
  }
}
```

Fig. 1. Monotonic class `IntVar` with one field of the primitive type `int`.

```
public class ObjectVar {

  boolean defined = false;
  Object value;

  public synchronized Object get()
  {
    if (!defined) wait();
    return value;
  }

  public synchronized void set(Object x)
  {
    if (!defined) {
      value = x;
      defined = true;
      notifyAll();
    }
    else if (value != x)
      throw new RuntimeException();
  }
}
```

Fig. 2. Non-monotonic class `ObjectVar` with one field of the reference type `Object`.

4. A Simple Monotonic Class Example

Parallel threads communicate by means of side effects on variables of various types. Our goal is to construct monotonic versions of classes representing such variables.

Consider the case where a variable is of primitive type `int`. The monotonic class `IntVar` shown in Fig. 1 declares 2 methods `set` and `get` with the following semantics:

- `set(x)` stores the value `x`, if an unequal value has not been stored already; otherwise throws an exception;

- `get()` returns the value stored by `set()`, or waits until `set` has been invoked, and then completes.

Thus, each `IntVar` object monotonically changes from the undefined state to the state defined with some integer and then possibly to raising an exception, which may be read as the "overdefined" state. We argue that such behavior satisfies the definition of monotonic objects. This is explained in more detail in our earlier paper [9].

5. Building Cyclic Data Structures

Notice that it is principally impossible to build a cyclic data structure without using mutable data. That is why purely functional languages allow us to efficiently manipulate only trees. This prohibits development of high-performance software that manipulates graphs, which imperative and object-oriented languages allow. In order to efficiently manipulate graphs, where cyclic relations between vertices and edges are denoted by references, one needs to mutate the representation. Our goal is to allow mutable data *and* preserve the main properties of functional languages, which we capture in the notion of monotonic objects.

Let us study various examples of declarations of mutating operations on objects and see which of them are monotonic and which are not.

Example 1: non-monotonic. Let us change in Fig. 1 the `value` field from the primitive to reference type. Figure 2 shows the code of the class `ObjectVar`, which coincides with the monotonic class `IntVar`, except for the type `Object` instead of `int`. However, this makes the class non-monotonic. The cause of this is the lack of *referential transparency* of the `new` operator as it generates a new reference to a new object on each evaluation, which fundamentally differs from the world of functional programming. Consider the following code fragment:

```
ObjectVar a = new ObjectVar();
a.set(new ObjectVar());    // first evaluation of an expression
a.set(new ObjectVar());    // second evaluation of the same expression
return a.get();
```

The notion of monotonicity requires that reevaluation of an expression returns an equivalent result and is idempotent with respect to side effects (that is, nothing changes). However, in the second invocation of the method `set`, the condition `(value != x)` in its body evaluates to `true` and the exception `RuntimeException` is thrown.

Example 2: non-monotonic. A natural idea to avoid this unpleasant exception is to replace the comparison of references with the comparison of the objects' contents by the method `equal`:

```
if (!value.equal(x)) throw new RuntimeException();
```

The method `equal` should perform deep comparison, paying no attention to the equality of references, except for the sake of optimization, which is invisible from the outside, and in order to avoid looping when traversing cyclic data.

Nevertheless, the class `ObjectVar`, thus defined, is still non-monotonic, because the result of `a.get()` depends on the order of evaluation of the two `new` expressions. We imply that the statements `a.set(new ObjectVar())` can be computed in parallel, hence the result can be either the reference to the object created by the first `new` sub-expression, or by the second one. This difference is programmatically visible.

Example 3: monotonic. To fix this, we must avoid returning from monotonic objects (by methods like `get`) the references passed in arguments of any of its methods (like `set`). This can be done in two ways. First, a clone of the stored object can be created in `set`, the reference to which is then returned by all invocations of `get`. Second, a new clone of the stored object can be generated on each invocation of `get`. The first version seems more efficient (in terms of the memory for extra objects generated in the second version). However, either of these versions could be useful, depending on the application. A library of monotonic classes should contain both versions, with different names.

However, this does not complete the definition of the monotonic `ObjectVar` semantics. The objects in the argument of the `set` method could be undefined, or partially defined. The latter case may occur when the object has many fields, and some of them are already defined, while others are not. Thus, a kind of unification of the objects from the arguments of several invocations of `set` is required, the result of the unification being stored in the `ObjectVar` object. Now, if properly formalized and coded, the class `ObjectVar` becomes monotonic.

Nevertheless, some degrees of freedom preserving monotonicity remain. Should the unified objects be changed as well? Should the information about the unified objects flow in one direction from the `set` arguments to the stored object only, or could it flow in the opposite direction as well? The answer is that both versions are monotonic, and their usefulness depends on the application.

Example 4: building a cycle. Now we can build a cycle of length 1 from an object of the class `ObjectVar` with the first version of the monotonic class semantics discussed in Example 3:

```
ObjectVar a = new ObjectVar();
a.set(a);    // a cycle is built
b = a.get();
c = b.get(); // c != b
d = c.get(); // d != b && d!= c
```

Notice that to preserve monotonicity according to the above semantics, different references are returned in variable `a`, `b`, and `c`. Thus, we met a problem that, although the cycle was built, it can never be recognized programmatically, while traversing the object structure. For some applications this may be appropriate, for example, when a graph is the representation of a finite automata, which is used only in its interpreter. But if, for example, we wanted to print the automata representation, we would not be able to write a terminating code.

Example 5: a cyclic graph with a finite number of references to vertices. Although we don't know how to represent and traverse graphs having access to the unique references to edges in the world of monotonic objects as easily as we do in object-oriented languages, there are particular solutions to this problem. Consider two of them.

Example 5a: Imagine a factory method that simultaneously creates a given number of objects and returns an immutable vector of the references to them. Its signature may be like this:

```
Vector<ObjectVar> createObjectVars(int n);
```

Then let us modify the class `ObjectVar` in such a way that its objects "know brothers", that is, each object has access to this vector, and the `get` method returns only references to the "brothers" and to the object itself. Classes with such semantics allow us to build an efficient representation of an arbitrary finite graph. We argue that such class declarations are monotonic.

Example 5b: Let us return to the monotonic class declaration in Examples 3 and 4 and use it for further modification. Let us prohibit returning reference values from monotonic objects until the whole of the deep structure accessible from the given object is fully defined (and hence, is finite). Then let us minimize the graph representation by means of the well-known algorithm of minimization of a finite automaton. We argue that the unique references to the objects of the minimal representation can now be returned by the methods like `get`, preserving monotonicity. There are a finite number of references to the accessible objects and while traversing the graph structure, we can programmatically catch the cycles.

More parallelism by suspending the equality checks and exceptions. One more subtlety that can limit parallelism is that the method `equal` (in the `if` statement of Example 2) cannot return `true` until the compared objects become fully defined. Such blocking of the computation is highly undesirable. Fortunately, there is an escape. The definition of monotonicity does not distinguish various exceptions raised in different program points. All exceptions are equivalent as the result of computation. This gives us a possibility to suspend execution of such `equal` predicates along with the surrounding `if` statements and immediately return from the `set` method. If, in the end, when the compared objects become defined, `equal` returns `true`, the suspended statement completes with no

effect. If a difference is found and `equal` returns `false`, the exception is raised from the suspended statement and propagated as the resulting exception of the whole computation. This behavior is monotonic according to our definition.

6. Acknowledgement

We express our gratitude to our English teacher, Philippa Jephson, who helped us edit this paper, fix a lot of errors and turn it into a much more readable form. The remaining glitches are ours.

7. Conclusion

The notion of a monotonic class and a monotonic object was presented. It was defined so that the use of monotonic objects in a program in a functional-like language with maximal parallelism preserves the main properties of functional languages: determinism of computation results (confluence) and possibility of repeated computations with the same result and side effect (idempotency). We use the term *monotonic* with the idea that a monotonic object changes in a certain (semi)lattice that can be derived from its class declaration.

Unlike functional languages, building cyclic data structures is possible with the use of monotonic classes, although their semantics and code are nontrivial and tricky, and impose certain overheads compared to common object-oriented programming. Further development of the methods of their efficient implementation is required taking into consideration special cases and using metacomputation methods like program specialization. Currently, we are prototyping an implementation of a language with monotonic classes.

The demonstrated examples show that it is not at all obvious whether a class declaration is monotonic or not. Designing such a class is like finding a nontrivial solution to an “equation” that is the property of monotonicity. Formal means (theory and software tools) are highly desirable in helping us prove monotonicity. This is an interesting program verification topic for future work.

References

1. Abramov S.M., Adamovich A.I., Kovalenko M.R. T-System—An Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Ray Tracing Method // Programmirovaniye, 25:2. 1999. P. 100–107. (In Russian).
2. Adamovich A.I. Fibers as the Basis for the Implementation of the Notion of the T-Process for the JVM Platform // Program Systems: Theory and Applications, 6:4 (27). 2015. P. 177–195. URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf. (In Russian).

3. Adamovich A.I. The Ajl Programming Language: The Automatic Dynamic Parallelization for the JVM Platform // Program Systems: Theory and Applications, 7:4 (31). 2016. P. 83–117. URL: http://psta.psiras.ru/read/psta2015_4_177-195.pdf. (In Russian).
4. Adamovich A.I., Klimov And.V. On Experience of Using the Metaprogramming Development Environment Eclipse/TMF for Construction of Domain-Specific Languages // Nauchnyy servis v seti Internet, Trudy XVIII Vserossiyskoy nauchnoy konferentsii (September 19–24, 2016, Novorossiysk, Russia). Moscow: Keldysh Institute of Appl. Math. of RAS: 2016. P. 3–8. URL: http://keldysh.ru/abrau/2016/45.pdf. (In Russian).
5. Adamovich A.I., Klimov And.V. How to Create Deterministic by Construction Parallel Programs? Problem Statement and Survey of Related Works // Program Systems: Theory and Applications, 8:4(35). 2017. P. 221–244. URL: http://psta.psiras.ru/read/psta2017_4_221-244.pdf. (In Russian).
6. Bocchino R.L. (Jr.), Adve V.S., Adve S.V., Snir M. Parallel Programming Must Be Deterministic by Default // Fifth USENIX Conference on Hot Topics in Parallelism, HotPar'09. Berkeley, CA, USA: USENIX Association, 2009. P. 4–4.
7. Bocchino R.L. (Jr.), Adve V.S., Dig D., Adve S.V., Heumann S., Komuravelli R., Overbey J., Simmons P., Sung H., Vakilian M. A Type and Effect System for Deterministic Parallel Java // SIGPLAN Not., 44:10. 2009. P. 97–116.
8. Klimov And.V. Dynamic Specialization in Extended Functional Language with Monotone Objects // SIGPLAN Not., 26:9. 1991. P. 199–210.
9. Klimov And.V. Deterministic Parallel Computations with Monotonic Objects // Nauchnyy servis v seti Internet: mnogoyadernyy komp'yuternyy mir. 15 let RFFI, Trudy Vserossiyskoy nauchnoy konferentsii (24–29 sentyabrya 2007 g., g. Novorossiysk). Moscow: Izd-vo Moskovskogo universiteta, 2007. P. 212–217, URL: https://pat.keldysh.ru/~anklimov/papers/2007-Klimov--Deterministic_Parallel_Computation_with_Monotonic_Objects.pdf. (In Russian).
10. Marlow S. Parallel and Concurrent Programming in Haskell. CA, USA: O'Reilly, 2013.
11. Kawaguchi M., Rondon P., Bakst A., Jhala R. Deterministic Parallelism via Liquid Effects // ACM SIGPLAN Not., 47:6. 2012. P. 45–54.
12. Kuper L. Lattice-Based Data Structures for Deterministic Parallel and Distributed Programming, Ph.D. Thesis. 2015. URL: https://users.soe.ucsc.edu/~lkuper/papers/lindsey-kuper-dissertation.pdf.
13. Kuper L., Todd A., Tobin-Hochstadt S., Newton R.R. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish // ACM SIGPLAN Not., 49:6. 2014. P. 2–14.
14. Kuper L., Turon A., Krishnaswami N.R., Newton R.R. Freeze after Writing: Quasi-Deterministic Parallel Programming with LVars // ACM SIGPLAN Not., 49:1. 2014. P. 257–270.
15. Kuper L., Newton R.R. LVars: Lattice-Based Data Structures for Deterministic Parallelism // 2nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC'13. New York, NY, USA: ACM, 2013. P. 71–84.

Operational Semantics of Reflex

Igor S. Anureev (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry, anureev@iis.nsk.su)

Reflex is a process-oriented language that provides design of easy-to-maintain control software. The language has been successfully used in several safety-critical cyber-physical systems, e. g. control software for a silicon single crystal growth furnace. Now, the main goal of the Reflex language project is development a support for computer aided software engineering targeted to safety-critical application. This paper presents formal operational semantics of the Reflex language as a base for applying formal methods to verification of Reflex programs.

This work has been supported by the Russian Foundation for Basic Research (grants 17-07-01600 and 17-01-00789).

Safety Analysis of Longitunal Motion Controllers during Climb Flight

Thomas Baar (Hochschule für Technik und Wirtschaft (HTW) Berlin, Department of Engineering I)

Horst Schulte (Hochschule für Technik und Wirtschaft (HTW) Berlin, Department of Engineering I)

During the climb flight of big passenger planes, the pilot directly adjusts the pitch elevator and the plane reacts on this by changing its pitch angle. However, if the pitch angle becomes too large, the plane is in danger of an airflow disruption on the wings, which can cause the plane to crash. In order to prevent this, modern planes take advantage of control software to limit the pitch angle. However, if the software is poorly designed and if system designers have forgotten that sensors might yield wrong data, the software might cause the pitch angle to become negative, so that the plane loses height and can - eventually - crash.

In this paper, we investigate on a model for a Boeing passenger plane how the control software could look like. Based on our model described in Matlab/Simulink®, it is easy to see based on simulation that the plane loses height when the sensor for the pitch angle provides wrong data. For the opposite case of a correctly functioning sensor, our simulation does not indicate any problems. This simulation, however, is not a guarantee that the control is indeed safe. For this reason, we translated the Matlab/Simulink®-model of the controler into a hybrid program in order to make this system amenable to formal verification using the theorem prover KeYmaera.

Formal Modelling and Verification for Heap-manipulating Programs

Bin Fang (Huawei, fangbin1990@me.com)

First, we demonstrate how to obtain formal specifications of sequential dynamic memory algorithms using a refinement-based approach. We define a hierarchy of models ranked by the refinement relation that capture a large variety of techniques and policies employed by memory allocation algorithms. This hierarchy forms an algorithm theory for memory allocators using list and it could be extended with other policies. The formal specifications are written in Event-B annotation and the refinements have been proved using the Rodin platform. Also we investigate applications of the formal specifications obtained, such as model-based testing, code generation and verification on code level.

Second, we define a technique for inferring precise invariants of heap-manipulating programs based abstract interpretation which is a framework used in static analysis. The abstract domain is constructed step-by-step by combining shape and numeric abstractions. The abstract elements are presented by our fragment of Separation Logic which can tack the complex construal and numerical properties on the structure of memory and on its size and content. To obtain compact elements of this abstract domain, we propose a composition operator to link hierarchical abstractions of the shape. It increases the readability and modularity of the specification as well as the modularity of the static analysis method.

Constructing Verification-Oriented Domain-Specific Process Ontologies

Natalia O. Garanina (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry, garanina@iis.nsk.su)

Igor S. Anureev (A.P. Ershov Institute of Informatics Systems, Institute of Automation and Electrometry, anureev@iis.nsk.su)

Vladimir E. Zyubin (Institute of Automation and Electrometry, Novosibirsk State University, zyubin@iae.nsk.su)

User-friendly formal specification and verification of concurrent systems from various subject domains are active research topics due to their practical significance. In this paper, we present the method for development of verification-oriented domain-specific process ontologies which are used to describe concurrent systems of subject domains. One of advantages of such ontologies is their formal semantics which makes possible formal verification of described systems. Our method is based on the verification-oriented process ontology. For constructing a domain-specific process ontology, our method uses techniques of semantic markup and pattern matching to associate domain-specific concepts with classes of the process ontology. We give detailed ontological specifications of these techniques. Our method is illustrated by the example of developing a domain-specific ontology for typical elements of automatic control systems.

The research has been supported by Russian Foundation for Basic Research (grants 17-07-01600 and 19-07-00762).

Towards automated error localization in C programs with loops

Dmitry A. Kondratyev (A.P. Ershov Institute of Informatics Systems SB RAS, apple-66@mail.ru)

Alexei V. Promsky (A.P. Ershov Institute of Informatics Systems SB RAS, promsky@iis.nsk.su)

The most recent trends in the C-light verification system are MetaVCG, semantic labels appropriate for verification condition (VC) explanation and symbolic method of definite iterations. MetaVCG takes a C-light program together with some Hoare's logic and produces on-the-fly a VC generator (VCG), which in turn processes the input program. Hoare's logic for definite iterations is a good choice if we try to get rid of loop invariants. Finally, if a theorem prover was unable to validate some VCs we could follow two ways. Obviously, we could revise/enrich specifications or/and underlying proof theory to prove the truth of VCs. Or, perhaps, we could concentrate upon establishment of falsity, which meant there were errors in annotated program. This is where semantic labels play crucial role providing some natural language comments about wrong VC as well as a back-trace to the error location. The newly developed ACL2 heuristics to prove VC falsity is the main theme of this paper.

This research is partially supported by RFBR grant 17-01-00789.

Proving safety by disproving: finite countermodel finding for the infinite-state and parameterized verification

Alexei Lisitsa (Department of Computer Science, University of Liverpool, a.lisitsa@liverpool.ac.uk)

Traditional model checking has been very successful in the verification of finite state systems. In many cases, though the finite state abstraction is not enough, as we might need to verify the algorithm or protocol for essentially unbounded computation/execution, e. g. to ensure that a protocol is correct for any number of participants, or for any size of a data structure. In such cases, we are facing infinite state, or parameterized systems, the verification of which in general is an undecidable problem.

In this talk, we introduce and overview conceptually simple but powerful and efficient method for automated safety verification of infinite-state and parameterized system. The method utilises modelling of reachability between the states of a system as derivability in the classical first-order (FO) logic. With such modelling to establish the safety of a system, that is non-reachability of unsafe states, it is sufficient to show that a particular first-order formula is not provable. In this FCM method, the latter task is delegated to the available automated finite model finding procedures.

In the talk we present theoretical foundations, show the relative completeness of the method and illustrate it by numerous applications for infinite-state and parameterized verification tasks selected from the areas of

- mutual exclusion protocols;
- cache coherence protocols, which ensure that multiple caches in multi-core systems have consistent copies of the fragments of the main memory;
- recreational mathematics, including the first fully automated solution of so-called MU-puzzle, taken from the famous book Goedel, Escher, Bach: An Eternal Golden Braid book by D. Hofstadter.

Quantifier solutions to interval systems of linear algebraic equations

Sergey P. Shary (Novosibirsk State University and Institute of Computational Technologies of Siberian Branch of Russian Academy of Sciences, Novosibirsk, shary@ict.nsc.ru)

The talk is devoted to interval systems of linear algebraic equations, a classical object, which is quite simple, but serves as a basis for a large number of mathematical models of the real world. Solutions and sets of solutions for such systems can be defined in a wide variety of ways, since the interval uncertainty, by its very nature, has a dual character, which is associated with the use of different logical quantifiers, either existence or universality. In this way, we obtain definitions of the so-called quantifier solutions of interval equations, having a clear meaning and can be interpreted as solutions of games or decision-making processes under conflict and uncertainty.

We consider various numerical methods for estimating sets of quantifier solutions for interval systems of linear algebraic equations. They are based on the use of the Kaucher complete interval arithmetic, where improper (reverse) intervals are allowed and basic arithmetic operations are algebraic and order completions of those for the usual real intervals.

An Application of Test-Driven Development Methodology into the Process of Hardware Creation (a View from a Software Perspective)

Sergey M. Staroletov (Polzunov Altai State Technical University, serg_soft@mail.ru)

Vladimir Fedorov (Polzunov Altai State Technical University, vladimir.fodorow@gmail.com)

The TDD (Test-Driven Development) methodology was created by Kent Beck, which proposed firstly to write a test then to make a trivial implementation of a module to satisfy this test. FPGA (Field-Programmable Gate Array) is a reprogrammable matrix of logical elements to solve different hardware tasks (signal processing, calculations, external devices driving). The aim of the research is an application of Test-Driven Development methodology which was originally created for producing software from the scratch, to the FPGA devices programming, from the software engineering point of view. In the work we make a short review of the methodology, describe briefly Verilog language for the FPGAs, propose recommendations how to follow this methodology while develop software for specify a hardware behavior on the basis on a demo functionality to control an audio device with WM8731 chip to play waveform files from an SD card.

1. Introduction

The Test-Driven Development (TDD / developing based on test creation / red-green refactoring) is one of the novel techniques of software engineering in the sphere of program testing. It was originally proposed by Kent Beck [1], who is well known as one of the major methodology specialist in the software engineering world.

Although the approach has been initially introduced in 2003, but till today there is not a big amount of software companies that use it in the real software production. However, questions of the applicability of this methodology are periodically posted in the major software testing conferences.

From our point of view, the industrial applicability of TDD is still having some issues primarily because of the following:

- The developers are not accustomed to start new implementations from initial tests.
- There are no good examples of complex software projects that have been completed entirely according to this methodology.
- Customers and managers mistakenly believe (in our opinion) that the test creation during the whole process of the software creation (and especially before it) increases the cost of the project.

The TDD itself is a process of preliminary creation of a next unit test and then a code to pass the test and this leads either to write a new code in the form of a class or a method (developing a new functionality), or to refactor existing code (clarification of a functionality, developing a new case of existing functionality). In the first, a test should fail (red) and therefore a developer should write functionality to pass the test (green), so another name of TDD is "red-green refactoring", that corresponds to the xUnit test results stripe in IDE, and incites code to be created.

In any case, TDD at its each step implies a minimal functionality, which only can pass a previously written test (for example, we have a test 2 + 2 for the addition method with the expected value of 4, that leads a minimum code for such a method that just returns 4 without actually writing the logic of the addition). It encourages the writing of stubs and eliminates developers from lengthy reflections on complex implementations. Each process step must be fixed as a commit to a version control system. Some arguments for this methodology were presented in [2]:

- it is an opportunity to demonstrate a working (according to current tests) product at any stage of the development (customers will be satisfied);
- it implies the maximum of the code coverage;
- the project at any time is a series of small changes that are easy to track (project managers are happy).

The additional time used to write tests potentially reduces the time for the further correction of errors that could have been detected if the functionality had been written without tests. Also, we get a code and tests as the result that can be used at every next step as a regressive control of the correctness of the old functionality when making new changes.

In the article [3], Intel architect Ron Wilson wondered if this methodology also applies to the FPGA hardware design. According to the comments to the article, hardware developers did not reach a common opinion on this issue.

2. Background

2.1. FPGA

FPGA (field-programmable gate array) is a device (a special processor) that contains developer-reconfigurable blocks with inputs and outputs, and their configuration code is written in a special language for describing integrated circuit hardware [4] (HDL, hardware description language).

In this paper, we consider the code in Verilog HDL, as one of the most popular languages today for developing programs for FPGAs. Among others, one of the ways of the applicability of unit testing and TDD is studying the capabilities of new libraries, protocols, new programming languages through tests for them, so our work is a research of applying known methodologies to unknown languages with a completely different order of execution and meaning of the tests.

The advantage of using FPGAs in working with external equipment to the microcontrollers (such as, for example, Arduino or STM32 and especially to devices with an operating system like Raspberry Pi) is that the processor here does not run some compiled machine code, and its logic is programmed especially to solve a given task; the logic is programmed as a reaction at changes of the clock edge or as a reaction at changes of input signals, so there is no need to use waiting cycles, interrupts, and so on, which makes it easy and efficient to implement communication protocols with external components. Therefore, novel outgoing SoC systems (System on Chip), such as Raspberry Pi4, are planned to be equipped with an additional FPGA processor, which will be responsible for programmed interaction with interconnected external hardware, while the rest will work with internal equipment, OS, user, etc.

2.2. Verilog

The Verilog language (the grammar of the language is available in [5]) is a high-level C- and Pascal-like language at the level of operations, conditions, and cycles, which is then synthesized into logical elements of the FPGA. Verilog program describes modules connected by inputs and outputs, and one can declare memory registers in the form of bitwise data and their arrays, connections, and control structures. The programs are written according to the event-oriented pattern, usually at a positive or negative front of a change of frequency clock or some other signal. For example, the code

```
always@(posedge clock_50M)
begin
dacclk_a<=dacclk;
end
```

is performed at a frequency of 50 MHz or 50,000,000 times per second and changes the state of an output signal, and also for this change (from 1 to 0 or from 0 to 1) in another place of code can be also defined a hook or this signal can be out to an external pin.

3. Some notable differences of device testing

The testing process of the code running on a device is quite different from the testing used in software engineering. We identified the following main differences:

1. The time parameter is added to the test, that is, the tests should check not just the correspondence of the output value according to a given input, but the correspondence of the output signals over time according to specified input signals (temporal correspondence).
2. A rather long synthesis time, project deployment, a large number of signals, a short device response time lead to the fact that static testing methods (simulation) become very important during the development.
3. The complexity of determining a current state of the working code on the device. Often, the so-called "printf injection" method [2] cannot be directly used here to print the current state, but it is still possible by flashing LEDs, outputting to an LCD screen, etc. (as the work with these components is turn and tested).
4. Dynamic testing is carried out by connecting external logic analyzers, recording the necessary signals and working with them on a computer to establish the correctness of work.

4. Hardware demo

As a part of this study, we are developing a hardware music player for music in the form of WAV files from an SD card with the possibility of making hardware mixes during the playback. The hardware consists of a demo board with a Cyclone IV EP4CE22E22 processor [6], a WM8731 [7] audio processor, and an SD card slot (Figure 1).

The development is based on the source code of the samples supplied with the equipment in the Intel Quartus Prime Lite Edition environment (free edition).

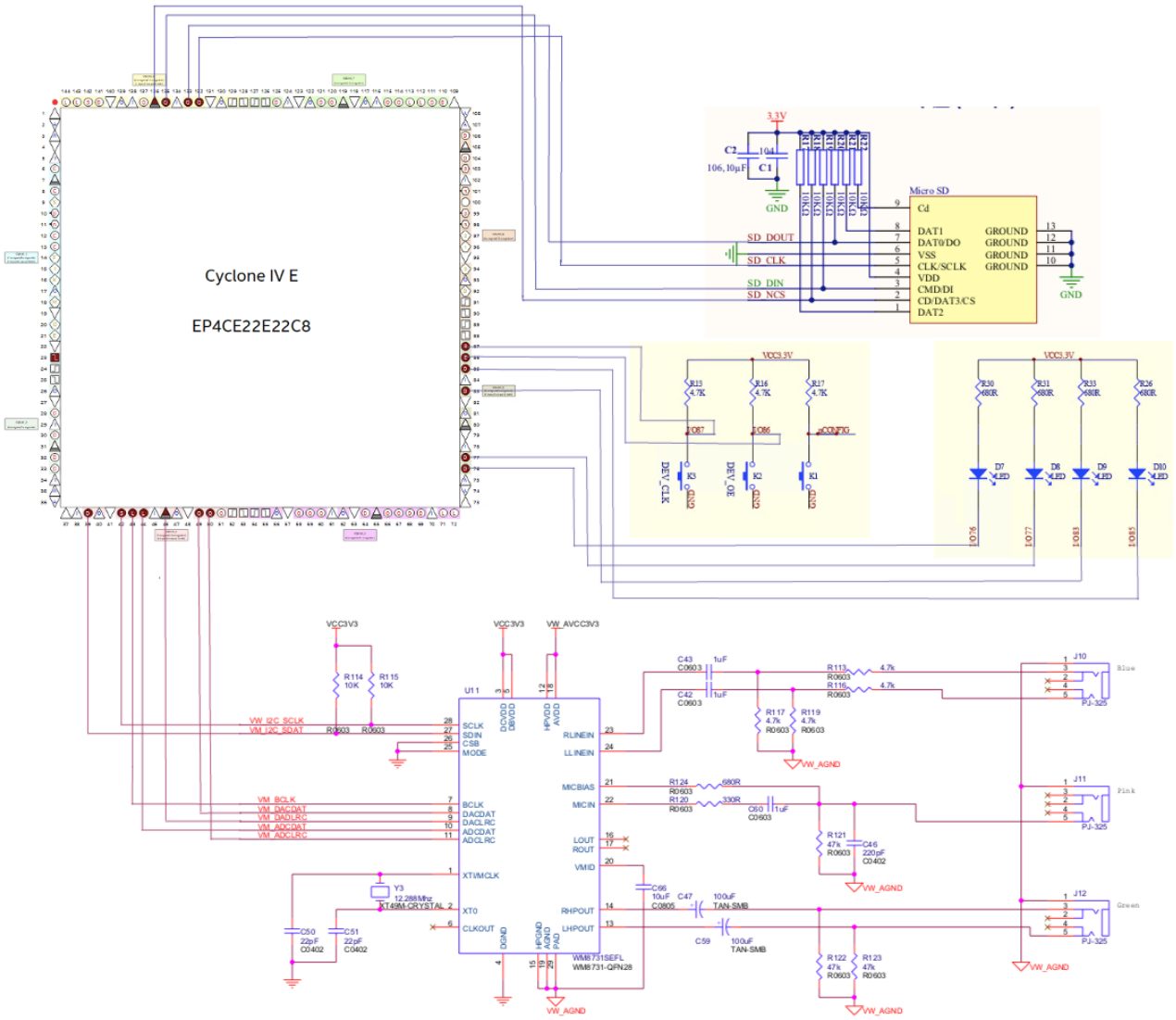

Fig. 1. Hardware solution scheme

5. A TDD Case Study

In this section, we propose a method for developing the demo system using the Test-Driven Developing approach. In this case, at each step, we will implement the minimum working functionality that corresponds to the tests for it.

1. Create a project for the target hardware (Cyclone IV EP4CE22E22). Create a git-repository and commit (in general, we propose to make commits at all steps after writing/changing a test and a code).
2. Define the external interface. According to our hardware scheme, we need to interact with the SD card, audio processor, as well as LED indicators on the board, clocking and buttons status. It is possible by knowing the input and output signals - they are defined in the

documentation (datasheets) and we have already established their connection according to our solution plan (Figure 1). Here it is necessary to act according to the MDD (Model-Driven Development) approach and define these signals (Figure 2) in a graphical editor, then generate a test (Test Bench) containing a link to the main program block with these signals and the code of this block (in this case, in Verilog language, using its "#" delay operator).

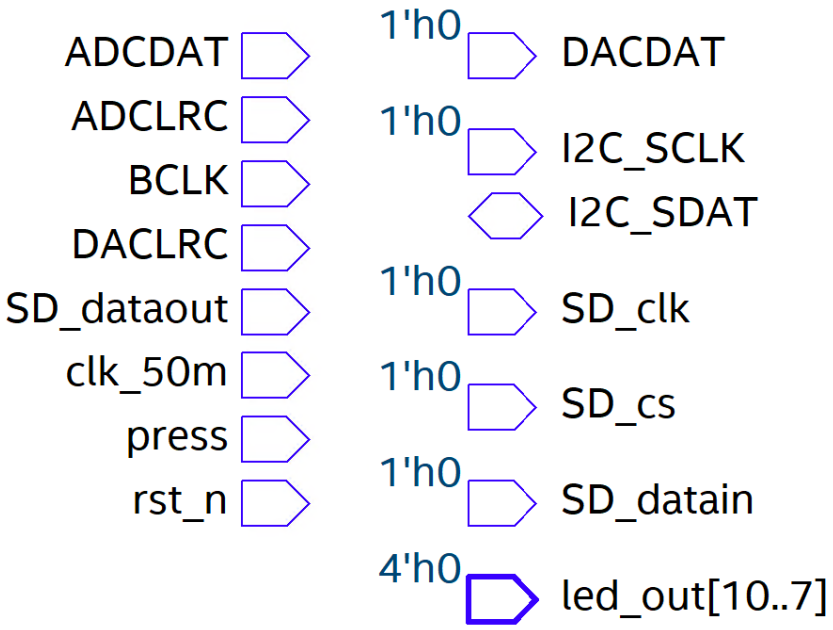

Fig. 2. We have designed the input and output signals of the solution

This step is similar to the description of interfaces and classes in order to pass the first test of the existence of a class in order to create this class in the OOP programming languages according to the TDD.

3. Next, it's time to implement the functionality. We implement the test of the existence of a block (component) to work with an SD card, therefore, we need to create such a block and describe its inputs and outputs (a subset of our signals).
4. According to the documentation, the SD card needs its own clocking. Therefore, we write a test that for a given clocking (clk_50m) signal, the SD_clk signal will be different. Using the component development model, we integrate an IP module (IP or Intellectual Property, similar to the components) of pll for frequency division and integrate it into the solution (Figure 3). In this case, the specified test will pass.

Here one can proceed to the creation of tests and some implementation of work with an audio processor, or continue working with an SD card, this depends on the goals of the

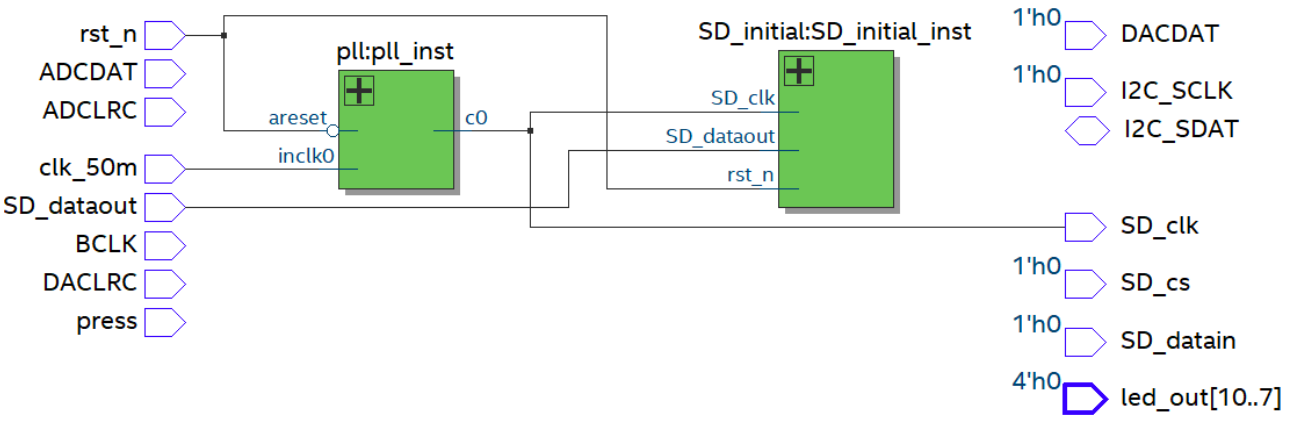

Fig. 3. Added a frequency divider for the SD

team, the division of tasks between different people, etc. We will continue to work with the SD.

5. Next, we need to write a test for the functionality of initializing and reading data from the SD card. For the validation, we can use an output resulting signal, which in the trivial implementation is set to 1. And even output it to the LED in order to dynamically observe the work of the implementation. But how to continue this process in order to proceed from the real testing goals?
6. So, we need some kind of information about logical signals associated with the SD based on time. To do this, either we can manually determine the signals by drawing them on a timing diagram, or use a logic analyzer and record the real signals for a while from a real device that works correctly. Figure 4 shows a recorded example of SD card signal logic.

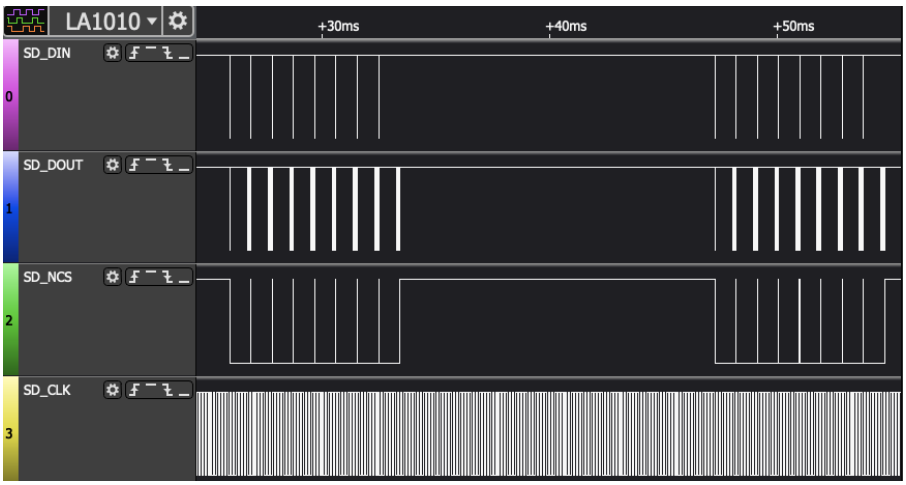

Fig. 4. SD signals on a logic analyzer

In this case, an analyzer allows us to export the signal values to a .CSV file, and it is then

possible to generate a test by following the following pattern: do a signal value setting according to an information from the analyzer, waiting for some time (delta to a next signal value) and setting the next signal value from the dataset, and continue it to all the signals in the dataset (according to the columns in a recorded .CSV file).

7. After generating or editing such a test, an implementation code is generated that simply repeats the test. Next, more tests are created / recorded and the functionality is gradually refined so that all tests should pass.
8. In the previous step, it is not necessary that the functionality was fully implemented and if at least one test passes (and it will pass, as the functionality initially meets the tests), then we can move on to other blocks (components of the system). This is consistent with TDD in programming, when after passing tests one can move on to implementing other methods, classes, or even create a build to show it to customers (only show the functionality according to current tests!).
9. Acting in this way, we implement the functionality as connected blocks for the entire solution (Figure 5), each one is validated at every moment by the tests.

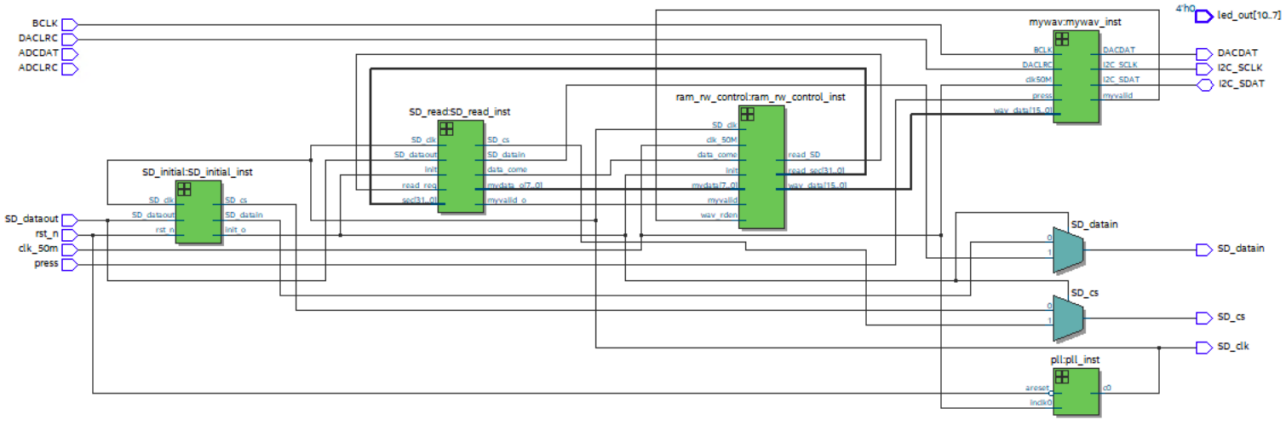

Fig. 5. The final block diagram of the solution (RTL viewer tool).

10. For this demo task, initial wave file data loading and playing can be implemented by reading raw data from a given byte in the file system, then we can implement a support for the normal file reading by adapting a C-code of work, for example, with FAT32, by rewriting it to Verilog, based on the file location tests in specified data areas. We can implement further the monitor of the volume of music on the LEDs, switching between files by pressing keys on the board and so on, based on tests and trivial implementations.

6. Suggestions for the implementation of this process

In the Intel Quartus environment (we mean the free Lite version) there are not all the necessary solutions for the implementation of the described process. Firstly, there is no integration with Git version control system, although there are some version control capabilities, but they are implemented in its own way and do not correspond to modern solutions about project management.

Secondly, the generation of diagrams is possible only based on the code or the creation of a module in the form of diagrams that is incompatible with the previous one. It is necessary to create a graphical editor of blocks with input and output signals with the generation of code and tests for them with the TDD process support.

Thirdly, there are no visual methods for creating tests in time and processing data from the logic analyzer. Therefore, it is necessary to create own solution integrated with Quartus Prime. Interaction with it can be organized using scripts in the Tcl language (Quartus supports launching such scripts and provides automation tools for working with the project), then creating a client-server solution (remote automation control) that will allow the implementation of the described TDD development approach integrated with the mainstream FPGA tool.

7. Conclusion

Finally, we can state that the TDD process can be applicable in the sphere of hardware code creation. The main different here is the tests with temporal organization. Such process can be done with using an MDD way to organize input and output signals in different blocs levels. The source of test signals can be either a graphical signal editor and recorded data from a logical analizer. The process requires writing a plugin to existing FPGA synthesis tools. As the tests are temporal, formal methods of verification are applicable here and LTL predicates can be used to describe requirements for FPGA system behavior. In [8] we introduce a step-by-step demo of current implementation of the process by implementing a simple hardware counter.

References

1. Beck, Kent. Test-driven development: by example. Addison-Wesley Professional, 2003.
2. Staroletov, Sergey. Basics of Software Testing and Verification [in Russian]. Lanbook, Saint Petersburg. 2018. ISBN 978-5-8114-3041-3.
3. Test-Driven Hardware Development: True or False? Available from: https://systemdesign.intel.com/test-driven-hardware-development-true-or-false/
4. Harris, David, and Sarah Harris. Digital design and computer architecture. Morgan Kaufmann, 2010.
5. Verilog 2001 grammar. Available from: https://github.com/antlr/grammars-v4/blob/master/verilog/Verilog2001.g4
6. Cyclone IV Device Handbook. Intel, Altera. Available from: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf
7. Wolfson WM8731 / WM8731L Portable Internet Audio CODEC with Headphone Driver. Available from: https://statics.cirrus.com/pubs/proDatasheet/WM8731_v4.9.pdf
8. Fedorov, Vladimir. Test Driven Hardware Development on System Verilog v1. Available from: https://www.youtube.com/watch?v=AsitQp2G3FI

Proving properties of Discrete-Valued Functions using Deductive Proof: Application to the Square Root

Vassil Todorov (Groupe PSA, France, vassil.todorov@lri.fr)

Safouan Taha (LRI, CentraleSupélec, Université Paris-Saclay, France, safouan.taha@lri.fr)

Frédéric Boulanger (LRI, CentraleSupélec, Université Paris-Saclay, France, frederic.boulanger@lri.fr)

Armando Hernandez (Groupe PSA, France, armando.hernandez1@mpsa.com)

For many years, automotive embedded systems have been validated only by testing. In the near future, Advanced Driver Assistance Systems (ADAS) will take a greater part in the car's software design and development. Furthermore, their increasing critical level may lead authorities to require a certification for those systems. We think that bringing formal proof in their development can help establishing safety properties and get an efficient certification process. Other industries (e.g. aerospace, railway, nuclear) that produce critical systems requiring certification also took the path of formal verification techniques. One of these techniques is *deductive proof*. It can give a higher level of confidence in proving critical safety properties and even avoid unit testing.

In this paper, we chose a production use case: a function calculating a square root by linear interpolation. We use deductive proof to prove its correctness and show the limitations we encountered with the off-the-shelf tools. We propose approaches to overcome some limitations of these tools and succeed with the proof. These approaches can be applied to similar problems, which are frequent in automotive embedded software.

On parallel composition of Finite State Machines with timed guards

Aleksandr S. Tvardovskii (Tomsk State University, tvardal@mail.ru),
Nina V. Yevtushenko (Ivannikov Institute for System Programming of the RAS, nyevtush@gmail.com)

Finite State Machines (FSMs) are widely used for analysis and synthesis of digital components of control systems. In order to take into account time aspects, timed FSMs are considered. In this paper, we address the problem of deriving a parallel composition of FSMs with timed guards and output delays (output timeouts). When the parallel composition is considered, component FSMs work in the dialog mode and the composition produces an external output when interaction between components is terminated. In this work, we formally introduce the parallel composition operator for FSMs with timed guards (TFSM) and show that unlike classical FSMs, a "slow environment" and the absence of live-locks are not enough for describing the behavior of a composition of deterministic TFSMs by a deterministic FSM with a single clock. Although the set of deterministic FSMs with timed guards is not closed under the parallel composition operator, some classes of deterministic TFSMs are still closed under this operator.

This work is partly supported by RFBR project № 19-07-00327/19.

Научное издание

X Workshop PSSV

Program Semantics, Specification and Verification: Theory and Applications

July 1–2, 2019, Novosibirsk, Akademgorodok, Russia

Abstracts

Издается в авторской редакции

Дизайн обложки Т.М. Бульонковой
Ответственный за выпуск И. С. Ануреев
Подготовка к печати Г. Р. Семенихиной,
Т. А. Марковой, С. В. Исаковой, Е. В. Неклюдовой

Подписано в печать 20.06.2019 г.
Формат 60x84 1/8. Уч.-изд. л. 6. Усл. печ. л. 5,6.
Тираж 50 экз. Заказ № 156.
Издательско-полиграфический центр НГУ
630090, г. Новосибирск, ул. Пирогова, 2

Organized by:

A. P. Ershov Institute of Informatics Systems

Novosibirsk State University
*THE REAL SCIENCE

Sponsored by:

Novosibirsk State University
*THE REAL SCIENCE

РОССИЙСКИЙ ФОНД ФУНДАМЕНТАЛЬНЫХ ИССЛЕДОВАНИЙ
РФФИ

Microsoft Research

ARQA TECHNOLOGIES

Siberis®

STI · INTERNATIONAL

STI · INNSBRUCK

ISBN 978-5-4437-0918-5