УДК 004.832.32

System Description: Russell - A Logical Framework for Deductive Systems

Vlasov D.Yu. (Sobolev Institute of Mathematics, Novosibirsk State University)

Russell is a logical framework for the specification and implementation of deductive systems. It is a high-level language with respect to Metamath language [7], so inherently it uses a Metamath foundations, i.e. it doesn't rely on any particular formal calculus, but rather is a pure logical framework. The main difference with Metamath is in the proof language and approach to syntax: the proofs have a declarative form, i.e. consist of actual expressions, which are used in proofs, while syntactic grammar rules are separated from the meaningful rules of inference.

Russell is implemented in c++14 and is distributed under GPL v3 license. The repository contains translators from Metamath to Russell and back. Original Metamath theorem base (almost 30 000 theorems) can be translated to Russell, verified, translated back to Metamath and verified with the original Metamath verifier. Russell can be downloaded from the repository `https://github.com/dmitry-vlasov/russell`

Keywords: *logical framework, formal mathematics, deductive system, proof checker*

1. Introduction

Recently the ambitious QED project [2] has celebrated its 20 year anniversary, while the claimed goals of this project are still far from being reached. Several papers [10], [9], [4], [6], [3] addressing the history of QED clearly state that yet there is no computer language, which has all the expected features of a QED system. Summarizing these papers, it can be said that the major barriers of QED are:

- the 'Balcanization' of QED-like systems, i.e. when there are different languages with different foundations and there is no simple way to share formalized proofs between them [6]
- the lack of a powerful automation, which could seriously reduce end user efforts to prove a theorem [4]
- the difference between the standard practical mathematical language, which is used in papers and textbooks, and the formal language of QED-like systems [9]

Russell is another system, which is intended to reach QED goals. It was developed to address

all these obstacles, and some of them are to some extent removed in the Russell design.

2. The Russell System

The Russell system is a general purpose framework for definition and usage of different formal deductive systems. The variety of formal systems, which can be represented in Russell, is quite wide, although limited. For example, non-monotonic deductive systems cannot be given in Russell. As a high-level language with respect to Metamath [7], Russell inherits its foundations, which are close to the notion of Post's canonical system [8]. The approach of Metamath to syntax of expressions is more general than that of Russell: syntactic rules in Metamath are indistinguishable from the meaningful rules of inference and can have meaningful (essential, in Metamath terms) premises. In Russell, the grammar of expressions must be context free by the language definition. In all other aspects Metamath and Russell share the same foundations.

2.1. Pure Logical Framework

What distinguishes Metamath and Russell from other logical frameworks is purity: their deduction engines don't use any built-in logic (in the form of axioms and inference rules). The deduction used in Metamath and Russell is concerned with making a proper substitution, applying it to a certain expression and checking the coincidence of the result with some other expression. From the very general point of view, such logic-neutrality is a good property, because we want to avoid the situation when some formal deductive system has an a-priory better fitness to the language than the other because of the propinquity with the underlying logic of a logical framework. Similar arguments are mentioned in the paper [6], where the statement of *foundational pluralism* is given. In fact, the property of logic-neutrality (or pureness) is crucial to the desired foundational pluralism, because it gives a uniform core language for a vast variety of deductive systems. Thus, 'Balkanization' of formal mathematics could be at least based on the same language (although the sharing of proofs between different foundations is still challenging).

What distinguishes Russell from Metamath:

- Expression grammar syntax rules are separated from the general logical inference rules, so the grammar is guaranteed to be context free.
- The Russell proof language is more human-friendly than the Metamath proof language.
- Russell uses a special syntactic construct for definitions, while in Metamath definitions

are simply axioms with labels, starting with 'df' prefix.

2.2. Brief Description and Comparison of Metamath and Russell

At first, let's get familiar with Metamath (the complete description of Metamath syntax and semantics may be found in [7]). Metamath is a very simple language, and the complete list of Metamath keywords is: `$c $v $d $f $e $a $p $= $. ${ $}`

The syntax of expressions in Metamath is not distinguished from the syntax of axioms and inference rules. The latter assertions are treated differently inside proofs, but on the syntax level there is no difference between these two kinds of assertions. This feature makes language simpler, as the same mechanism is used for syntactic inferences and for the logical inference.

For example, the definition of well-formed-formula with $\rightarrow$ and $\neg$ logical connectives in Metamath looks like:

```
$c ( ) -> -. $.
$v ph ps $.
${
    wph $f wff ph $.
    wn $a wff -. ph $.
$}
${
    wph $f wff ph $.
    wps $f wff ps $.
    wi $a wff ( ph -> ps ) $.
$}
```

Here we define constants with `$c <c_1> .. <c_n> $.` construction. Here `->` stands for $\rightarrow$ (implication) and `-.` stands for $\neg$ (negation), and this is how implication and negation are represented in the original Metamath theorem base. All constants are used as a terminal grammar symbols. Note, that brackets are also treated as definable symbols, not pre-defined ones. Variables, symbols which may be substituted with, are also defined with `$v <v_1> .. <v_n> $.` construction. Two constructions with labels `wi` and `wn` are essentially grammar rules with a single non-terminal `wff`, in BNF notation:

```
wff ::= -. wff | ( wff -> wff )
```

The definition of axioms of the Hilbert-style propositional calculus looks like:

```
${
    wph $f wff ph $.
    wps $f wff ps $.
    ax-1 $a |- ( ph -> ( ps -> ph ) ) $.
$}
${
    wph $f wff ph $.
    wps $f wff ps $.
    ax-mp.1 $e |- ph $.
    ax-mp.1 $e |- ( ph -> ps ) $.
    ax-mp $a |- ps $.
$}
${
    a1i.1 $e |- ph $.
    a1i $p |- ( ps -> ph ) $=
        wph wps wph wi a1i.1 wph wps ax-1 ax-mp $.
$}
```

The first two are classical axiom ($\varphi \to (\psi \to \varphi)$) and a rule of inference (modus ponens):

$$\frac{\varphi \quad (\varphi \to \psi)}{\psi}$$

Now let's describe the Metamath syntactic construction in details. Technically the construction `<id> $f <type> <var> $.` is called a 'floating' hypothesis, and means that `<var>` has a type `<type>`. Since Metamath semantics is based on a stack machine, when `id` label is executed, the expression (pair of symbols) `<type> <var>` is pushed on a stack. The `<id> $e <s_1> ... <s_n> $.` construction is called an 'essential' hypothesis, and has a classical meaning of a premise of a proposition. When met in a proof (which essentially is an reverse polish notation (RPN) program for the stack machine), its expression (i.e. sequence of symbols `<s_1>...<s_n>`) is pushed on a stack.

The propositions may be axiomatic or provable, and axiomatic ones are designated as `<id> $a <s_1>...<s_n> $.` Provable propositions are written as a syntactic construction `<id> $p <s_1>...<s_n> $= <l_1>...<l_n> $.` and the sequence of labels `<l_1>...<l_n>` here is a proof of the proposition, i.e. the program for a stack machine, written in RPN form.

The braces `${` and `$}` are used to define a scope of an assertion as a whole, with hypotheses (both, floating and essential) and proposition. When met in a proof, the label of an assertion acts as an operation on the stack: it fetches the appropriate number of expressions from stack, accordingly to its arity (here arity is a sum of the number of floating and essential hypotheses), then it matches floating hypothesis with the corresponding expressions from stack, checks their type and forms a substitution. Then this substitution is applied to the essential hypotheses of the assertion, and proof checker verifies, that the corresponding expressions on the stack are symbol-wise the same. After all these checks, the substitution is applied to the statement of an assertion and the result is pushed on a stack.

What is left to finish the verification of a Metamath proof, when a proof of some provable assertion (i.e. theorem) is executed on a stack, is to ensure, that exactly one expressoin is left on the stack and it coincides with the proposition of the theorem symbol-wise.

As a note on Metamath language, we can imagine, that hypothetically one could add substantial (i.e. essential, in the Metamath terminology) hypotheses to the syntactic grammar rules, but of course, it would make little sense, since the common practice is to use context-free grammars for a language.

Now let's look, how the same rules and axioms look like in Russell syntax:

```
constant { symbol ( ;; }
constant { symbol ) ;; }
constant { symbol -> ;; ascii -> ;; latex \rightarrow ;; }
constant { symbol -. ;; ascii -. ;; latex \lnot ;; }
type wff ;;
rule wn (ph : wff) {
    term : wff = # -. ph ;;
}
rule wi (ph : wff, ps : wff) {
    term : wff = # ( ph -> ps ) ;;
}
axiom ax-1 (ph : wff, ps : wff)  {
    prop 1 : wff = |- ( ph -> ( ps -> ph ) ) ;;
}
axiom ax-mp (ph : wff, ps : wff)  {
```

```
    hyp 1 : wff = |- ph;;
    hyp 2 : wff = |- ( ph -> ps );;
    ----------------------
    prop 1 : wff = |- ps;;
}
theorem a1i (x : wff, y : wff)  {
    hyp 1 : wff = |- x ;;
    ----------------------
    prop 1 : wff = |- ( y -> x );;
}
proof of a1i {
    step 1 : wff = ax-1 () |- ( x -> ( y -> x ) );;
    step 2 : wff = ax-mp (hyp 1, step 1) |- ( y -> x );;
    qed prop 1 = step 2 ;;
}
```

At first, the constant symbols are defined, just as in Metamath. Note, that in Russell a symbol may have a unicode representation (in UTF-8 encoding), ascii representation and latex representation. The ascii representation is used for correct translation of Russell sources to Metamath. Latex representation may be used for the latex sources generation, and unicode representation is used in Russell sources in order to represent mathematical symbols as close to the real mathematical practice as possible.

Then we define a type `wff` - a non-terminal symbol of grammar of expressions of propositional logic (in Metamath `wff` is just a constant and is not treated specifically). The next two syntactic constructions are the rules (productions) of a corresponding context-free grammar and are read as: if `ph` and `ps` are of type `wff` (i.e. are well formed formulas), then the expressions `-. ph` and `( ph -> ps )` are also of type `wff`, i.e. are also well formed formulas. Context-freeness here is obligatory by design, because it is impossible to use any kind of hypothesis other then floating (typing) ones. The `term` keyword in the definition of the rule body means, that the following sequence of symbols is just a raw expression, any may not be directly inferable. For example, the expression `term : set = # { a , b }` is a set, and therefore has no truth-value semantics. The keyword `;;` is a symbol sequence terminator, like `$.` in Metamath.

Then there are two axioms: `ax-1` and `ax-mp` (rules of inference are considered a non-zero-ary

axiomatic assertions, so are called also 'axioms'). The typing hypotheses here are represented in a common for many programming languages manner: a comma-separated list of variable-type pairs, separated by colon, like `ph : wff, ps : wff`. The proposition of assertion is marked up with `prop` keyword, while hypotheses are marked up by `hyp` keyword and are separated from the propositions (there may be several propositions) by a `--------` keyword (not less then 5 minus symbols), which mimics the separation line in the classical representation of rules of inference as $\frac{H_1,\ldots,H_n}{P}$. The expressions in assertions have a sequence starter symbol $\vdash$ (a turnstile), which means, that these expressions are *logical*, i.e. directly used in inferences.

Then the theorem `a1i` and its proof follows. The syntactic form of a theorem is just the same as the form of an axiomatic assertion, except for the heading keyword `theorem` instead of `axiom`. A theorem must have at least one proof. The proof has a classical linear structure, with explicit links, showing, from which previously proven step or hypothesis the current one follows and by which assertion. Also each step is provided with the appropriate expression, which is essentially what is proved in this step. The `qed` statements shows, which step (usually the last one) is symbol-wise the same as the proposition of a theorem.

So, if we compare the proof languages of Metamath and Russell, we can see, that Russell proofs are much closer to the human mathematical practice and may be understood without any external program tools. Metamath proof is an RPN programm, so the only way to understand the Metamath proof is to compute it as an RPN program, which demands a proof assistant (except for some trivial cases).

Note, that the exact substitution are not explicitly presented in Russell proof, while they are in Metamath. The substitutions for each proof step may be obtained by matching appropriate expressions from the proof with the expressions from an assertion of the step. In the example above, the substitutions for the first and second steps will be $\{x/ph, y/ps\}$ (in real mathematics substitutions may be much more complex then just a variable replacement). Technically, the proof in Russell as a sequence of steps is a stack trace obtained from executing a proof in an RPN form, which is stripped off all intermediate steps related to the syntax formation. When a Russell proof is verified or translated to Metamath, these intermediate steps are restored from the syntax tree of an expression and corresponding matching substitutions.

Thus, the Russell proof language is natural and simple, which imposes minimum restrictions and allows for user-defined grammars for expressions. Together with the possibility to use the conventional set theory it bridges the gap between the computer-based mathematics and the

common practice mathematics from textbooks.

2.3. Definitions in Metamath and Russell

One of the most important features of any computer-based deductive system (framework) is safety and reliability [1]:

to what extent one can trust computer proofs ?

Reliability of a formal system is a complex subject, which involves several aspects. One of these aspects is the size of the axiomatic base used by a theory. If it is large then there can be some (unintentionally) hidden inconsistency inside of it, and if so, this will lead to the triviality of the whole theory (in case of an explosive logic). On the other hand, if the set of the true axioms is small and well-known (like some variant of ZF set theory) then its degree of reliability is very high.

If each definition is introduced as a new axiom, like it is done in Metamath, then the number of axioms increases fast as the theory grows, and at some point there is no guarantee that all of these axioms are consistent. To address this issue, definitions in Russell are introduced as a special syntactic construct and certain properties are checked for each definition to ensure that adding the underlying axiom will give a conservative extension of the theory. Conservativity here means that if something can be proved with the help of some definition, it can also be proved without it. This property is strictly proven, so it gives some more certainty about correctness of Russell theories.

Example of a definition in Russell:

```
definition df-or (ph : wff, ps : wff)  {
    defiendum : wff = # ( ph \/ ps ) ;;
    definiens : wff = # ( -. ph -> ps ) ;;
    ----------------------
    prop : wff = |- ( defiendum <-> definiens ) ;;
}
```

Here we define a logical 'or' connective on the basis of negation and implication. Keyword `defiendum` denotes what is defined (the disjunction of two formulas), and `definiens` denotes the extension of the notion 'or' - its definition as a formula. So far as both constructions are terms, and Russell semantics (as well as Metamath semantics) doesn't provide a mechanism for internal term rewriting (since it is a pure inference engine), we cannot just 'substitute'

any occurrence of disjunction with the appropriate negation and implication. Instead, we make a new *axiom* out of the definition, which allows us to replace disjunction with its definition indirectly, by common logical rules. The proposition of this axiom is obtained from the `prop` expression of the definition by replacing `defiendum` and `definiens` meta-variables by the corresponding terms. For this particular definition this axiom's proposition will be:
`( ( ph \/ ps ) <-> ( -. ph -> ps ) )`

The fact that all Russell sources can be translated back to Metamath and checked with its original proof checker shows that Russell at least as reliable as Metamath. Moreover, the declarative format of proofs in Russell makes it possible for a human to do an independent verification of proofs. Of course, it would look strange to exploit human ability of checking a formal proof in a QED system, but still, from the philosophical point of view, human understanding is an ultimate judge, and is very important.

3. Implementation

Currently the Russell language is implemented as a translator from / to Metamath and is written in c++14. The Russel repository includes test scripts, which run a chain of translations:

```
Metamath -> Russell -> Metamath
```

The translation of the whole Metamath base (about 30 000 theorems) is rather fast in all directions. The most problematic from the performance point of view is expression parsing in Russell. Metamath uses an explicit construction of expressions in proofs, so it does not require any parsing or matching algorithm when checking its source. Unlike Metamath, Russell must parse expressions in order to get syntax trees and such parsing takes the most of time in comparison to all other steps, like matching or translation.

One of the features implemented in the toolchain of translators is that it can automatically divide the original Metamath source (the single file of almost 150 megabytes) into reasonably small parts following the internal layout inside the source file. After breaking it into pieces, one can browse the source file tree with the standard desktop navigation tools and watch source files in the standard desktop editors without the necessity to handle a single 150 Mb file.

Russell is not considered by the author as an experimental or model language. It is supposed to be a useful, universal, and convenient tool for all kinds of activity in the field of formal deduction. To achieve this, the language of implementation (c++), quality of source code, efficiency of algorithms, and usefulness to the end user are of a great importance. Russell

implementation should be able to work with hundreds of thousands of assertions in a reasonable time, which is the subject of ongoing research and development.

4. Conclusion and Future Work

The Russell logic framework is a robust, fast, and reliable general purpose tool for representation of formal deductive systems. The Russell language is designed to be simple and easy-to-learn and provides proofs in a declarative form (which is a standard practice in informal mathematical texts).

Essentially, powerful automation is the only one blocking property left in the list of the desirable QED features. Therefore, the next challenge is to implement the automated proving feature so that the process of formal proof design would be easier. The first step to making an automated proof engine for Russell has been already taken and it showed the potential feasibility of such a goal. Since the proof search space suffers from an extreme combinatorial explosion, some extraordinary means are needed to cope with it. Standard techniques will not work here, since the nature of the underlying deductive system is apriori unknown to the prover (which is a consequence of the logic-neutrality property). For example, in general we can not assume, that the underlying logic is cut free (and actually it is not in the case of the Metamath theorem base).

To create a powerful prover for Russell we plan to use advanced machine earning techniques to make the prover use the experience of the already proven theorems. Ideally, it should be able to generate human-like proofs, formed as a combination of previously obtained proofs.

The other important goal is to support importing of other bases of formalized mathematics into Russell. Some successful attempts of importing HOL theorem base into Metamath have been already undertaken [5], so there is a hope that it would possible to join a large part of the already formalized mathematical knowledge under a common logical framework, but with different foundations. A more ambitious goal is to join these bases upon a common foundation.

Список литературы

1. *Adams M.* Proof Auditing Formalized Mathematics, Journal of Formalized Reasoning Vol. 9, No. 1, (2016), pages 3 – 32
2. *Anonymous.* The QED Manifesto, in: A. Bundy (ed.), Automated Deduction - CADE 12, LNAI 814, Springer-Verlag, pages 238 – 251. (1994)

3. *Barendregt H., Wiedijk F.*, The Challenge of Computer Mathematics, Transactions A of the Royal Society 363 no. 1835, pages 2351 – 2375, (2005)
4. *Blanchette J. C., Kaliszyk C., Paulson L. C., Urban J.* Hammering towards QED, Journal of Formalized Reasoning Vol. 9, No. 1, (2016), pages 101 – 148.
5. *Carneiro M.M.* Conversion of HOL Light proofs into Metamath, Journal of Formalized Reasoning Vol. 9, No. 1, (2016), pages 187 – 200.
6. *Kohlhase M., Rabe F.* QED Reloaded: Towards a Pluralistic Formal Library of Mathematical Knowledge, Journal of Formalized Reasoning Vol. 9, No. 1, (2016) pages 201 – 234.
7. *Megill N.* Metamath: A Computer Language for Pure Mathematics, Lulu Press, Morrisville, North Carolina, (2007)
8. *Post E.* Formal Reductions of the General Combinatorial Decision Problem, American Journal of Mathematics 65 (2), pages 197 – 215, (1943).
9. *Wiedijk F.* The QED Manifesto Revisited. In: From Insight to Proof, Festschrift in Honour of Andrzej Trybulec. (2007), pages 121 – 133.
10. *Wiedijk F.* The Seventeen Provers of the World, Volume 3600 of Lecture Notes in Computer Science, (2006) Springer-Verlag.

UDC 004.05

A Method to Verify Parallel and Distributed Software in C# by Doing Roslyn AST Transformation to a Promela Model

Sergey Staroletov and Anatoliy Dubko

(Polzunov Altai State Technical University)

In this paper, we describe an approach to formal verification of parallel and distributive programs in C#. We use Microsoft Roslyn technique to get syntax and semantic information about interesting constructions in the real source code to generate some corresponding code in Promela language, designed to model actor-based interoperation systems, so we do a program-to-model transformation. Then, we verify the usual problems of parallel and distributive code by checking pre-defined LTL formulas for the model program. We are able to provide checking of data races, improper locking usage, possible deadlocks in distributive service interoperations using the Model Checking approach. This method can be used to construct a static analyzer for the .NET platform.

Keywords: *Roslyn, Verification, Static Analyzer, LTL, SPIN*

1. Introduction

This work is dedicated to improving the quality of modern software which has parallel executable entities and acts as a distributive system or microservice. Such kind of systems can have tricky errors, just exposed in rare situations. It is usually impossible or very challenging to detect such errors by testing.

Formal verification methods were introduced to ensure the correctness and reliability of such type of program systems, to detect faults at the different stages of software development and maintenance to consistently reduce them.

We assume that the formal verification approach [1] should be applied here but it will be hard to understand by an ordinal software developer how to create different kinds of models to verify [2], so such techniques should be a transparent part of developing process, and additional checking should be integrated into a compiler or an IDE.

The problems of verification and creation of a verifying compiler are examples of the fundamental problems of modern programming that are in the progress of being solved.

In this paper, we deal with C# parallel programs and WCF services. We use Microsoft Roslyn technique to obtain the AST (Abstract Syntax Tree) from C# input sources to generate a corresponding code in Promela modeling language intended to make further verification of the generated model code using SPIN verifier according to defined classes of possible parallel and distributive errors and generated requirements as LTL (Linear Temporal Logic) formulas.

2. Related Work

2.1. .NET and C# for parallel and distributed systems creation

C# is an industrial, type-safe, object-oriented modern language designed to develop applications running in the .NET Framework environment [3]. Using C#, developers can create general-purpose software including standard and universal desktop applications, mobile applications, web services, distributed components; client-server, database and web applications.

The C# programs run within the .NET Framework – an integrated Windows component that contains the Common Language Runtime (CLR) virtual system and a unified set of Base Class Libraries (BCL). The CLR is an implementation of the Common Language Infrastructure (CLI) – an international standard, the foundation of execution and development environments with close interaction of languages and libraries. Novel platform implementation named .NET Core provides the developers with some abilities to create application not only for Windows platform, but also for Linux and Mac, and this implementation is even open-sourced [4].

Source code written in C# is compiled into the Intermediate Language (IL) following the CLI specification. IL code and resources, such as bitmaps and strings, are stored on disk in an executable file, called an assembly, with EXE or DLL extension.

The .NET platform and the C# language provide the ability to create distributed components or applications:

- Web API. The Web API is a RESTful HTTP web service that can interact with various components. These can be ASP.NET web-, mobile or regular desktop applications.
- WCF (Windows Communication Foundation). WCF provides a platform for building service-oriented (SOA) applications and implements a manageable approach to create web services and clients for them [5]. With WCF, developers can send data in the form of asynchronous messages from one service endpoint to another. The endpoint can be a part of a permanently available service hosted in IIS (a web-based Internet Information Services server) or represent a service hosted inside an application. Messages can be in

the form of a complex stream of binary data.

- .NET Sockets. Sockets are used to build traditional client-server applications. By connecting two sockets explicitly, the applications can transfer data between different processes, nodes or platforms.

In the current work, we model distributive interoperations only as WCF services and clients due to their high-level logical structure.

2.2. Roslyn

Roslyn [7] is a platform that provides the system developers with various powerful tools for analyzing and parsing .NET languages (mostly C#) code. The source code of this platform is freely available on MS GitHub account [6].

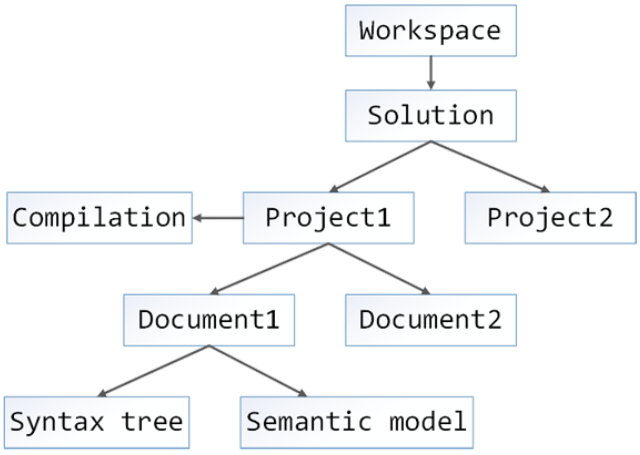


Fig. 1. Retrieving data for analysis with Roslyn [8]

With the help of tools provided by the Roslyn platform, it is possible to perform a full syntax analyzes of the code by traversing all supported language constructs. The Visual Studio environment allows developers to create tools embedded in the IDE itself (as Visual Studio extensions) and independent applications on the basis of Roslyn.

When analyzing code with Roslyn tools, it is possible to get a list of files from a solution whose source code is being checked, to get the necessary entities for parsing (syntax model), and then to get access to the semantic model of the program after compiling it (Figure 1).

For a complicated analysis, it is necessary to obtain a syntax tree and a semantic model. A syntax tree is built from a program source code and is used to link various language constructs. A semantic model provides information about program objects and their types.

For every language structure, Roslyn defines corresponding type nodes. Moreover, for each

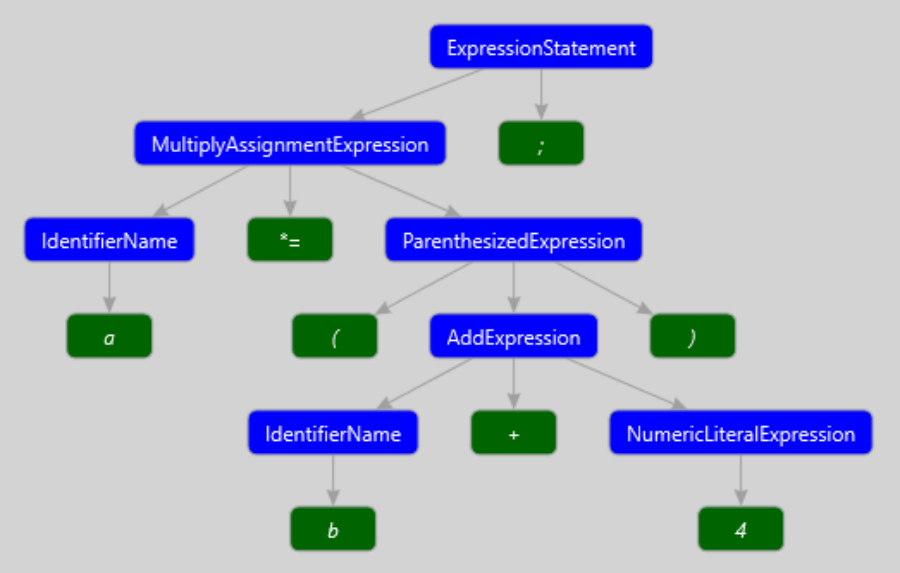


Fig. 2. An expression tree for a * = (b + 4) with lexemes [8]

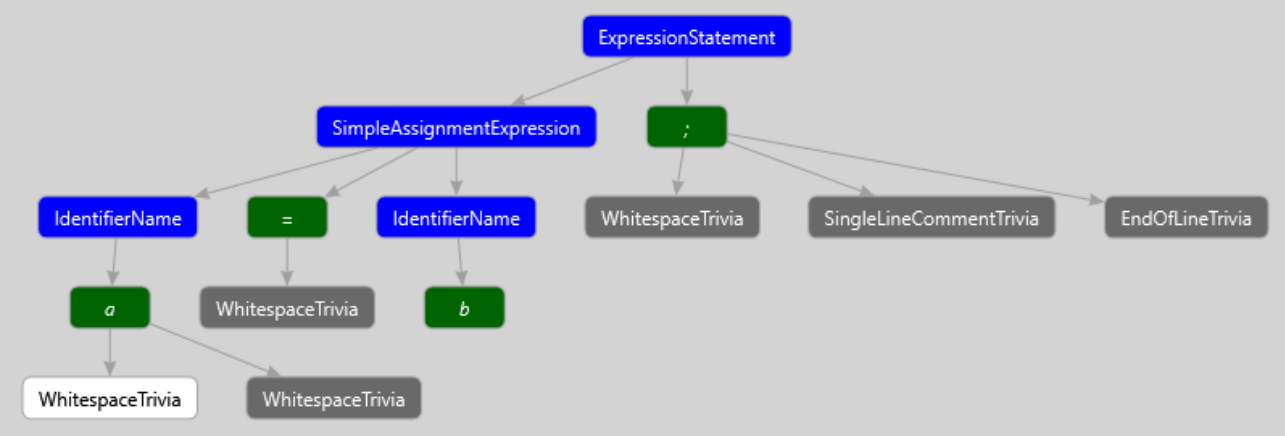


Fig. 3. A sintax trivia tree for $a = b; //Comment$ [8]

node type, a method in code can be defined that performs a crawl through the nodes of this type. Thus, by adding handlers to traverse of various nodes, we can analyze only the constructions of language we are interested in. An example tree for the expression $a* = (b + 4)$ in syntax tokens is shown in Figure 2.

In addition, there exist "syntax trivia" – elements in the tree that will not be compiled into IL code, they are stands for some additional syntax information. This category includes comments, preprocessor directives, spaces, etc. Figure 3 shows a tree with extra syntax information for the expression $a = b; //Comment$. This data can be possibly used for implementing additional processing for constructions as special comments, for example, it is a usual way to write control annotations for deductive verification approach now, as ACSL annotations for C programs to prove with Frama-C tool are written in C comments [9].

The semantic model in Roslyn provides useful information about objects and their types. It is a powerful source for building deep and complex analyzers. That is why it is essential to have a correct compilation and a correct semantic model.

2.3. Actor model

Today, the dominant way to programs creation is the imperative approach based on the general state (it is also used for C# programs). Very often, the code of such programs from the very beginning is written without considering the possibility of parallelization, and parallel actions occur in the code only when needed.

The actor model on the other side, "forces" the code to be parallel from the starting point. This model is a mathematical representation of parallel computing, which uses the concept of "actor" as a universal primitive. It was applied [10] as a basis for understanding the calculation using processes and as a theoretical basis for some practical implementations of parallel systems.

The basic idea is that the actor-based application is built from many lightweight processes called *actors* [11]. Each actor is responsible for one tiny task, so it is easy to understand what does it responsible for. Programs with more complex logic can be implemented as an interaction of several actors that are concurrently sending messages between each other.

So, the actor is a computational entity that, in response to a received message, can simultaneously:

- send a finite number of messages to other actors;
- create a finite number of new actors;
- select the behavior that will be used when processing the next received message.

It does not assume the existence of a specific sequence of the above-described actions and all of them can be performed in parallel. Separating the sender from the messages was a fundamental achievement of the actor model. Message recipients are identified by the address (PID, process identifier), which is sometimes called the mailbox address. Thus, an actor can interact only with those actors whose addresses it has, and it can extract addresses from received messages or know them in advance.

The most outstanding implementation of the actor model was made in Erlang language [12]. A rather popular implementation of this model is Akka library [13] for Scala.

2.4. Promela and SPIN

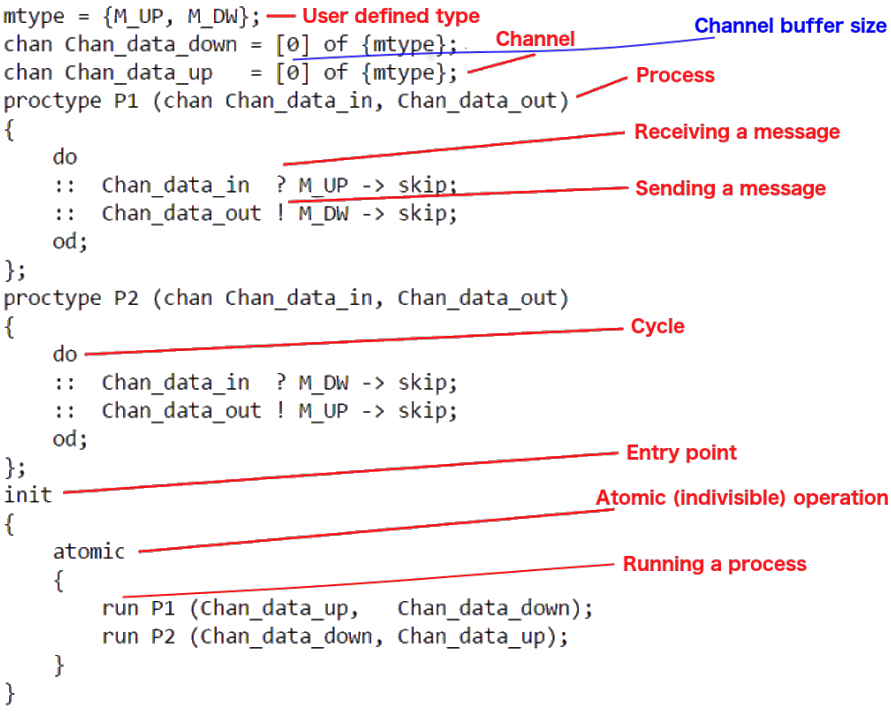


Fig. 4. An example Promela model

SPIN [14] is a utility for verifying the correctness of distributed software models. The abbreviation stands for Simple Promela INterpreter. This utility is used for automated verification of models, and it can also work as a simulator, executing one of the possible traces of a model of system behavior.

The SPIN system checks *not the programs themselves, but their models.* To build a model for an original parallel program or an algorithm, the engineer (usually manually) builds a representation of this program in the C-like input language, called Promela (Protocol MEta-LAnguage). This program in Promela language can be considered a model of the verified program. Promela language constructs are simple, they have clear and distinct semantics, which allows translating any program in this language into a transition system with a finite number of states for verification purposes. The requirements for the model are expressed in the language of LTL (Linear-time Temporal Logic) [15].

Input models in Promela are different from original verifiable programs, usually written in high-level programming languages. Promela programs do not have classes, and they represent a flat structure of interacting parallel processes, as we described in the actor model section (Figure 4), have a minimum of control structures, all variables have finite domains. Therefore, such a

program can be considered a model of the system being analyzed; it represents an abstraction of the original system, in which the engineer should reflect those aspects and characteristics of the real system that are very significant for the properties specified for the verification.

The system description is expressed in Promela language must preserve the essential properties of this system. It should be stressed that the resulting verification quality of Promela programs entirely depends on the degree of adequacy of the constructed model. The model construction process can be done manually or automatically, and in this paper, we present the ways of auto model generation based on the C# compiler information.

2.5. Existing solutions to do C# code verification

To check the C# programs statically, Microsoft Research offered a solution called Spec# (Specification Sharp) which extends C# language with constructs for non-null types, preconditions, postconditions, and object invariants [17]. With this solution, the developer should manually specify additional code to describe the program with special requirements in the form of logical predicates. Later, in [18] an embedded code contracts approach was presented, it became a part of the .NET platform, introducing annotations to the C# classes and ways to statically check the contracts assumptions while code writing and compiling from the IDE. It is intended to prove the program logic, and it is hard to check the interoperations with this approach.

MS Research has some trying to create a formal language to describe models with message passing, and in [19] Sing# language was presented as an extension to Spec# but the current state of the project is unclear and it seems they stopped developing and using it.

In [20] ISP RAS introduced an approach to do static analysis of C# programs based on the symbolic execution method with using advanced SMT solvers.

In [21] the PVS-Studio static analyzer for C, C++ and C# programs was described and its internal methods were discussed. It can detect some threading issues, and it uses Roslyn as a backend.

We can state that none of the described tools uses Promela and SPIN as a way to check the extracted models from C# input code. The analyzers can use Roslyn to obtain AST, but then they use own special techniques. None of the methods can verify distributed service interoperations.

3. Code Analysis and Model Generation

We consider here some algorithms for transforming syntax elements we are interested in the C# programs, to a Promela code according to our goals. Some ideas of it were given in our paper [22].

3.1. Ways of thread creation and its modeling

Consider some common ways to create parallel threads in C# language. Firstly, a thread can be created with the Thread class from the System.Threading namespace (Figure 5).

```
var thread = new Thread(() => { });
thread.Start();

var threadWithParam = new Thread(param => { });
threadWithParam.Start(new object());
```

Fig. 5. Creating Threads with the Thread Class

This class is used to create two types of threads: parameterized and non-parameterized. Its constructor takes a delegate of type ThreadingStart or ParametrizedThreadingStart, explicitly or implicitly. In Figure 5, the parameter is passed as a lambda expression, which is implicitly reduced to the above types.

Also, a parallel code can run with the Task class from the System.Threading.Tasks namespace (Figure 6).

```
Task.Run(() => { });

var task = Task.Run(() => "result");
```

Fig. 6. Creating Threads with the Task Class

Using classes from this namespace, one can create high-level thread types, taking advantage of thread-pooling. The static Run method accepts an Action type delegate with no parameter or Func delegate (from the System namespace), which can return a result. The method returns its result in the form of a new object of type Task, which represents the running task passed as the parameter.

Another method is the Parallel class from the System.Threading.Tasks namespace (Figure 7).

```
Parallel.For(0, int.MaxValue, i => { });

Parallel.ForEach(new[] {1, 2, 3}, entry => { });
```

Fig. 7. Creating Threads with the Parallel Class

Also, we consider some methods are used to create parallel loop processing. The static For method takes as its parameters the initial value of the counter, the final value of the counter, and a parameterised delegate (of type Action) that handles the current value of the counter.

The static ForEach method takes as its parameters a collection that implements the typed IEnumerable <T> interface and a parameterized delegate (of type Action) that accepts the current collection element in the iteration.

Now consider modeling the interaction of threads in C# as Promela structures. For the simulation, we decided to use a separate Promela process for each running thread in C#. The interaction between threads, as well as the awaiting of their completion, are modeled by transmitting synchronization messages through Promela channels. The types of processes are described using the declaration with the *proctype* keyword [16]. Processes are always declared globally. Processes are started from other processes by the means of the *run* operator.

Later we describe a way to the code generation for this body from all the possible variants of parallel entity creation described in this section.

3.2. Analysis of the syntax, semantics and data flow of the C# programs and its modeling

Before starting the analysis of C# sources, it is necessary to construct a graph of method calls inside the program being analyzed. This graph will help:

- Firstly, detect calls of methods that trigger a new thread.
- Then determine the order of calls, starting from an arbitrary method to generate definitions in Promela before their use further.
- And as a result, switch to a pure interprocess communication model in Promela without calling intermediate methods.

Consider an example of a method call in C# (Figure 8).

We see, in order to find all method calls in the source code of the program, we need to find all nodes of the InvocationExpression type in its syntax tree.

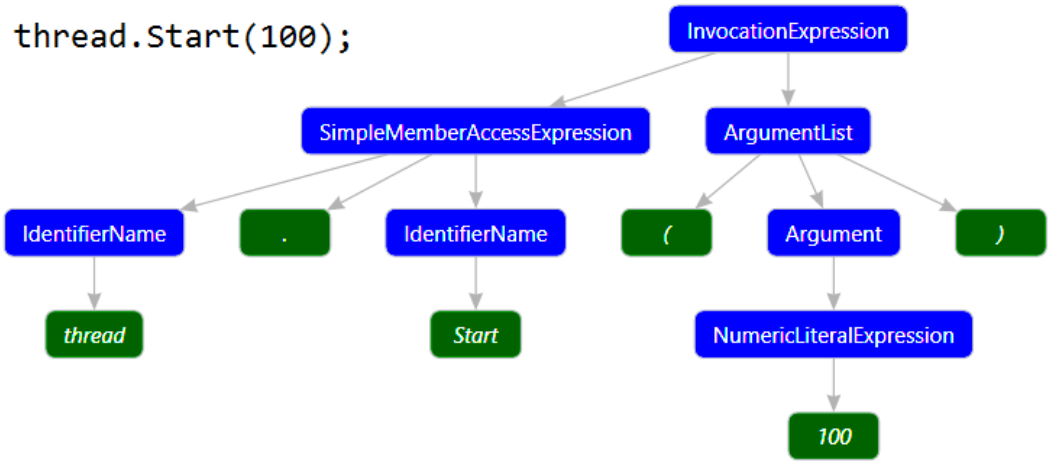


Fig. 8. Method Call syntax tree

To construct the graph, it is necessary to determine the relationship *"which method is called from a which one"*. It is required for each method call to find a definition of its parent method or a lambda expression (Figure 9).

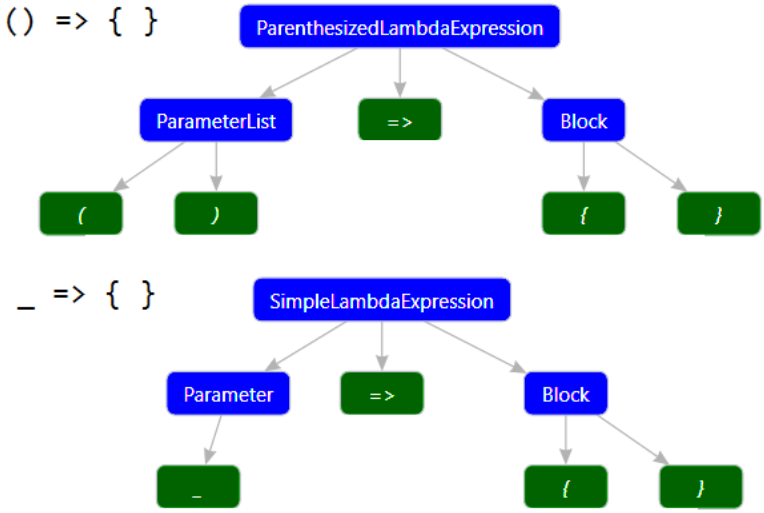


Fig. 9. Lambda Expression syntax tree

Moving up in the tree starting from a method call, we should find such a parent definition of the method or a lambda expression. This link will be an edge in the call graph.

We discovered that each node in the syntax tree has already defined hash code. Due to this, it is possible to use it as a unique value of a vertex in the graph without fear of the collisions occurrence between identical syntactic structures in the source code.

An example of the resulting call graph is shown in Figure 10.

The graph is acyclic, but it is not always connected. With further code generation on

```
private static void Main(string[] args)
{
    if (!args.Any()) return;
    var integer = ParseSomeData(args[0]);
    Console.WriteLine(integer);
}

private static int ParseSomeData(string data)
{
    Console.WriteLine(data);
    return int.Parse(data);
}
```

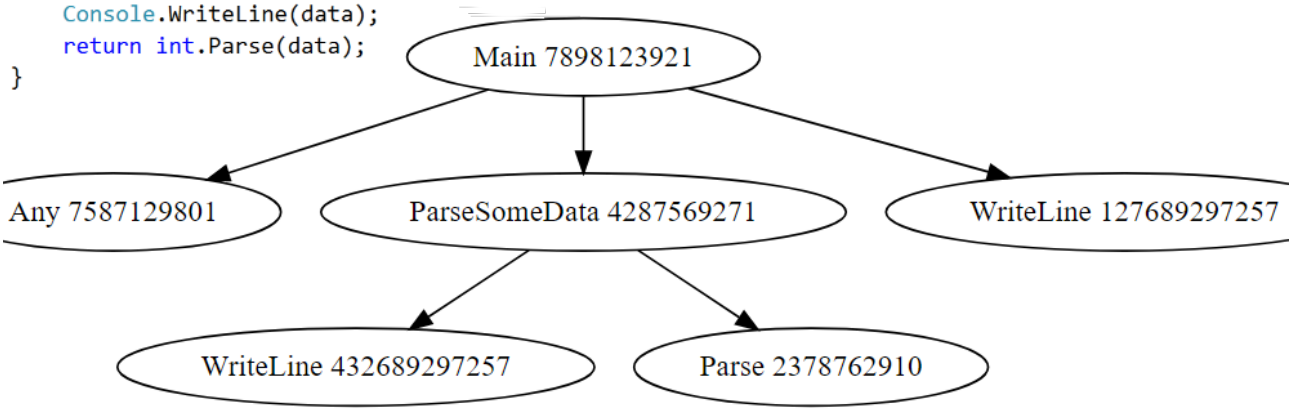


Fig. 10. Call graph in C# source code

Promela, acyclicity is easily removed when using the semantic model, since in this model the same characters have the same hash codes.

During the construction of the call graph, we can immediately determine which of the calls starts a new thread. For this, it is necessary to make a semantic analysis of the method call, setting some conditions:

- The name of the method that starts the thread.
- The type of object with which the method is called.
- If the method is not static, check, was the constructor called with the delegate of this thread object.

The first two conditions are checked using the appropriate properties of the method call symbol in the semantic model. To check the last condition, it is necessary to carry out an additional analysis:

- If a static method is called to start a thread, then check its argument, which must be a delegate.
- If the method is called immediately after the creation of the object, then analyze the child nodes of the call tree to check whether the constructor with the delegate was called.
- If the reference to the object was passed to a local variable, field or class property before calling the method, then we need to analyze the data flow of the parent method body,

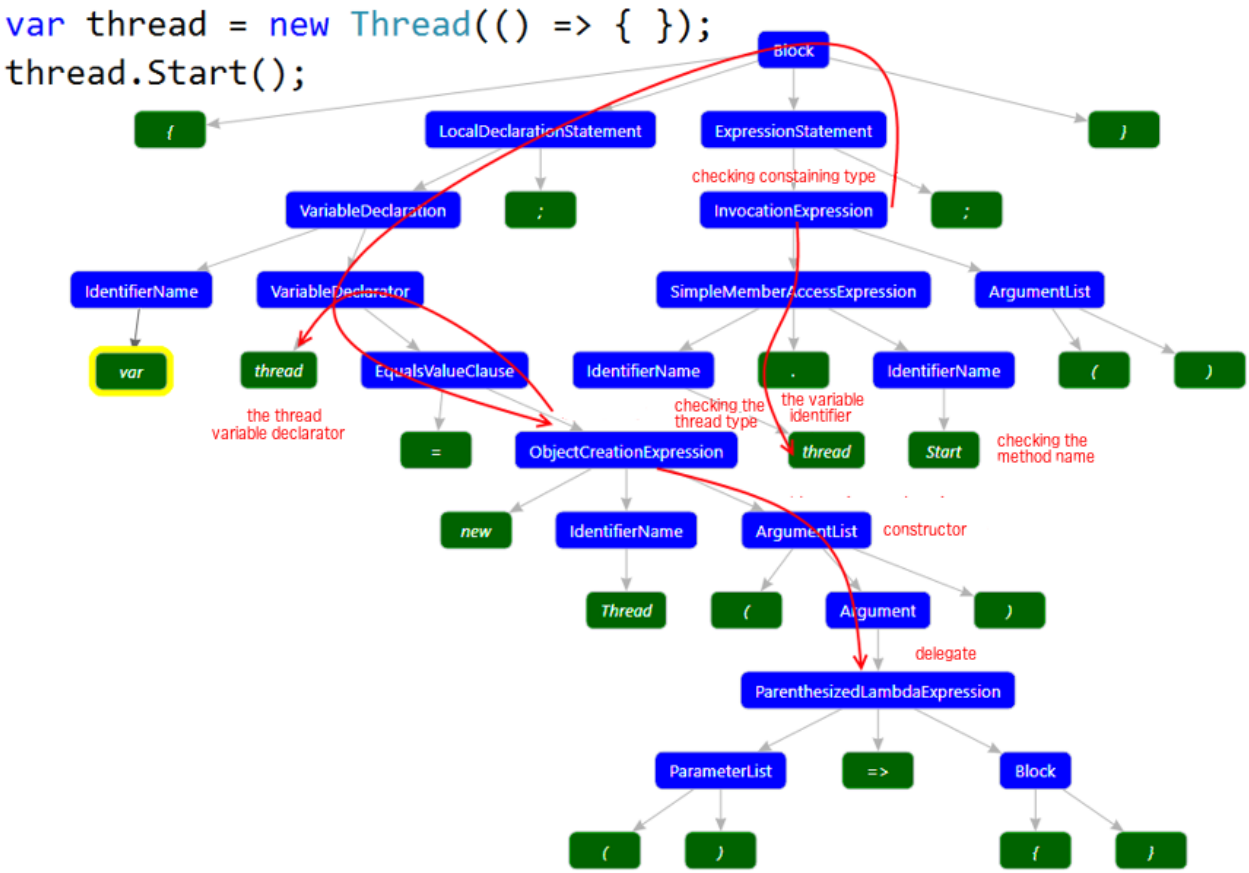


Fig. 11. Creating and starting a thread through a variable

delegate or class and link the local variable and the corresponding constructor call to the delegate, if any (Figure 11).

The Roslyn platform provides a number of useful classes to work with syntactic trees. One of them is the CSharpSyntaxWalker class, which implements the Visitor Design Pattern [23]. After inherited it, we can start a traversal through all syntactic nodes of the tree, simultaneously redefining the methods of visiting nodes of various types.

The idea of generating Promela code is precisely this crawling of the entire syntax tree of C# code with pre-determined methods for generating syntax nodes we need (Figure 12).

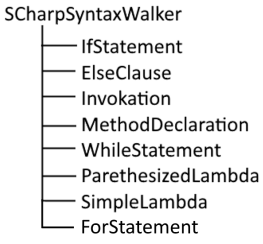


Fig. 12. C# syntax nodes where Promela code will be generated

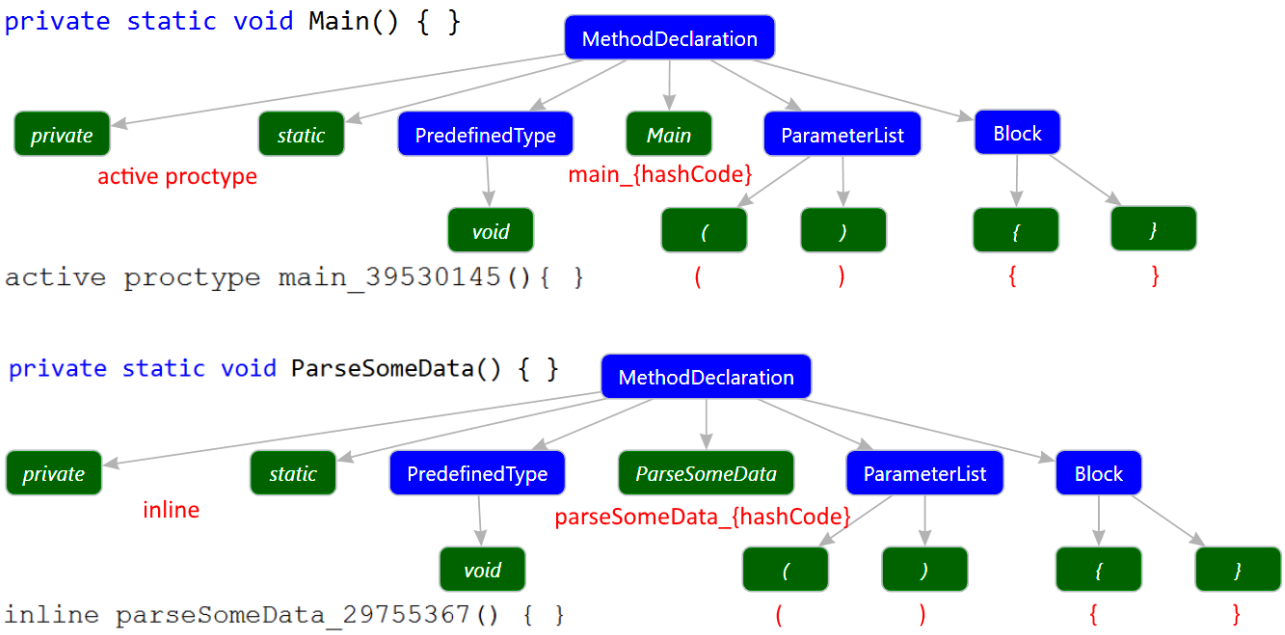


Fig. 13. Promela code generation for MethodDeclaration

Now we proceed to describe some nodes processing.

Each traversal of the MethodDeclaration, ParenthesizedLambda and SimpleLambda nodes will create a separate unit to generate Promela code, which will do this for the remaining nodes which belong to each of these declarations. Also during the crawl, an entry point to the program will be selected, and it is becoming later as the initial state when generating the resulting code using the call graph.

When visiting nodes of the MethodDeclaration type, we provide an algorithm for generating code in Promela presented in Figure 13. If the declaration is a program entry point, then it is generated as the initial running process in Promela, otherwise as an inline function [24].

When generating declarations and calls in Promela, the hash codes of corresponding symbols from the semantic model are used for names, because the same symbols, unlike syntax nodes, have the same hash codes.

When visiting nodes of ParenthesizedLambda and SimpleLambda types, which are threads delegates, a graphical illustration of this Promela code generation algorithm is shown in Figure 14. Here for each delegate, a new, not yet started process is created.

When visiting nodes of the Invocation type, a Promela code generation algorithm is provided in Figure 15. If an invocation launches a new thread, then a call code is generated for a previously generated process corresponding to the delegate. Otherwise, an inline function call is generated.

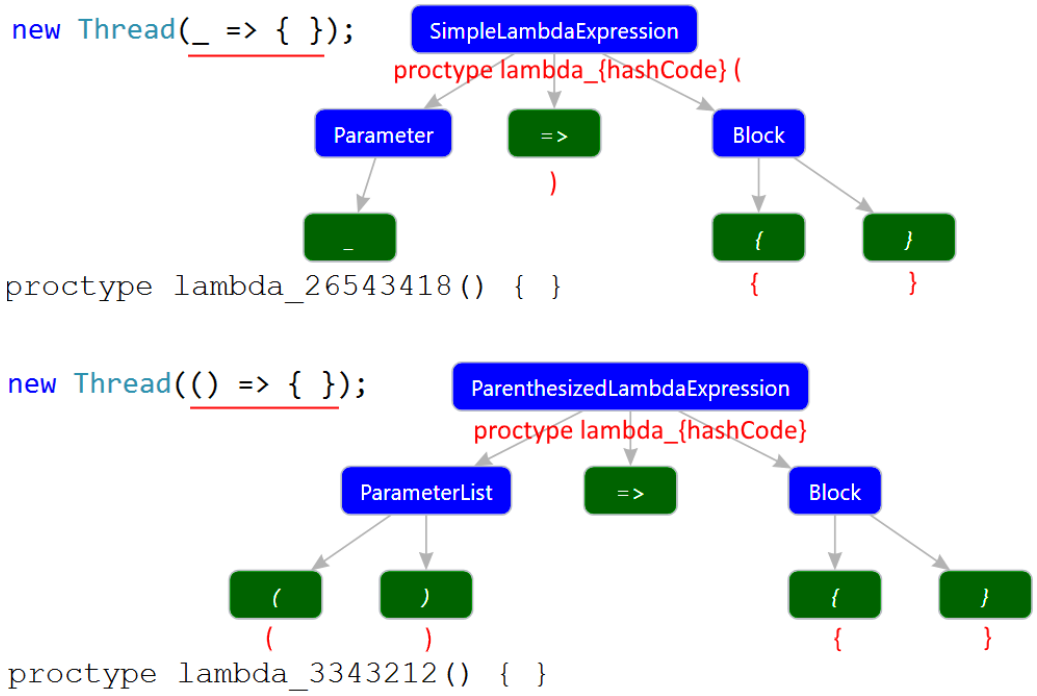


Fig. 14. Promela code generation for thread delegates

When visiting nodes like IfStatement and ElseClause, an algorithm for generating code is provided in Figure 16. A non-deterministic control flow is generated in Promela both by the if condition and by the else alternative branch, since all the variants of possible executions as steps in the control flow are necessary for further verification.

When visiting nodes of type WhileStatement, an algorithm for generating code on Promela is provided in Figure 17. A non-deterministic control flow is generated on Promela both with the do(while) clause and with the break operator, which may be absent in the C# source code since we need all variants of the possible development of program behavior and due to Promela blocking guarded condition semantic [25].

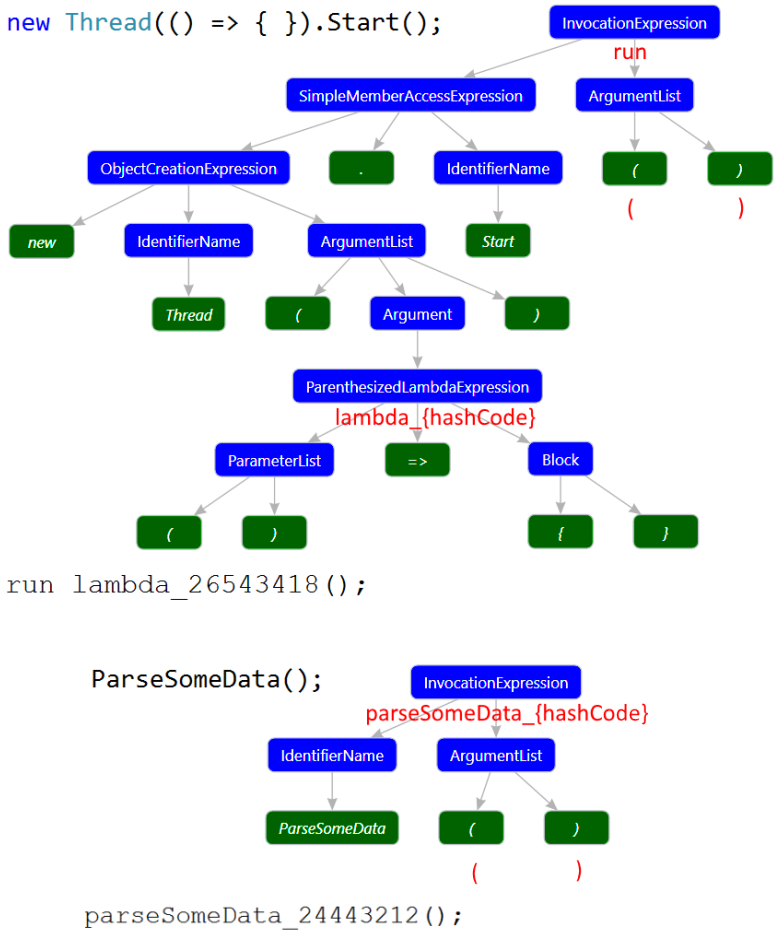
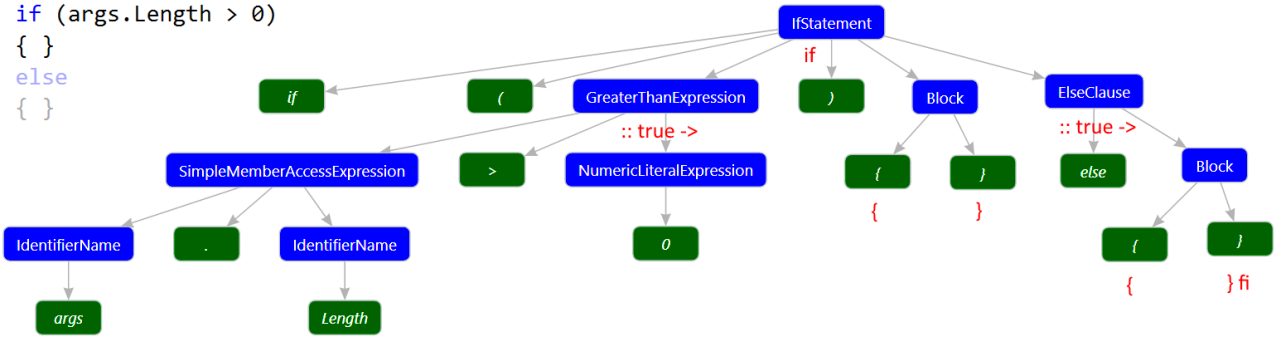


Fig. 15. Promela code generation for Invocation




```
if
:: true -> {
    printf("args.Length > 0");
}
:: true -> {
    printf("else");
}
fi
```

Fig. 16. Promela code generation for IfStatement and ElseClause

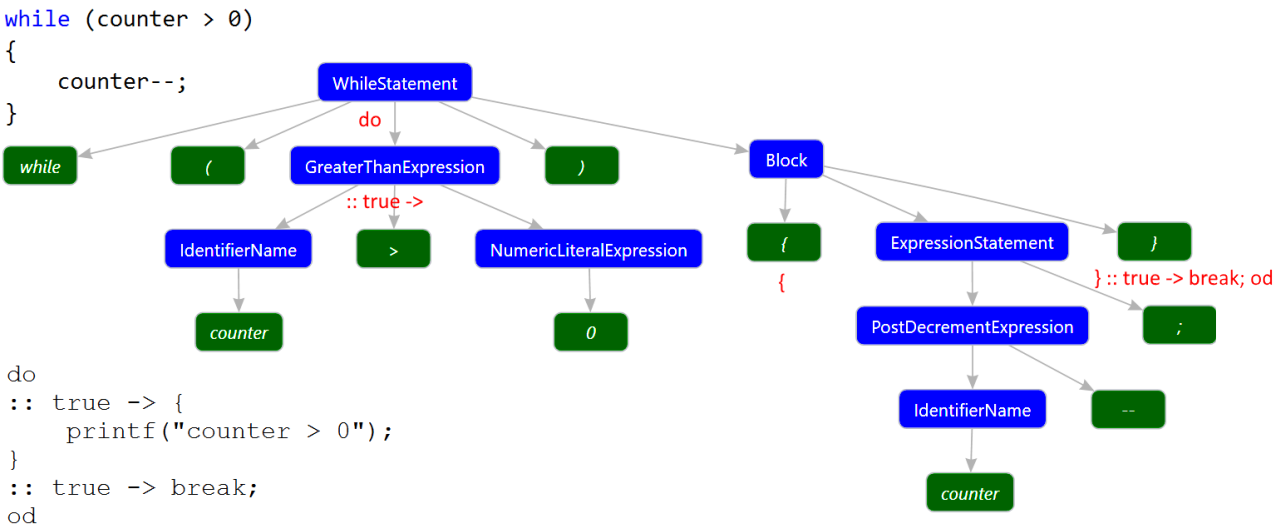


Fig. 17. Promela code generation for WhileStatement

3.3. Thread waiting and its modeling

Figure 18 shows some possible ways to wait for threads termination in C# code.

```
var thread = new Thread(_ => { });
thread.Start();
thread.Join();

var task = Task.Run(() => { });
task.GetAwaiter().GetResult();

var taskResult = Task.Run(() => "result");
var result = taskResult.Result;

var taskAsync = Task.Run(() => { });
await taskAsync;
```

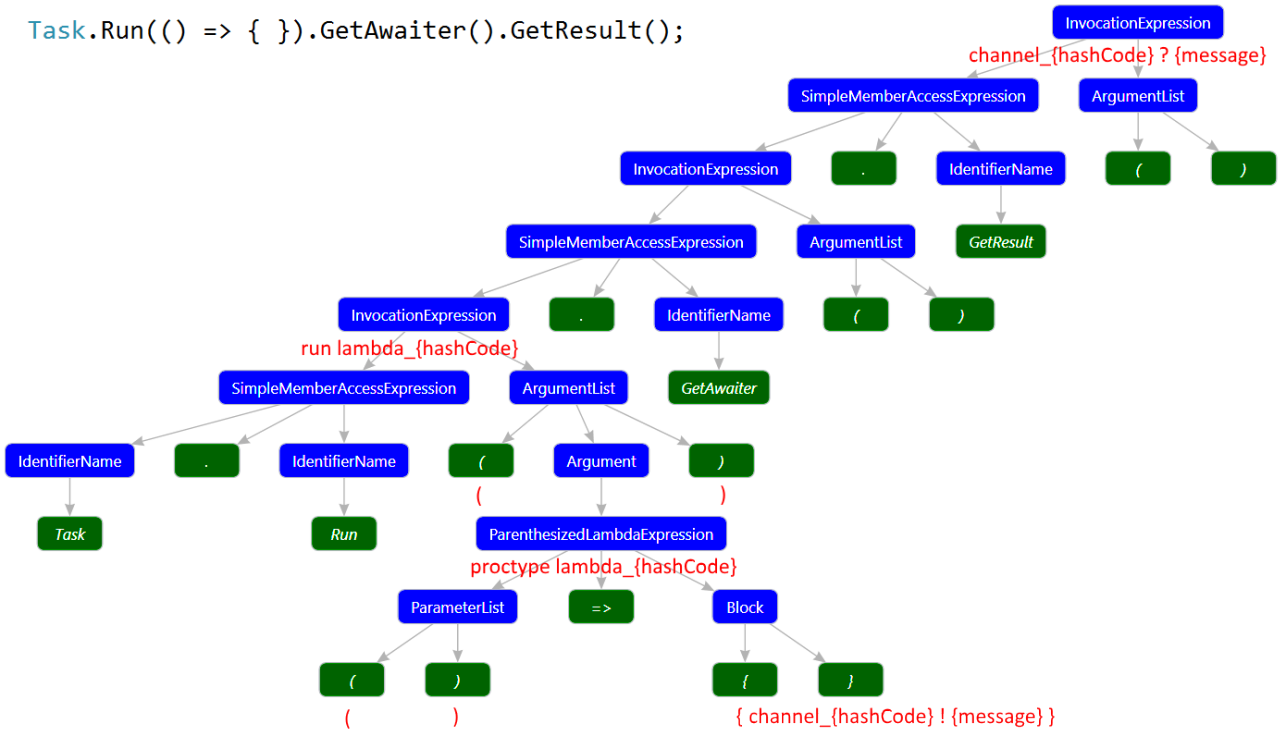
Fig. 18. Ways to wait for threads in C#

When modeling of waiting for the completion of C# threads in Promela, we use message channels. For each child task, we create a separate message channel with a buffer size of 1. This buffer size is chosen so that the child processes should be executable when sending synchronization messages at the end of their work (computation body). The channel buffer size larger than 1 is redundant, since for each child process an individual channel is created, and the buffer size equals to 0 (the rendezvous channel) will make the child process impracticable if the parent does not want to wait for its completion.

Each child process sends a message according to a specific pattern to its message channel at the end of the execution. The parent process, in turn, can stop its execution by calling the receive statement from this channel along with the same pattern. Also, this process will resume its work as soon as a message from a finished child appears on the channel.

To cover all the considered ways of waiting for threads in C#, some additional rules for syntactic nodes of the InvocationExpression type are added using the similar pattern, that rules find calls to methods which trigger a new thread. Only method names and object types here should be changed, for example, the TaskAwaiter type and the GetResult method, or the Thread type and the Join method.

For code generation, following the scheme for simulating awaiting when generating definitions of new processes, we add an appropriate channel in front of them with the same hash code as the process, and then at the end of the process body, we generate a message send to this channel (Figure 19).




```
mtype = { thread_is_done };

chan channel_43603258 = [1] of { mtype }

proctype lambda_43603258()
{
    channel_43603258 ! thread_is_done;
}

active proctype main_39530145()
{
    run lambda_43603258();
    channel_43603258 ? thread_is_done;
}
```

Fig. 19. Promela code generation to model the thread awaiting

3.4. Blocking and its modeling

To model the blocking, we consider modeling operations with the Semaphore class due to the generality of the locking process with it. This approach can be extended to other synchronization primitives and patterns. Figure 20 shows some of the methods for accessing semaphores in C# language.

```
var semaphore = new Semaphore(1, 1);
semaphore.WaitOne();
semaphore.Release();
```

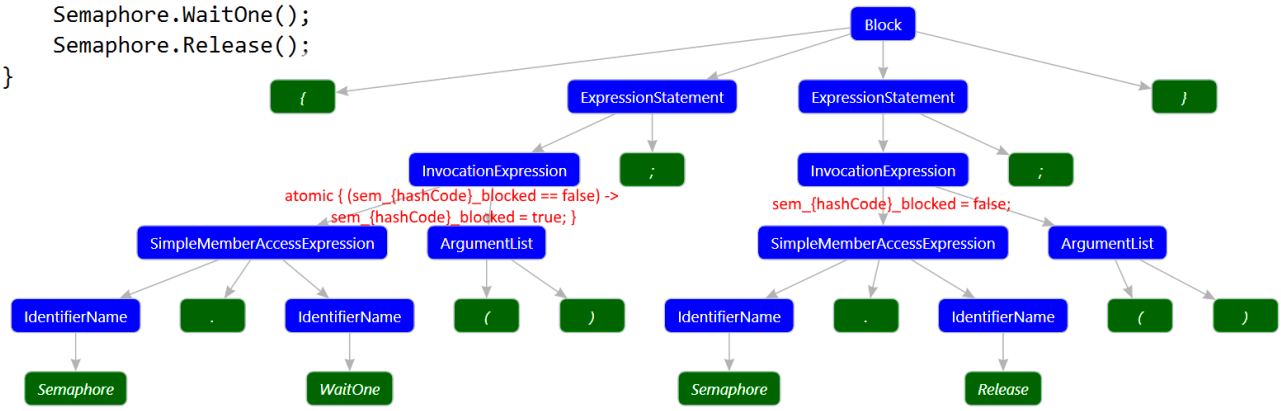
Fig. 20. Ways to create and use semaphores in C#

When modeling a semaphore, we add a shared resource, in the form of a global variable, for which wait and release actions will be modeled.

When we proceed to await and resource capture modeling, we check the resource value, and when it is considered free, then we change this value. These two steps should be concluded in one indivisible operation. For this, we use the possibility of Promela language to wrap the code into an atomic block. When we model the release of a resource, we assign the value to the variable which is considered to be free and a possible blocked code in some different process can continue to run [22] (and it is a subject to do further checks of it).

To cover the methods of interaction with semaphores in C#, additional rules are added for syntactic nodes of the InvocationExpression type using a similar pattern we did for finding calls to methods that trigger a new thread. We should change only the method name and object type: the Semaphore type, the WaitOne and Release methods (Figure 21).

```
private static readonly Semaphore Semaphore = new Semaphore(1, 1);
private static void Main(string[] args)
{
    Semaphore.WaitOne();
    Semaphore.Release();
}
```




```
bool sem_63390070_blocked = false;
active proctype main_33639718()
{
    atomic {
        (sem_63390070_blocked == false) -> sem_63390070_blocked = true;
    }
    sem_63390070_blocked = false;
}
```

Fig. 21. Generating Promela code for C# Semaphore

3.5. Creating and using WCF services and its modeling

Figure 22 shows how to create and run a WCF service in C# code. The service here contains two things – a contract (an interface) and an implementation so that the service publishes the contract and clients can call its methods by using proxy-classes based on the contract interface.

```
[ServiceContract]
internal interface IService1
{
    [OperationContract]
    string GetData(int value);
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple)]
internal class Service1 : WcfService1.IService1
{
    public string GetData(int value) => string.Empty;
}

internal class Program
{
    private static void Main(string[] args)
    {
        var baseAddress = new Uri("http://localhost:8080/myservice");
        using (var host = new ServiceHost(typeof(Service1), baseAddress))
        {
            host.Open();
        }
    }
}
```

Fig. 22. Creating and running a WCF service in C#

Figure 23 shows how to create and use a WCF-service client. It connects to an endpoint where the service is operating and calls the service methods.

```
[System.CodeDom.Compiler.GeneratedCode("System.ServiceModel", "4.0.0.0")]
[System.ServiceModel.ServiceContract(ConfigurationName = "ServiceReference1.IService1")]
public interface IService1...

[System.Diagnostics.DebuggerStepThrough()]
[System.CodeDom.Compiler.GeneratedCode("System.ServiceModel", "4.0.0.0")]
public partial class Service1Client ...

internal class Program
{
    private static void Main(string[] args)
    {
        var binding = new BasicHttpBinding();
        var address = new EndpointAddress("http://localhost:8080/myservice");
        var client = new Service1Client(binding, address);
        client.GetData(0);
    }
}
```

Fig. 23. A way to create and use a WCF client in C#

When modeling the WCF services, here we consider two models – a single-threaded and a

multi-threaded server. In C# code both models are implemented by using special annotations (see "ConcurrencyMode=" in Figure 22), in Promela code we model them by generating code to communicate with the client process in the same server process or by using an additional one.

The single-threaded service runs in an infinite loop, receives a message from a client via a rendezvous service channel (buffer size is 0), and there is an indication of a method being called in this message. Then is made a comparison of the requested method with the predefined service contract and the execution of the resolved method in the same process which get requests. The multi-threaded service model is similar, except that each handle of the service method occurs in a separate process.

The client model is synchronous. It sends a message to the service channel with the name of the method being called. Then it waits for a response from the service on a separate rendezvous channel.

When generating the model code for the client, we search for an endpoint address it refers and a contract it implements, generate the channels followed by a hash code of the above combination and the messages for service call emulation (Figure 24).

```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost:8080/myservice");
var client = new Service1Client(binding, address);
client.GetData(0);


mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

active proctype main_63390070()
{
    iservice1_1995207842 ! getdata_1995207842;
    iservice1_1995207842_answer ? getdata_1995207842;
}
```

Fig. 24. Generating Promela code for a WCF service client

When generating the model code for the service, a search for the service contract (the interface to be implemented) is performed to generate the corresponding methods.

To determine the type of service, the ServiceBehavior attribute of the service class is analyzed. If the argument ConcurrencyMode is set to Multiple, then the multi-threaded model is generated (Figure 25), otherwise the single-threaded one (Figure 26).

```
var baseAddress = new Uri("http://localhost:8080/myservice");
//ConcurrencyMode = Multiple
using (var host = new ServiceHost(typeof(Service1), baseAddress))
{
    host.Open();
}

mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

proctype getdata_1995207842_handler()
{
    printf("getdata");
}

active proctype main_58400626()
{
    mtype service_message;
    do
    :: true -> {
        iservice1_1995207842 ? service_message;
        if
        :: service_message == getdata_1995207842 -> {
            run getdata_1995207842_handler();
            iservice1_1995207842_answer ! service_message;
        }
        fi
    }
    od
}
```

Fig. 25. Generating Promela code for a multi-threaded WCF service in C#

```
var baseAddress = new Uri("http://localhost:8080/myservice");
//ConcurrencyMode = Single
using (var host = new ServiceHost(typeof(Service1), baseAddress))
{
    host.Open();
}

mtype = { getdata_1995207842 };

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

active proctype main_58400626()
{
    mtype service_message;
    do
    :: true -> {
        iservice1_1995207842 ? service_message;
        if
        :: service_message == getdata_1995207842 -> {
            printf("getdata");
            iservice1_1995207842_answer ! service_message;
        }
        fi
    }
    od
}
```

Fig. 26. Generating Promela code for a single-threaded WCF service in C#

4. Verification of distributed system properties

The verification process includes the generation of model code, control variables and rules expressed in LTL, and then model checking (validation of the model correctness).

In this section, we show that by means of generating special control variables and simple LTL formulas, it is possible to model and check some distributed system properties. This process of generation applies to each needed construction as described in the previous section in code independently, so for a particular real program we generate and check different variables and Promela constructions and we do not need to generate complex formulas and structures – we verify the generated model program with respect to all the generated LTL formulas step by step.

4.1. Data race

```
var localList = new List<double>();
Parallel.For(1, 100, _ => localList.Add(0));
```

```
mtype = { thread_complete };
chan awaiter_65946577 = [1] of { mtype }

bool var_458175731_modified = false;
int var_458175731 = 0;

ltl race_conditions_check {
    [] (!var_458175731_modified -> var_458175731 == 0) }

active proctype main_9369539()
{
    do
    :: true -> {
        run lambda_65946577();
    }
    :: true -> break;
    od
}

proctype lambda_65946577()
{
    var_458175731_modified = true;

    var_458175731++;
    var_458175731--;

    var_458175731_modified = false;

    awaiter_65946577 ! thread_complete;
}
```

Fig. 27. A generated Promela code to check the data race using the example of a collection change

The data race problem is the change of object state from different processes (threads) simultaneously that can spoil the correct value of the object.

To model the change we create a pair of control variables $var\_hashcode\_modified = false$ and $var\_hashcode = 0$. The first is set to true before the object is changed, and then to false after the change. The object change itself is modeled using two consecutive operators to increment and decrement the second variable. When the code is correct, this variable should always remain 0.

Thus, data race verification here – is generating and checking the LTL rule (1).

$$G(!var\_hashcode\_modified \rightarrow var\_hashcode == 0) \tag{1}$$

It means *"always now and in the future, from the fact that the variable does not change, it implies that its reference value is zero"*.

An example of generated Promela code to verify the correct changes of a collection object variable that implements the ICollection <T> interface is shown in Figure 27.

4.2. Improper blocking usage

When modeling a semaphore object, a global shared resource $sem\_hashcode\_blocked$ is used, therefore, when verifying the usage of the semaphore we only need to generate the corresponding LTL rule in the form (2).

$$G(sem\_hash\_code\_blocked \rightarrow FG(!sem\_hash\_code\_blocked)) \tag{2}$$

It means *"always now and in the future, if the semaphore is locked, then once now or in the future it will be released forever"*.

Note that the rule is stronger than, for example, a rule $G(sem\_hash\_code\_blocked \rightarrow F(!sem\_hash\_code\_blocked))$ because that semaphore object can be used multiple times and we need to ensure that the object finally is stay unlocked if it became locked.

An example of generated Promela code to check the use of a semaphore is shown in Figure 28.

```
var semaphore = new Semaphore(1, 1);
semaphore.WaitOne();
semaphore.Release();

bool sem__556094084_blocked = false;

ltl sem__556094084_correct {
    [] (sem__556094084_blocked == true -> <>[] (sem__556094084_blocked == false)) }

active proctype main_48266778()
{
    atomic {
        (sem__556094084_blocked == false) -> sem__556094084_blocked = true;
    }
    sem__556094084_blocked = false;
}
```

Fig. 28. Generated Promela code to check a semaphore usage

4.3. Deadlock when accessing an WCF service in a microservice system

Microservices now is a popular concept for scalable [26] applications architectures. Services can act together to solve a huge task by dividing it into some small parts and providing message passing ability between the units. So, a WCF service can be considered as a microservice, if it acts as a service when it receives a task from a client and, in the same time, acts as a client to ask a different service in a chain to get some needed information to the task solving. However, in the chain of service calls, we can get a deadlock when we request a service which is waiting for some data from us. Deadlock can be checked as a liveness condition of the system.

When modeling the service call, we add a separate control variable $service\_call\_hash\_code = false$ before each call to the service method. After receiving a response from the service, we set its value to true.

To verify a deadlock when accessing a service, we generate the LTL rule in the form (3).

$$F(service\_call\_hash\_code == true) \tag{3}$$

It means *"once in the future a response from the service will be received"* (Figure 29).

Using this type of paired constructions of assignments, we ensure that each call to a service function has the corresponding return due to the nature of synchronous calls in WCF. In a microservice chain it is usually hard to capture the logic of corresponding calls (after a call to a service there can exist an inner call to a different service and another next call to another

```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost:8080/myservice");
var client = new Service1Client(binding, address);
client.GetData(0);

mtype = { getdata_1995207842 };

bool service_call__364229018 = false;

chan iservice1_1995207842 = [0] of { mtype };
chan iservice1_1995207842_answer = [0] of { mtype };

ltl deadlocks_check { <> (service_call__364229018 == true) }

active proctype main_42353227()
{
    printf("wcf call GetData");

    iservice1_1995207842 ! getdata_1995207842;
    iservice1_1995207842_answer ? getdata_1995207842;

    service_call__364229018 = true;
}
```

Fig. 29. Generated Promela code to check for a deadlock when accessing a WCF service

service and so on), and by expecting a call return for all calls we can check the correctness of the whole system and easily find a place where the integrity of the service calls might violate.

5. An example of generated model verification

Consider an example of starting and running SPIN verifier with a generated model and an LTL formula that checks it for the data race.

An example source code in C# and a corresponding generated model in Promela are shown in Figure 30.

Figure 31 shows the result of the verification signaling the violation of the absence of data race condition.

Since the verifier found a counterexample, we can build a call trail containing it in the simulation mode with SPIN (Figure 32).

```
public static readonly List<int> GlobalList = new List<int>();
private static void Main(string[] args)
{
    AddToList(0);
    AddToList(0);
}
private static void AddToList(int value)
{
    Task.Run(() => GlobalList.Add(value));
}
```

```
mtype = { thread_complete };

chan awaiter_66898905 = [1] of { mtype }

bool var_31747823_modified = false;
int var_31747823 = 0;

ltl race_conditions_check {
    [] (!var_31747823_modified -> var_31747823 == 0) }

active proctype main_20561848()
{
    run lambda_66898905();
    run lambda_66898905();
}

proctype lambda_66898905()
{
    var_31747823_modified = true;
    var_31747823++;
    var_31747823--;
    var_31747823_modified = false;
    awaiter_66898905 ! thread_complete;
}
```

Fig. 30. Generated source code for checking the race condition

```
./spin -a src.pml
ltl race_conditions_check: [] ((! (! (var_1275380733_modified))) || ((var_1275380733==0)))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000 -a -N race_conditions_check
Pid: 2760
warning: only one claim defined, -N ignored
pan:1: assertion violated !( !(( !( !(var_1275380733_modified))||(var_1275380733==0)))) (at depth 20)
pan: wrote src.pml.trail

(Spin Version 6.4.2 — 8 October 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
never claim + (race_conditions_check)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 52 byte, depth reached 25, errors: 1
27 states, stored
6 states, matched
33 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.002 equivalent memory usage for states (stored*(State-vector + overhead))
0.291 actual memory usage for states
128.000 memory used for hash table (-w24)
0.534 memory used for DFS stack (-m10000)
128.730 total actual memory usage

pan: elapsed time 0 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Fig. 31. Verification result indicating the violation of the absence of data race condition

```
spin -p -s -r -X -v -n123 -l -g -k src.pml.trail -u10000 src.pml
ltl race_conditions_check: [] ((! (! (var_1275380733_modified))) || ((var_1275380733==0)))
starting claim 2
using statement merging
MSC: ~G line 4
1: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
Never claim moves to line 4 [(1)]
Starting lambda_14476932 with pid 2
2: proc 0 (main_9021196:1) src.pml:12 (state 1) [(run lambda_14476932())]
3: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
Starting lambda_14476932 with pid 3
4: proc 0 (main_9021196:1) src.pml:13 (state 2) [(run lambda_14476932())]
5: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
6: proc 2 (lambda_14476932:1) src.pml:18 (state 1) [var_1275380733_modified = 1]
7: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
8: proc 2 (lambda_14476932:1) src.pml:20 (state 2) [var_1275380733 = (var_1275380733+1)]
9: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
10: proc 2 (lambda_14476932:1) src.pml:21 (state 3) [var_1275380733 = (var_1275380733-1)]
11: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
12: proc 1 (lambda_14476932:1) src.pml:18 (state 1) [var_1275380733_modified = 1]
13: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
14: proc 2 (lambda_14476932:1) src.pml:23 (state 4) [var_1275380733_modified = 0]
15: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
16: proc 2 (lambda_14476932:1) src.pml:25 (state 5) [awaiter_14476932!thread_complete]
17: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
18: proc 2 terminates
19: proc - (race_conditions_check:1) _spin_nvr.tmp:4 (state 4) [(1)]
20: proc 1 (lambda_14476932:1) src.pml:20 (state 2) [var_1275380733 = (var_1275380733+1)]
MSC: ~G line 3
21: proc - (race_conditions_check:1) _spin_nvr.tmp:3 (state 1) [(!((!(!(var_1275380733_modified))||(var_1275380733==0))))]
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!((!(!(var_1275380733_modified))||(var_1275380733==0)))))
#processes: 2
21: proc 1 (lambda_14476932:1) src.pml:21 (state 3)
21: proc 0 (main_9021196:1) src.pml:14 (state 3)
21: proc - (race_conditions_check:1) _spin_nvr.tmp:3 (state 2)
3 processes created
Exit-Status 0
```

Fig. 32. Counterexample in the simulation mode

6. Conclusion

As the result of the work, we created a possible extension of the C# language compiler toolset for the formal verification of parallel and distributed applications, which:

- transforms a C# program into an actor model in Promela;
- creates control variables;
- generates LTL formulas for checking:
  - deadlocks;
  - data races;
  - synchronization errors;
- verifies the generated model according to the generated rules.

The aim of the paper was not to create a complete solution for proving parallel and distributed software, we just wanted to show a possible method that uses AST C# code model to Promela model transformation, generates LTL formulas and checks them with the SPIN verifier.

The process is based on analyzing Roslyn structures of full C# grammar programs. The results can be used in static checkers to prove some properties of sophisticated code.

The method we used can be a bridge from real programs to its formal models with the automatic transformation between them.

Bibliography

1. Clarke Jr, Edmund M., et al. Model checking. Cyber-Physical Systems, 2018.
2. Staroletov, Sergey. Basics of Software Testing and Verification [in Russian]. Lanbook, Saint Petersburg, 2018. -344p. ISBN 978-5-8114-3041-3.
3. Richter, Jeffrey. CLR via C#. Pearson Education, 2012. ISBN 978-0-7356-6876-8
4. Home repository for .NET Core. Available from: https://github.com/dotnet/core
5. McMurtry, Craig, et al. "Windows Communication Foundation Unleashed (WCF)." (2007).
6. The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. Available from: https://github.com/dotnet/roslyn
7. Harrison, Nick. Code Generation with Roslyn. Apress, 2017. ISBN 978-1-4842-2211-9
8. Vasiliev, Sergey. Introduction to Roslyn and its use in program development. Available from: *https* : *//www.viva*64.*com/en/b/*0399/
9. Burghardt, Jochen, et al. "ACSL By Example." (2016).
10. Hewitt, Carl, Peter Bishop, and Richard Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence." Advance Papers of the Conference. Vol. 3. Stanford Research Institute, 1973.
11. Agha, Gul A. Actors: A model of concurrent computation in distributed systems. No. AI-TR-844. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.
12. Armstrong, Joe. Programming Erlang: software for a concurrent world. Pragmatic Bookshelf, 2013.
13. Vernon, Vaughn. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015.
14. Holzmann, Gerard J. "The model checker SPIN." IEEE Transactions on software engineering 23.5 (1997): 279-295.
15. Pnueli, Amir. "The temporal logic of programs." Foundations of Computer Science, 1977., 18th Annual Symposium on. IEEE, 1977.
16. Proctype – for declaring new process behavior. Promela. Available from: *http* : *//spinroot.com/spin/Man/proctype.html*
17. Michael Barnett, K. Rustan, M. Leino and Wolfram Schulte. "The Spec# programming system: An overview." International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. Springer, Berlin, Heidelberg, 2004.
18. Manuel Fähndrich, Michael Barnett and Francesco Logozzo. "Embedded contract lan-

guages." Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, 2010.

19. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. Proceedings of EuroSys2006. Leuven, Belgium, April 2006. ACM SIGOPS.
20. Koshelev, Vladimir Konstantinovich, Valery Nikolayevich Ignatyev, and Artem Il'ich Borzilov. "C# static analysis framework." Proceedings of the Institute for System Programming of the RAS 28.1 (2016): 21-40.
21. Karpov, Andrey. How PVS-Studio does the bug search: methods and technologies. Available from: https://www.viva64.com/en/b/0466/
22. Staroletov, Sergey. Model of a program as multi-threaded stochastic automaton and its equivalent transformation to Promela model. Ershov informatics conference. PSI Series, 8th edition. International workshop on Program Understanding. Proceedings. – Novosibirsk, 2011. p. 33-38
23. Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 1993.
24. Inline – a stylized version of a macro. Promela. Available from: *http* : *//spinroot.com/spin/Man/inline.html*
25. Do – repetition construct. Promela. Available from: *http* : *//spinroot.com/spin/Man/do.html*
26. Dragoni, Nicola, et al. "Microservices: How to make your application scale." International Andrei Ershov Memorial Conference on Perspectives of System Informatics. Springer, Cham, 2017.

УДК 004.05

Верификация предикатной программы бинарного поиска объекта произвольного типа

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Описывается построение и дедуктивная верификация предикатной программы бинарного поиска, идентичной программе bsearch на языке Си из библиотеки ОС Linux. В языке предикатного программирования определяются новые конструкции для произвольных типов в качестве параметров программ. Для объектов произвольного типа вводятся трансформации кодирования через указатели.

Ключевые слова: *дедуктивная верификация, трансформации программ, бинарный поиск, язык программирования Си, функциональное программирование*.

1. Введение

Алгоритм бинарного поиска является классическим часто используемым алгоритмом. Программа бинарного поиска bsearch из библиотеки ядра ОС Linux максимально универсальна. Допускается произвольный тип элементов. В связи с этим размер элементов в байтах задается параметром. Операция сравнения элементов также поставляется как параметр программы bsearch. В дополнение к этому, из соображений эффективности используются битовые операции и адресная арифметика. Перечисленные особенности определяют повышенную сложность дедуктивной верификации программы bsearch. В частности, произвольный размер элементов и подстановка программы параметром существенно затрудняют применение набора инструментов FramaC – Why3 [1, 2], успешно используемых для дедуктивной верификации многих программ ядра ОС Linux.

В настоящей работе проводится верификации данной программы применением следующей технологии. В рамках предикатного программирования можно воссоздать любую императивную программу из класса программ-функций [11]. Строится предикатная программа, эквивалентная исходной императивной программе. Далее эта программа получается из предикатной программы применением некоторого набора оптимизирующих трансформаций. Данная технология построения предикатной программы, применения оптимизирующих трансформаций и дедуктивной верификации представлена в работе [10].

Дедуктивная верификация предикатной программы существенно проще и быстрее в сравнении с дедуктивной верификацией императивной программы [9].

Во втором разделе дается краткое описание языка предикатного программирования. Определяется синтаксис и семантика гиперфункций. Метод дедуктивной верификации описывается в третьем разделе. В четвертом разделе определено построение эквивалентной предикатной программы bsearch. В следующем разделе описывается процесс оптимизирующей трансформации программы. Процесс дедуктивной верификации предикатной программы в системах Why3 [25] и PVS [24] определяется в шестом разделе. Далее обзор работ. В заключении суммируются результаты работы. В расширенной версии данной статьи [26] включены приложения, где подробно документируется процесс дедуктивной верификации.

2. Язык предикатного программирования

Полная предикатная программа состоит из набора рекурсивных *предикатных программ* на языке P [5] следующего вида:

```
<имя программы>(<описания аргументов>: <описания результатов>)
pre <предусловие>
post <постусловие>
measure <выражение>
{ <оператор> }
```

Предусловие и постусловие являются формулами на языке исчисления предикатов. Они обязательны при дедуктивной верификации [8, 12, 23]. Мера задается только для рекурсивных программ и используется для доказательства их завершения.

Ниже представлены основные конструкции языка P: оператор присваивания, блок (оператор суперпозиции), параллельный оператор, условный оператор, вызов программы и описание переменных, используемое для аргументов, результатов и локальных переменных.

```
<переменная> = <выражение>
{<оператор1>; <оператор2>}
<оператор1> || <оператор2>
if (<логическое выражение>) <оператор1> else <оператор2>
<имя программы>(<список аргументов>: <список результатов>)
<тип> <пробел> <список имен переменных>
```

Всякая переменная характеризуется *типом* – множеством допустимых значений. *Описание типа* **type** T(p) = D с возможными параметрами p связывает имя типа T с его изображением D. Типы **bool**, **int**, **real** и **char** являются *примитивными*. Значением типа **array**(T_e, T_i) является *массив* с *элементами массива* типа T_e и *индексами* конечного типа T_i.

Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами.

Пусть E(x) – логическое выражение. Тип **subtype**(T x**:** E(x)) определяет *подтип* типа T при истинном предикате E(x), т.е. множество {x∈T | E(x)}. Определенный в языке P тип целых чисел **nat** представляется описанием:

type nat = **subtype**(**int** x: x ≥ 0) .

Допускаются подтипы, параметризуемые переменными. Примером является тип *диапазона* целых чисел:

type Diap(**nat** n) = **subtype**(**int** x: x ≥ 1 & x ≤ n) .

В языке P для изображения типа диапазона используется конструкция 1..n.

Тип может быть передан программе в качестве параметра. Для такого типа возможны лишь операции, передаваемые в виде функций другими параметрами программы. В целях верификации свойства такого типа задаются в виде теории. См. раздел 4.2.

Описания типов переменных являются частью спецификации программы. Описание переменной T x есть утверждение x∈T, которое становится частью предусловия, если x – аргумент предикатной программы, или частью постусловия, если x – результат программы. При этом утверждение x∈T обычно не пишется в составе предусловия или постусловия, хотя предполагается.

Гиперфункция – программа с несколькими *ветвями* результатов. Гиперфункция B(x**:** y**:** z) имеет две ветви результатов y и z. Исполнение гиперфункции завершается одной из ветвей с вычислением результатов по этой ветви; результаты других ветвей не вычисляются.

Вызов гиперфункции записывается в виде B(x**:** y #M1**:** z #M2). Здесь M1 и M2 – метки программы, содержащей вызов. Операторы перехода #M1 и #M2 встроены в ветви вызова. Исполнение вызова либо завершается первой ветвью с вычислением y и переходом на метку M1, либо второй ветвью с вычислением z и переходом на метку M2.

Вызов гиперфункции может комбинироваться с операторами обработки ветвей:

B(x**:** y #M1**:** z #M2) **case** M1: C(y**:** u) **case** M2: D(z**:** u) .

Конструкция вида B(x**:** y #M1**:** z #M2); M1**: ...** может быть представлена оператором B(x**:** y**:** z #M2).

Формально гиперфункция определяется через предикатную программу следующего вида:

```
B(x: y, z, e)
pre P(x) post e = E(x) & (E(x) ⇒ S(x, y)) & (¬E(x) ⇒ R(x, z))
{ ... };
```

Здесь x, y и z – непересекающиеся возможно пустые наборы переменных; P(x), E(x), S(x, y) и R(x, z) – логические утверждения. Предположим, что все присваивания вида e = **true** и e = **false** – последние исполняемые операторы в программе B. Программа B может быть заменена следующей программой в виде *гиперфункции*:

```
B(x: y #1:  z #2)
pre P(x)   pre 1: E(x) post 1: S(x, y) post 2: R(x, z)
{ … };
```

В теле гиперфункции каждое присваивание e = **true** заменено оператором перехода #1, а e = **false** – на #2. *Метки* 1 и 2 – дополнительные параметры, определяющие два различных *выхода* гиперфункции.

Спецификация гиперфункции состоит из двух частей. Утверждение после "**pre** 1" есть предусловие первой ветви; предусловие второй ветви – отрицание предусловия первой ветви. Утверждения после "**post** 1" и "**post** 2" есть постусловия для первой и второй ветвей, соответственно.

Аппарат *гиперфункций* является более общим и гибким по сравнению с известным механизмом обработки исключений, например, в таких языках, как Java и C++. Традиционные подходы в реализации обработки аварийных ситуаций предполагают заведение дополнительных структур, усложняющих программу. Этого удается избежать при использовании гиперфункций. Использование гиперфункций делает программу короче, быстрее и проще для понимания [12, 13]. Подробнее в работе [6, разд. 4], подразделе «баланс информационных и управляющих связей».

В языке предикатного программирования P [5] нет указателей, серьезно усложняющих программу. Вместо указателей используются объекты алгебраических типов: списки и деревья. Предикатная программа существенно проще в сравнении с императивной программой, реализующей тот же алгоритм. Эффективность предикатных программ достигается применением *оптимизирующих трансформаций* [4]. Они определяют отличную от классической оптимизацию среднего уровня с переводом предикатной программы в эффективную императивную программу.

Базовыми трансформациями являются:

- склеивание переменных, реализующее замену нескольких переменных одной;
- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- кодирование объектов алгебраических типов (списков и деревьев) при помощи массивов и указателей.

3. Дедуктивная верификация

Предикатная программа относится к классу *программ-функций* [11]. Программа-функция должна всегда **нормально завершаться** с получением результата, поскольку бесконечно работающая и невзаимодействующая программа бесполезна.

Спецификацией предикатной программы H(x**:** y) являются два предиката: *предусловие* P(x) и *постусловие* Q(x, y). Спецификация записывается в виде: [P(x), Q(x, y)].

Для языка **P0** построена формальная операционная семантика ℛ(H)(x, y) и доказано тождество ℛ(H) = H [14]. На базе языка **P0** последовательным расширением и сохранением тождества ℛ(H) = H построен язык предикатного программирования **P** [5].

Тотальная корректность программы относительно спецификации определяется формулой:

$$H(x{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)] \cong \forall x.\ P(x) \Rightarrow [\forall y.\ H(x{:}\ y) \Rightarrow Q(x, y)\ ]\ \&\ \exists y.\ H(x{:}\ y)$$

Формулу тотальной корректности будем представлять в виде правила **COR**:

$$\textbf{COR}:\ \frac{\begin{array}{l}\forall x,y.\ P(x)\ \&\ H(x{:}\ y) \Rightarrow Q(x, y);\\ \forall x.\ P(x) \Rightarrow \exists y.\ H(x{:}\ y)\end{array}}{H(x{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)]}$$

Для базисных операторов (параллельного, условного и суперпозиции) разработана универсальная система правил доказательства их корректности [7, 16], в том числе и при наличии рекурсивных вызовов, существенно упрощающая процесс доказательства по сравнению с исходной формулой тотальной корректности. Корректность правил доказана [3] в системе PVS. В системе предикатного программирования реализован генератор формул корректности программы. Часть формул доказывается автоматически SMT-решателем CVC4. Оставшаяся часть формул генерируется для системы интерактивного доказательства PVS [24]. Данный метод опробован для дедуктивной верификации более сотни программ [8, 12, 15].

Предположим, что наборы переменных x, y и z не пересекаются, а x может быть пустым. Ниже приведены некоторые правила доказательства корректности операторов.

$$\textbf{QP}:\ \frac{B(x{:}\ y)\ \textbf{corr}\ [P(x), Q(x, y)];\ C(x{:}\ z)\ \textbf{corr}\ [P(x), R(x, z)];}{\{B(x{:}\ y)\ ||\ C(x{:}\ z)\}\ \textbf{corr}\ [P(x), Q(x, y)\ \&\ R(x, z)]}$$

$$\textbf{QC:}\ \frac{\begin{array}{l}B(x\textbf{:}\ y)\ \textbf{corr}\ [P(x)\ \&\ E(x),\ Q(x,\ y)];\\ C(x\textbf{:}\ z)\ \textbf{corr}\ [P(x)\ \&\ \neg E(x),\ Q(x,\ y)]\end{array}}{\{\textbf{if (}E(x))\ B(x\textbf{:}\ y)\ \textbf{else}\ C(x\textbf{:}\ y)\}\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Далее следует правило для частного случая оператора суперпозиции, соответствующего *сведению к более общей задаче* C(x, z**:** y).

$$\textbf{RB:}\ \frac{\begin{array}{l}\forall z\ C(x,\ z\textbf{:}\ y)\ \textbf{corr*}\ [P_C(x,\ z),\ Q_C(x,\ y)];\\ P(x) \Rightarrow P_B(x)\ \&\ P^*_C(x,\ B(x));\\ \forall y\ (\ P(x)\ \&\ Q_C(B(x),\ y) \Rightarrow Q(x,\ y)\ );\end{array}}{C(x,\ B(x)\textbf{:}\ y)\ \textbf{corr}\ [P(x),\ Q(x,\ y)]}$$

Запись вида z = B(x) является эквивалентом B(x**:** z). Истинность трех посылок правила **RB** гарантирует корректность следующей программы:

H(x**:** y) **pre** P(x) **post** Q(x, y) { C(x, B(x)**:** y) }

В случае рекурсивного вызова C(x, B(x)**:** y) обозначение **corr*** означает, что первая посылка опускается, а $P^*_C(x, B(x))$ заменяется на $P_C(x, B(x))\ \&\ m(x) < m(v)$. Здесь m – натуральная функция *меры*, строго убывающая на аргументах рекурсивных вызовов, а v обозначает аргументы рекурсивной программы C.

4. Построение программы бинарного поиска

4.1. Постановка задачи

Имеется следующая программа бинарного поиска на языке Си:

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
        int (*cmp)(const void *key, const void *elt))
{       const char *pivot;
        int result;
        while (num > 0) {
                pivot = base + (num >> 1) * size;
                result = cmp(key, pivot);
                if (result == 0)
                        return (void *)pivot;
                if (result > 0) {
                        base = pivot + size;
                        num--;
                }
                num >>= 1;
        }
        return NULL;
}
```

Имеется описание для пользователей: https://elixir.bootlin.com/linux/latest/source/lib/bsearch.c. Программа оперирует элементами произвольного размера size в байтах.

Элемент, доступный по указателю key размером size байтов, ищется в массиве по указателю base с элементами размера size байтов и числом элементов num. Массив упорядочен по возрастанию. Упорядоченность определяется функцией cmp(key, pivot), являющейся параметром программы bsearch. Параметры key и pivot являются указателями на элементы. Значения функции cmp интерпретируются следующим образом:

cmp(key, pivot) > 0 -→ key* > pivot*
cmp(key, pivot) < 0 -→ key* < pivot*
cmp(key, pivot) = 0 -→ key* = pivot*

Программа bsearch выдает результатом указатель на элемент key в исходном массиве при условии, что элемент со значение key* существует в массиве. В противном случае результат программы есть NULL.

Требуется построить соответствующую предикатную программу, провести ее дедуктивную верификацию и трансформировать ее в эффективную императивную программу, эквивалентную приведенной выше.

4.2. Предикатная программа бинарного поиска

Определим тип элементов массива:

type E;

Тип E произвольный и ограничен лишь размером элементов size. Однако информация о размере size будет существенна лишь на стадии реализации соответствующей императивной программы. Принципиально, что для значений типа E недоступны операции, введенные в языке **P**, кроме операции “=” равенства элементов, определенной для всех типов. В предикатной программе мы будем использовать операторы присваивания, которые исчезают при склеивании переменных. Для значений типа E применимы лишь операции, специально для него определенные. В нашем случае это единственная операция сравнения cmp.

Определим тип функции cmp для сравнения двух элементов типа E:

type CMP= **predicate**(E, E**: int**) **pre true post true**;

Данное описание определяет предикатный тип с двумя аргументами типа E и результатом типа **int**. Никаких других ограничений на значения аргументов и результата нет.

Спецификация вида [**true**, **true**] является наиболее общей и абсолютно бесполезной. А другой здесь быть не может, поскольку предусловие и постусловие можно было бы выразить лишь с помощью других операций на типе E, а их нет.

Свойства операции cmp задаются в виде теории в стиле алгебраических спецификаций.

```
theory Compare {
    type E;
    axiom EnotEmpty: ∃ E x. true; // тип E не пустой
    CMP cmp;
    axiom TotalCmp: ∀ E x, y. ∃ int z. cmp(x, y) = z; // тотальность функции cmp
    axiom GrCmp: ∀ E x, y. cmp(x, y) > 0 ⇒ cmp(y, x) < 0;   // симметрия
    axiom TrCmpE: ∀ E x, y, z. cmp(x, y) <= 0 & cmp(y, z) < 0 ⇒ cmp(x, z) < 0;
    axiom TrCmpF: ∀ E x, y, z. cmp(x, y) < 0 & cmp(y, z) <= 0 ⇒ cmp(x, z) < 0;
}
```

В точности этот набор свойств теории Compare был задействован при доказательстве формул корректности предикатной программы bsearch. Тип E не может быть пустым, поскольку содержит элемент, поставляемый аргументом key. Тотальность функции cmp необходима для доказательства тотальности программы bsearch. Остальные свойства используются при доказательстве двух последних формул корректности RB13 и RB23 (см. Разд. 6.2).

Определим тип массива с элементами типа E:

```
type di(nat p, n) = p..p+n-1;
type Ar(nat p, n) = array(E, di(p, n));
```

Здесь n – число элементов массива. Тип di определяет индексы массива: от p до p+n-1. Параметром типа Ar является также тип E. Он не указывается в списке параметров, хотя подразумевается.

Ниже перечислены аргументы программы bsearch, которые для удобства написания программы определены как глобальные переменные.

```
type E;
E key;
CMP cmp;
nat num;
```

Здесь key – элемент типа E, который ищется в исходном массиве base, num – число элементов в массиве base и cmp – функция сравнения.

Программу бинарного поиска bsearch определим в виде гиперфункции:

```
bsearch( Ar(0, num) base : nat p, n, Ar(p, n) b #1 : #2 )
```

Аргумент base – исходный массив длины num с индексами от 0 до num-1. Первая ветвь гиперфункции реализуется при наличии в массиве base элемента, равного key; вторая ветвь – при отсутствии элемента key. Результатом первой ветви является массив b – вырезка массива base от элемента с индексом p до конца массива base, причем b[p] = key. Другой результат n – длина массива b. По второй ветви гиперфункции результатов нет.

Отметим, что в библиотечной программе bsearch на языке Си нет результата n – длины оставшейся части массива. Хотя такой результат нужен – в программе на языке Си он вычисляется после вызова bsearch.

Вырезка b[p..p+n-1] используется в предикатной программе bsearch как аналог указателя, на который вырезка заменяется на этапе оптимизирующей трансформации. Возможен более простой аналог указателя в виде пары (base, p).

Ниже представлена программа bsearch. Спецификация определяется общим предусловием pBsearch, предусловием по первой ветви p1key и постусловием по первой ветви qBsearch. Постусловие по второй ветви отсутствует, так как во второй ветви нет результатов.

```
formula pBsearch(nat p, n, Ar(p, n) b) =
                    p+n=num & ∀ di(p, n) i, j. i < j ⇒ cmp(b[i], b[j])≤0;
formula p1key(nat p, n, Ar(p, n) b) = ∃ di(p, n) j. cmp(key, b[j]) = 0;
formula qBsearch(Ar(0, num) base, nat p, n, Ar(p, n) b) =
                              p+n = num & b = base[p..num-1] & b[p] = key;
```

Предусловия определены в более общем в виде для использования также в другой программе bse. Для программы bsearch параметр p в предусловиях равен нулю.

```
bsearch( Ar(0, num) base : nat p, n, Ar(p, n) b #1 : #2)
pre pBsearch(0, num, base)
pre 1: p1key(0, num, base)
post 1: qBsearch(base, p, n, b)
{   bse(0, num, base : p : #2);
   { n = num – p; b = base[p..p+n-1] }
   #1
};
```

Вызов гиперфункции bse по первой ветви вычисляет индекс p массива base, на котором base[p] = key. При завершении первой ветвью исполняются операторы после вызова. При завершении вызова bse второй ветвью реализуется переход #2, завершающий исполнение гиперфункции bsearch второй ветвью, что означает отсутствие key в массиве base.

Гиперфункция bse для массива b[p: p+n-1] определяет индекс p′, на котором b[p′] = key. Ветви гиперфункции определяются аналогично гиперфункции bsearch.

```
bse(nat p, n, Ar(p, n) b: nat p′ #1 : #2)
pre pBsearch(p, n, b)
pre 1: p1key(p, n, b)
post 1: ∃ nat m. cmp(key, b[p+m]) = 0 & p′ = p+m
measure n
{  if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p′ = p + m  #1 }
   if (result > 0)
         bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p′ : #2)
   else  bse(p, m, b[p..p + m - 1] #1: p′ : #2)
};
```

Здесь штрих в имени p′ означает, что в итоговой императивной программе переменные p и p′ должны быть склеены: p′ заменяется на p.

Алгоритм реализуется по следующей схеме. Представим диапазон p..p + n – 1 в виде:

[p, … , p+m-1] p+m [p+m+1, … , p+n-1]

Если cmp(key, b[p+m]) = 0, то решение: p′ = p+m. Иначе в зависимости от знака cmp(key, b[p+m]) требуемый элемент key ищется в одном из диапазонов: p..p+m-1 или p+m+1..p+n-1.

Итак, полная предикатная программа состоит из программ bsearch и bse.

5. Оптимизирующие трансформации

Определим набор трансформаций, превращающих программу бинарного поиска, состоящую из программ bsearch и bse, в эффективную императивную программу. На первом этапе реализуется трансформация склеивания переменных. Например, операция склеивания num ← n реализует замену всех вхождений переменных n на переменную num.

Склеивание: num ← n.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{   bse(0, num, base: p : #2);
   {num = num – p;  b = base[p..p+num-1]}
   #1
};
```

Склеивание: p ← p′.

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{  if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p : #2)
   else  bse(p, m, b[p..p + m - 1] #1: p : #2)
};
```

На втором этапе проводится замена хвостовой рекурсии циклом в программе bse:

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{loop {
   if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         |p, n, b| =|p+m+1, n - m - 1, b[p+m+1..p+n-1]|
   else  |p, n, b| =|p, m, b[p..p + m - 1]|
 }
};
```

Раскрываются групповые операторы присваивания. При раскрытии первого оператора необходимо поменять порядок присваивания.

```
bse(nat p, n, Ar(p, n) b: p #1 : #2)
{loop {
   if (n=0) #2
   nat m = n / 2;
   E pivot = b[p+m];
   int result = cmp(key, pivot);
   if (result = 0) { p = p + m  #1 }
   if (result > 0)
         { b = b[p+m+1..p+n-1];  p = p+m+1; n = n - m - 1 }
   else  { n = m; b = b[p..p + m - 1] }
 }
};
```

На третьем этапе тело программы bse подставляется на место вызова в bsearch. При этом оператор #1 заменяется на **break**.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{   //bse(0, num, base: p : #2);
    | p, n, b | = | 0, num, base |;
  loop {
    if (n=0) #2
    nat m = n / 2;
    E pivot = b[p+m];
    int result = cmp(key, pivot);
    if (result = 0) { p = p + m; break }
    if (result > 0)
            { b = b[p+m+1..p+n-1];  p = p+m+1; n = n - m - 1 }
    else   { n = m; b = b[p..p + m - 1] }
  };
    {num = num – p;  b = base[p..p+num-1]}
     #1
};
```

В теле цикла заменим n на num, а b на base. Далее операторы за циклом втянем внутрь тела цикла, изменив их порядок.

```
bsearch( Ar(0, num) base : num, nat p, Ar(p, num) b #1 : #2)
{ p = 0; nat num0 = num;
  loop {
    if (num=0) #2
    nat m = num / 2;
    E pivot = base[p+m];
    int result = cmp(key, pivot);
    if (result = 0) {
      p = p + m;
      num = num0 – p ;
      b  = base[p..p+num-1]
      #1
    }
    if (result > 0)
            { base = base[p+m+1..p+ num -1];  p = p+m+1; num = num - m – 1 }
    else   { num = m; base = base[p..p + m - 1] }
  };
};
```

Здесь num0 обозначает значение num в начале исполнения bsearch.

На четвертом этапе трансформаций реализуем кодирование типов и операций с ними через типы и операции языка Си.

Все переменные типа E кодируются указателем на переменную. Причем используется указатель типа **void***. Массив base получает тип **void***. Элемент массива base[p+m] идентифицируется указателем base + m*size. Ниже приведено представление операций и операторов программы bsearch.

```
E pivot = base[p+m]                → void*pivot = base + m*size;
base = base[p..p + m - 1]          → base = base
base = base[p+m+1..p+ num -1]      → base = base + (m+1)*size;
num / 2                            → num >> 1
```

Результатом кодирования объектов является следующая программа.

```
bsearch(void *base, nat num : num, nat p, void *b #1 : #2)
{ p = 0; nat num0 = num;
  loop {
    if (num=0) #2
    nat m = num >> 1;
    void*pivot = base + m*size;
    int result = cmp(key, pivot);
    if (result = 0) {
      b  = base + m*size;
      p = p + m;
      num = num0 – p
      #1
    }
    if (result > 0)
            { base = base + (m+1)*size;  p = p+m+1; num = num - m – 1 }
    else   { num = m; base = base }
  };
};
```

Вычисления указателей не зависят от p. Поэтому все операторы, вычисляющие p, можно удалить. Кроме того, ненужными становятся операторы p = p + m и num = num0 – p. После удаления неиспользуемых вычислений программа преобразуется к виду:

```
bsearch(void *base, nat num :  void *b #1 : #2)
{ loop {
    if (num=0) #2
    nat m = num >> 1;
    void*pivot = base + m*size;
    int result = cmp(key, pivot);
    if (result = 0) {
       b  = base + m*size
      #1
    }
    if (result > 0)
            { base = base + (m+1)*size;  num = num - m – 1 }
    else   num = m;
  };
};
```

Далее превратим гиперфункцию в обычную функцию, совместив два выхода в один. С этой целью выход #2 закодируем оператором b = NULL с учетом того, что нулевой

указатель не появится на первой ветви. Проведем также очевидные упрощения при вычислении указателей. Получим итоговую программу.

```
bsearch(void *base, nat num :  void *)
{ loop {
        if (num=0) return NULL;
        nat m = num >> 1;
        void*pivot = base + m*size;
        int result = cmp(key, pivot);
        if (result = 0) return pivot;
        if (result > 0) { base = pivot + size;  num = num - m - 1 }
        else  num = m;
  };
};
```

Сравним ее с исходной программой на языке Си:

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
              int (*cmp)(const void *key, const void *elt))
{       const char *pivot; int result;
        while (num > 0) {
                pivot = base + (num >> 1) * size;
                result = cmp(key, pivot);
                if (result == 0) return (void *)pivot;
                if (result > 0) { base = pivot + size;  num--; }
                num >>= 1;
        }
        return NULL;
    }
```

Чтобы определить эквивалентность программ, достаточно проверить, что значение переменной num в конце цикла является одинаковым для двух программ при условии, что значения num в начале цикла совпадают для двух программ. Рассмотрим случаи: num = 2*k и num = 2*k + 1. Проведя несложные вычисления для num в начале цикла легко проверить, что значения num в конце цикла для каждого из случаев будут совпадать.

Таким образом, программы эквивалентны. Разумеется, можно было бы построить предикатную программу, которая после трансформации совпадает с исходной программой.

6. Процесс дедуктивной верификации

Теория с формулами корректности (Разд. 6.2) была закодирована на языке спецификаций why3 системы Why3 [25]. При этом массивы вида b[p: p+n-1] всюду были сведены к массивам b[0: 0+n-1]. Как следствие, во многих конструкциях появилось вычитание значения p. Эти изменения нетривиальны. Несмотря на то, что вырезки полноценно

присутствуют в языке программирования whyML, в языке спецификаций why3 и системе доказательства Why3 вырезки никак не поддержаны. Лямбда-функций нет в why3. В итоге вырезки были определены через предикаты sliceL и sliceR.

В рамках системы Why3 в течение полутора дней были доказаны все формулы корректности, кроме RB13 и RB23, наиболее сложных. Причем для доказательства формулы корректности Rp2 были введены промежуточные леммы Rp2_key, Rp2_key0 и KeyEx. Примечательно, что все формулы и леммы были доказаны с помощью одного решателя AltErgo. Попытки доказать RB13 и RB23 за пару дней оказались безуспешными.

Было решено перевести теорию из why3 на язык спецификаций PVS в проекции на RB13 и RB23. Из-за сложной системы ограничений на диапазоны два дня не удавалось пройти семантический контроль (type checking) в PVS. Пришлось изменить спецификацию постусловия программы bse. Доказательство в PVS для формул RB13 и RB23 удалось построить за пару часов.

Сопутствующей целью верификации было определение точных ограничений на функцию cmp. Эти ограничения определены и представлены в виде теории Compare (Разд. 4.2).

7. Обзор

Программа бинарного поиска является весьма популярной. Довольно часто она используется для иллюстрации новых методов верификации.

В работе [18] методом дедуктивной верификации оценивается объем используемой памяти, в частности, размер стека, для программ на языке Си. В качестве тестовой использовалась преобразованная в рекурсивную программа быстрого поиска bsearch с логарифмическими оценками.

Программа bsearch использована для демонстрации аннотаций, вставляемых для оценки наихудшего времени исполнения, в руководстве к пользованию сертифицированным оптимизирующим компилятором CompSert [21], корректность которого доказана в системе верификации Coq.

В инструменте верификации Frama C [1] применяется язык спецификаций ACSL [17] программ на языке Си. На языке ACSL записываются спецификации программ в виде контрактов. Контракт вида behavior определяет вариант спецификации. Полная спецификация слагается из набора вариантов. Такой способ спецификации аналогичен спецификации предусловия и постусловия для одной ветви гиперфункции. Метод спецификации иллюстрируется на программе bsearch.

С применением технологии программирования в ограничениях в работе [22] предлагается новый метод автоматического построения инвариантов с проведением доказательства их корректности. Программа bsearch использована в качестве тестовой.

В работе [19] сообщается об обнаружении ошибки балансировки в программе поиска в бинарном дереве. При этом программа оставалась корректной, но более медленной. Например, если бы в нашей программе мы использовали бы вызов

```
bse(p+1, n - 1, b[p+1..p+n-1] #1: p' : #2)
```

вместо вызова

```
bse(p+m+1, n - m - 1, b[p+m+1..p+n-1] #1: p' : #2).
```

Тогда вместо логарифмической оценки сложности на определенных входных данных была бы линейная оценка. В работе [20] разработан метод дедуктивной верификации, гарантирующий не только функциональную корректность, но и определяющий наихудшую оценку сложности. Метод иллюстрируется на программе bsearch.

8. Заключение

В настоящей работе представлена дедуктивная верификация программы бинарного поиска bsearch из библиотеки ядра ОС Linux. Программа максимально универсальна. Размер элемента и операция сравнения элементов задаются параметрами программы bsearch. Обработку объектов произвольного типа можно запрограммировать в языке Си только через указатели общего вида **void** *. Из соображений эффективности используются битовые операции и адресная арифметика. Перечисленные особенности определяют повышенную сложность дедуктивной верификации программы bsearch.

Библиотечная программа bsearch получена из предикатной программы (разд. 4) применением набора оптимизирующих трансформаций. В действительности, в результате трансформаций получена другая программа, близкая к исходной библиотечной. Их эквивалентность легко проверяется. Следует отметить, что полного совпадения исходной и трансформированной программы не было также и для программы конкатенации строк strcat [10].

Дедуктивная верификация предикатной программы bsearch должна бы быть существенно проще и быстрее в сравнении с дедуктивной верификацией исходной императивной программы. Однако верификация предикатной программы bsearch осложнилась из-за неучтенных особенностей системы автоматического доказательства Why3 [25]. Завершение дедуктивной верификации проводилось в системе PVS [24]. Система Why3 предполагает

иной стиль спецификации программ по сравнению со спецификацией для системы PVS. Этот стиль предстоит освоить для успешной верификации других программ.

Дедуктивная корректность подтвердила корректность программы bsearch. Сопутствующей целью верификации было определение точных ограничений на операцию сравнения элементов. Эти ограничения представлены в виде теории Compare (Разд. 4.2), определяющей симметричность, транзитивность и тотальность операции сравнения. Тотальность – это возможность применения операции к любым аргументам. В пользовательской документации к программе bsearch данной информации нет.

Чтобы обеспечить высокий уровень доверия к инструментам верификации, необходимо гарантировать корректность применяемой технологии и сопутствующих инструментов. С этой целью на базе модели внутреннего представления транслируемой программы с языка P предполагается построить модель оптимизирующих трансформаций и провести ее верификацию.

Имеется расширенная версия данной статьи [26] с приложениями, где подробно документируется процесс дедуктивной верификации.

Список литературы

1. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. [Электронный ресурс]. URL: http://linuxtesting.org/astraver, 15.10.2017 (дата обращения 30.11.2018).
2. Ефремов Д.В, Мандрыкин М.У. Формальная верификация библиотечных функций ядра Linux. Труды ИСП РАН, том 29, вып. 6, 2017. С. 49-76. DOI: 10.15514/ISPRAS-2017-29(6)-3
3. Доказательство правил корректности операторов предикатной программы. [Электронный ресурс]. URL: http://www.iis.nsk.su/persons/vshel/files/rules.zip (дата обращения 12.11.2018)
4. Каблуков И.В., Шелехов В.И. Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика, № 11. Новосибирск, 2017. С. 21-48. Электрон. журн. 2018. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения 12.11.2018)
5. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования P. Версия 0.14. Новосибирск, 2018. 45с., [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения 12.03.2019).

6. Тумуров Э.Г., Шелехов В.И. Технология автоматного программирования на примере программы управления лифтом // «Программная инженерия», Том 8, № 3, 2017. – С.99-111. http://persons.iis.nsk.su/files/persons/pages/lift1.pdf
7. Чушкин М.С. Система дедуктивной верификации предикатных программ // «Программная инженерия». 2016. № 5. С. 202-210. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/paper.pdf (дата обращения 12.11.2018).
8. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21. https://www.iis.nsk.su/files/preprints/154.pdf
9. Шелехов В.И., Чушкин М.С. Верификация программы быстрой сортировки с двумя опорными элементами // Научный сервис в сети Интернет. М.: ИПМ им. М.В.Келдыша, 2018. 26с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf (дата обращения 12.11.2018).
10. Шелехов В.И. Дедуктивная верификация и оптимизация предикатной программы конкатенации строк // Системная информатика, № 12. Новосибирск, 2018. С. 61-84. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/strcat.pdf (дата обращения 12.11.2018).
11. Шелехов В.И. Классификация программ, ориентированная на технологию программирования // «Программная инженерия», Том 7, № 12, 2016. С. 531–538. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prog.pdf (дата обращения 12.11.2018).
12. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препр. / ИСИ СО РАН. № 164). [Электронный ресурс]. URL: https://www.iis.nsk.su/files/preprints/164.pdf (дата обращения 12.11.2018)
13. Шелехов В.И. Разработка программы построения дерева суффиксов в технологии предикатного программирования. Новосибирск, 2004. 52с. (Препр. / ИСИ СО РАН; N 115).
14. Шелехов В.И. Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13с. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения 12.11.2018).
15. Шелехов В.И. Синтез операторов предикатной программы // Труды конф. «Языки программирования и компиляторы ’2017», Ростов-на-Дону. 2017. С.258-262.

[Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/sintr.pdf (дата обращения 12.11.2018).

16. Шелехов В.И. Правила доказательства корректности предикатных программ. — Новосибирск, ИСИ СО РАН, 2019. [Электронный ресурс]. URL: http://persons.iis.nsk.su/files/persons/pages/prrules.pdf (дата обращения 12.06.2019).
17. Baudin P., Filliatre J. C., Cuoq P., March C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language, 2015. [Электронный ресурс]. URL: http://frama-c.com/download.html. (дата обращения 15.05.2019)
18. Carbonneaux Q., Hoffmann J., Ramananandro T. , Shao Z. End-to-end verification of stack-space bounds for C programs // PLDI '14, 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014. P. 270-281.
19. Filliatre J.C., Letouzey P. Functors for proofs and programs. // European Symposium on Programming (ESOP). LNCS 2986, 2004, P. 370–384.
20. Guéneau A., Charguéraud A., Pottier F. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification // Programming Languages and Systems. ESOP. LNCS 10801. 2018. P. 533-560.
21. Leroy X. The CompCert C verified compiler: Documentation and user's manual. 2014. 60p.
22. Ponsini O., Collavizza H., Fedele C., Michel C., Rueher M. Automatic Verification of Loop Invariants // 2010 IEEE International Conference on Software Maintenance, 2010, P.2-11.
23. Shelekhov V. I. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. 2011. Vol. 45, No. 7, P. 421–427.
24. PVS Specification and Verification System. SRI International. [Электронный ресурс]. URL: http://pvs.csl.sri.com/ (дата обращения 12.11.2018).
25. Why 3. Where Programs Meet Provers. [Электронный ресурс]. URL: http://why3.lri.fr, (дата обращения 15.05.2019)
26. Шелехов В.И. Верификация предикатной программы бинарного поиска объекта произвольного типа. Новосибирск, 2019. 26с. http://persons.iis.nsk.su/files/persons/pages/fsearch2.pdf

УДК 81'322.2

Анализ тематических кластеров текстовых коллекций и исследование временно`й динамики тем (на материале конференций по Argument Mining)

Пименов И.С. (Новосибирский государственный университет),
Саломатина Н.В. (Институт математики им. С.Л. Соболева),
Сидорова Е.А. (Институт систем информатики им. А.П. Ершова)

В статье представлены результаты исследования изменений, происходящих в тематических кластерах, построенных на коллекции текстов конференций предметной области Argument mining. Выявление терминов, установление связей между ними и тематическая кластеризация проведены с помощью сторонних программных средств, позволяющей извлекать термины в форме именных словосочетаний, проводить их кластеризацию на базе алгоритма, основанного на применении функции модулярности. Приводится оценка качества полученных кластеров по трем критериям. Трансформацию терминологического состава кластеров во времени предлагается анализировать с помощью ориентированных графов, построенных на основе критерия, который позволяет фиксировать наиболее важные изменения. Терминологическая лексика выявленных тематических кластеров характеризует отдельные направления, в которых ведутся исследования, а трансформация терминологического состава кластеров во времени демонстрирует смещение интересов.

Ключевые слова: *тематическая кластеризация, коллекции текстов предметной области, динамика тем во времени, Argument Mining*

1. Введение

Исследование временно`й динамики тем различных предметных областей (ПО) представляет интерес не только для решения масштабных задач: отслеживания эволюции ПО, построения прогнозов дальнейшего развития, но также и утилитарных, таких как знакомство с ПО, обновление профессиональных знаний и пр. Данная работа направлена на создание инструмента для поддержки исследований такого типа, например, в помощь при построении реферативных обзоров для того, чтобы выявить актуальные направления ПО, применяемые в них методы. Автоматическая обработка коллекций текстов дает возможность

увеличить объем анализируемой литературы и позволяет ослабить субъективное влияние экспертов ПО.

Коллекции текстов, как правило, формируются из статей, полученных по запросам к базам данных сетей цитирования и\или из рейтинговых публикаций в тематических журналах. Особенность данного исследования заключается, в частности, в том, что оно проводится на материале трудов конференций. Для них, в отличие от журнальных статей, как правило, рейтинг неизвестен. Но статус конференции и сам факт прохождения отбора говорит о значимости анализируемых текстов. Следует признать, что риск ошибок в оценивании тематических изменений ПО по материалам конференций выше, чем по рейтинговым статьям. Тем не менее, такое исследование представляет интерес, поскольку позволяет оперативно получать представление о смещении научных интересов на самом раннем этапе.

Структура ПО, задаваемая тематическими кластерами, может быть построена путем применения к текстовым коллекциям любых известных методов кластеризации (иерархических, k-средних, плотностных, графовых и пр., см., например, [1]), используемых для анализа нетекстовых типов данных.

Подходы к кластеризации можно разделить на кластеризацию текстов по *совместной встречаемости* в них *терминов* (co-word analysis), например, [2] и по *совместному цитированию* (co-citation analysis), в частности, упоминанию в одной публикации пары двух других или, наоборот, цитированию одной и той же публикации другими работами [3, 4]. Как показано в работе [5], результаты структурирования ПО, полученные с применением этих двух подходов к одним и тем же данным, близки.

Исследование динамики изменений, происходящих в тематических кластерах с течением времени, проводится, как правило, с установлением всех возможных изменений в составе терминов. К примеру, в работе [6] применяется метод скользящего окна, структура кластеров фиксируется в перекрывающихся временных интервалах, что позволяет максимально подробно фиксировать зависимость состава кластеров от большого числа параметров. В других работах, таких как [5], применяется такой показатель как индекс включения, не дифференцирующий типы происходящих изменений. В данной работе предлагается критерий для отслеживания только существенных изменений в развитии тематической структуры ПО.

Цели работы: 1) разработать pipeline, позволяющий проводить анализ динамики тематических кластеров во времени; 2) провести апробацию на коллекции текстов ПО.

Результаты, представленные в работе, являются предварительными, поскольку они будут уточняться на коллекции, пополненной текстами исследуемой ПО данного и других временных периодов.

Исследование выполнено при поддержке РФФИ в рамках проектов № 18-00-01376 (18-00-00889) и № 18-00-01376 (18-00-00760).

2. Методы

Методика проводимого исследования включает следующие шаги:

1. Сбор коллекции.
2. Предобработка коллекции (токенизация, морфологический анализ).
3. Извлечение терминов.
4. Структурирование ПО (тематическая кластеризация текстов коллекции).
5. Оценка качества построенных кластеров.
6. Построение графов трансформации тем.

Для относительно новых ПО сбор коллекций является непростым делом, большая часть актуальных текстов часто оказывается вне зоны свободного доступа. Основными требованиями к коллекции текстов для анализа тематической трансформации во времени является примерно равное количество текстов для каждого рассматриваемого периода и жанровая однородность (статья, доклад, тезисы, …).

В качестве инструмента для решения задач, указанных в пп. 2, 3 и 4, для отдельного временного среза применяются методы, которые реализованы в свободно распространяемой программе VOSviewer [7]. Они позволяют проводить кластеризацию терминов как по совместному цитированию, так и по совместной встречаемости терминов (co-occurrence links) в полных текстах. В результате реализуется четкая классификация по терминам ПО и нечеткая по текстам.

Изменения кластеров во времени отражаются в двудольном графе, построенном на основе критерия, позволяющего отслеживать трансформацию тем избирательно.

2.1. Методы, реализованные в программе VOSviewer

Извлечение терминоподобных словосочетаний (в дальнейшем для краткости «терминов») производится по шаблону, определяемому как последовательность существительных и прилагательных, оканчивающаяся на существительное.

Структура ПО формируется в виде сети путем связывания терминов согласно вычисленной силе связи (s_{ij}) между терминами i и j, определяемой мерой ассоциации:

$s_{ij} = 2mc_{ij}/c_ic_j$, где c_{ij} – число текстов, в которых термины i и j встречаются совместно, а c_i, c_j – число текстов, в которых встречается i-ый и j-ый термин соответственно, m – общее число связей в сети [8].

Унифицированный (единый) подход к кластеризации и визуализации, реализуемый программой VOSviewer, опирается на нахождение экстремума функции модулярности, в которой имеется параметр (r), позволяющий кластеризацию сети с необходимой степенью детализации. Термины, объединенные в кластеры, характеризуют отдельные темы ПО. Программа дает возможность отфильтровывать служебную и общеупотребительную лексику по предлагаемому пользователем Стоп-словарю, объединять кластеры с малым количеством элементов с ближайшими более крупными.

2.2. Методы оценки качества кластеров

Известно, что проблема проверки адекватности кластеризации не решена в теоретическом плане. Без априорного знания принадлежности объектов к построенным кластерам задача оценки качества чаще всего решается вручную. Формальные критерии проверки сводятся, в основном, к проверке таких свойств как компактность, концентрация и отделимость [9]. Построенные критерии обычно объединяют проверку свойств компактности и отделимости, проверка концентрации элементов кластера вокруг его центра представлена чаще неявно.

В данном исследовании оценка качества кластеров позволяет подобрать наилучшим образом параметры программы VOSviewer для проведения кластеризации. Были использованы три критерия, по которым производилось оценивание компактности и отделимости строящихся кластеров:

1. Validity index (*CS*) [10] измеряет отношение максимального суммарного расстояния между элементами одного кластера к минимальному суммарному расстоянию между центрами кластеров. Чем меньше значение критерия, тем более качественным считается разбиение на кластеры.

2. Calinski-Harabasz index (*VCR*) [11] вычисляет нормированное отношение суммарного внутрикластерного расстояния к суммарному расстоянию в кластеризуемом множестве. Для "идеальной" кластеризации оно равно 1, т.е. чем ближе значение индекса к 1, тем лучше разбиение множества на кластеры.

3. Silhouette индекс (*SWC*) [12] фактически является мерой несхожести элементов одного кластера с элементами других кластеров и гарантирует лучшее разбиение при высоком значении *SWC*.

2.3. Построение графов трансформации тем

В данной работе для отслеживания временных изменений в кластере используются ориентированные графы, отражающие изменение терминологического состава кластера при переходе из отсчета времени t в t+1.

Пусть $C_t = \{c_i^t\}$ – множество кластеров в коллекции текстов, относящихся к периоду времени t, i = 1, …, n_{cit}, n_{cit} – число терминов в кластере c_i^t.

Пусть при переходе от временного среза t к t+1 каждый кластер c_i^t трансформируется в упорядоченное (по убыванию числа содержащихся в нем терминов отсчета t) множество $\{c_j^{t+1}\}$, j = 1,…, K_j, K_j – число вновь образовавшихся кластеров. В анализе трансформации кластера c_i^t при переходе к отсчету t+1 учитываются четыре типа преобразований, отвечающие следующим условиям:

1) кластер c_i^t трансформировался большей частью в c_1^{t+1}, так что $\Delta_{1,2} > p$ (p – порог, регулирующий объем пересечения кластеров);

2) кластер c_i^t трансформировался большей частью в c_1^{t+1} и c_2^{t+1}, так что $\Delta_{1,2} \le p$ и $\Delta_{1,3} > p$.

3) число кластеров, в которые трансформировался большей частью кластер c_i^t, превышает два (j > 2 и $\Delta_{1,j} > p$);

4) кластер c_i^t отсутствует во множестве $\{c_j^{t+1}\}$ большей частью своих элементов: $\Delta_{0,1} > p$ ($c_0^t = c_i^t \setminus c_j^{t+1}$), где $\Delta_{1,n} = |c_1^{t+1} \cap c_i^t| / |c_i^t| - |c_n^{t+1} \cap c_i^t| / |c_i^t|$, (n > 1).

Тип 1) можно интерпретировать как сохранение темы, возможно, с обновлением, 2) – как выделение самостоятельной темы. Типы 3), 4) – это два типа прекращения существования темы (поглощение другими кластерами, практически полное исчезновение). Выполнение условий обеспечивает отбор кластеров, играющих существенную роль в трансформации.

3. Эксперимент

Обработка коллекции текстов ПО Argument Mining (токенизация, морфологический анализ, извлечение терминов, кластеризация) для данных каждого временного отсчета (2015, 2016, 2017 гг.) выполнена программой VOSviewer. Большая часть служебных и общеупотребительных слов отфильтрована по собранному авторами Стоп-словарю, содержащему более 500 слов и словосочетаний. Оценка качества построенных кластеров позволяет считать проведенную кластеризацию удовлетворительной. Изменение терминологического состава кластеров зафиксировано в графе развития тем, что дает возможность выявлять исследовательские приоритеты для данной ПО и каждого временного среза.

3.1. Используемые данные

В анализируемую коллекцию всего рассматриваемого периода (2015 – 2017 гг.) вошел 51 доклад, сделанный на конференциях WorkShop 2015, 2016, 2017: 16 докладов, 20 докладов и 15 докладов соответственно, общим объемом около 184 тыс. словоупотреблений. Из текстов докладов были удалены ссылки на литературу, параграфы, посвященные обзору литературы.

3.2. Результаты эксперимента

Выявленные программой VOSviewer термины представляют собой, в основном, одно- и двух (реже трех-) словные именные группы с текстовой частотой $f_t > 2$. В первом периоде таких сочетаний оказалось 209, во втором – 308 и 216 – в третьем, всего за весь период 2015 – 2017 рассмотрено 443 разных термина. В результате кластеризации терминов в отдельные периоды получены 4, 5 и 3 непересекающихся кластера соответственно, объемом от 25 (2015 г.) до 97 (2016 г.) терминов.

3.2.1. Содержание тематических кластеров

Терминологический состав каждого кластера характеризует подход к исследованиям в целом: параметры используемых текстовых ресурсов (темы и объемы данных), применяемые методы (включая информацию об анализируемых текстовых фрагментах), распознаваемые объекты (см. таблицу 1). Кластеры в таблице пронумерованы и упорядочены по убыванию объема содержащихся в них терминов.

Таблица 1. Термины, характеризующие кластеры докладов 2015, 2016, 2017 гг.

временной отсчет	№ кластера	термины		
		данные	методы	объекты распознавания
2015 г.	1	opinion, corpus, corpora, wordnet, …	dataset, training set, annotation process, expert, indicator, semantic similarity, latent dirichlet allocation, machine learning, topic model, unigram, tree, …	claim, major claim, argument (-ative) structure, argument component, premise, attack relation, …
	2	collection, document, ...	sentence, phrase, token, word, verb, noun, lemma, vector, weight, model, graph, rule,	negative sentiment, argument, original claim, ...

			pattern, frequency, cosine similarity, score, measure, distance, cluster, ...	
	3	debate, politic, large set, ...	classifier, training data, test set, classification, recall, precision, fold cross validation, n-gram, adjective, adverb, boundary token, iteration, probability distribution, ...	...
	4	article, post	annotation, inter annotator agreement, context, label, argumentation mining, baseline, experimental setup, feature vector, modal verb, ...	argumentation relation, support relation, negation, ...
2016 г.	1	opinion, gay right, debate, online debate, collection, document, article, post, ...	word, phrase, sentence, unigram, model, topic, feature set, sentiment feature, verb, classifier, evaluation, test set, baseline model, context, stance classification, tree, graph,...	sentiment, negative sentiment, ...
	2	persuasive essay, alcohol, ...	annotation process, expert, inter annotator agreement, annotator, expert annotator, annotation guideline, annotation procedure, consensus, proposition, statement, clause, training, occurrence, indicator, rule, distribution, confusion matrix, elementary discourse unit, linguistic feature, ...	argumentation scheme, attack, major claim, support relation, inference, ...

	3	corpus, forum, argumentative microtext, ...	link, node, pattern, frequency, training data, classification, testing, false negative, false positive, recall, precision, f1 score, manual analysis, statistic, cohens kappa, controversial topic, parser, location, segmentation, tree,... structure	Argument (-ative, -ation) structure, argument component, premise, attack relation, negation, rebuttal, ...
	4	data set, obama, marijuana, ...	experimental setup, human annotator, annotation task, token, semantic similarity, score, measure, individual feature, random forest, ...	claim, main claim, ...
	5		title, whole sentence, training set, bigram, n-gram, fold cross validation, svm, support vector machine, binary feature, cross validation, ...	
2017 г.	1	opinion, debate, corpus, report, web, ...	proposition, sentence, noun, human annotator, inter annotator agreement, manual analysis, rule, pattern, distribution, cohens kappa, lda, cosine similarity, distance, character, metric, ranking, semantic, supervised machine, validity, recall, f1 score,...	attack, argument structure, negation,
	2	dataset, collection, corpora, Wikipedia, ...	natural language processing, model, topic, token, word, unigram, classifier, training data, test set, context, stance	claim, sentiment, ...

			classification, prediction, pipeline, baseline, precision, score, svm, probability, logistic regression, tf idf, word embedding, graph, ...	
	3	document, article, argumentative text, persuasive essay, ...	annotator, annotation scheme, expert classification, genre, labeling, clause, feature set, occurrence, confusion matrix, statistic, measure, criterium, error analysis, ...	argument scheme, argument component, argumentative structure, argumentative unit, argumentative relation, major claim, premise, ...

Следует заметить, что в таблице есть ячейки, которые слабо заполнены или не заполнены вовсе. Например, "объекты распознавания" отсутствуют в кластере 3 в 2015 г. и в кластере 5 в 2016 г. Это означает, что термины, относящиеся к этим аспектам текста, встречаются менее, чем в 3-х текстах коллекции и не прошли фильтрацию по текстовой частоте. В кластере 3 это, например, энтимемы, в кластере 5 – противоречивые темы, предложения для аргументации о значении терминов, свидетельства отсутствия оппозиции в эссе.

3.2.2. Оценка качества кластеров

Полученные количественные значения критериев оценки качества кластеров (см. п. 2.2), представленные в табл. 2, показали, что наилучшее разделение терминов на кластеры достигается при значении параметра r (детализация сети терминов) равном 1. При выбранном значении резолюции кластеры с малым числом элементов в нашем случае не образуются и параметр, позволяющий производить объединение мелких кластеров с ближайшими бо`льшими, на результат кластеризации влияния не оказывает.

Таблица 2. Количественные оценки качества точной кластеризации в каждый из временных отсчетов по критериям *VCR*, *CS*, *CWS*

год	резолюция	Число класт.	*VCR*	*CS*	*CWS*
2015	1,00	4	30,708	0,333	0,139
	1,05	5	26,275	1,223	0,040

	1,10	11	12,329	8,204	-0,165
2016	1,00	5	40,109	0,391	0,087
	1,05	6	30,716	0,430	-0,023
	1,10	12	18,464	0,787	-0,090
2017	1,00	3	57,267	0,076	0,313
	1,05	4	40,505	0,597	0,183
	1,10	8	15,761	7,584	-0,007

Оценка качества кластеров при изменении параметра r, регулирующего число кластеров, показала, что при росте r качество кластеризации падает, хотя при r = 1 и r = 1,05 увеличение числа кластеров на единицу не влечет большого изменения состава кластеров, а вызывает лишь отделение одного небольшого по объему кластера. Но уже для r = 1,10 стабильность кластеризации нарушается.

3.2.3. Динамика развития тем

Для построения графа развития тем применяемый критерий (см. п. 2.3) был скорректирован в силу особенностей данных: небольшого объема коллекции и фильтрации терминов по текстовой частоте. Порог p повышен до значения 0,16. В знаменателе критерия учитывалась мощность множества $\underline{c}_i^t = c_i^t \setminus L_i^t$, где L_i^t – множество терминов, отсутствующих во множестве терминов во временном срезе t+1 (вышедших из оборота). Результирующий граф представлен на Рис. 1. Узлы, обозначенные символом N, содержат термины, не встречавшиеся в текстах докладов предыдущего временного среза. Толщина линий пропорциональна объему терминов, общих для смежных временных срезов.

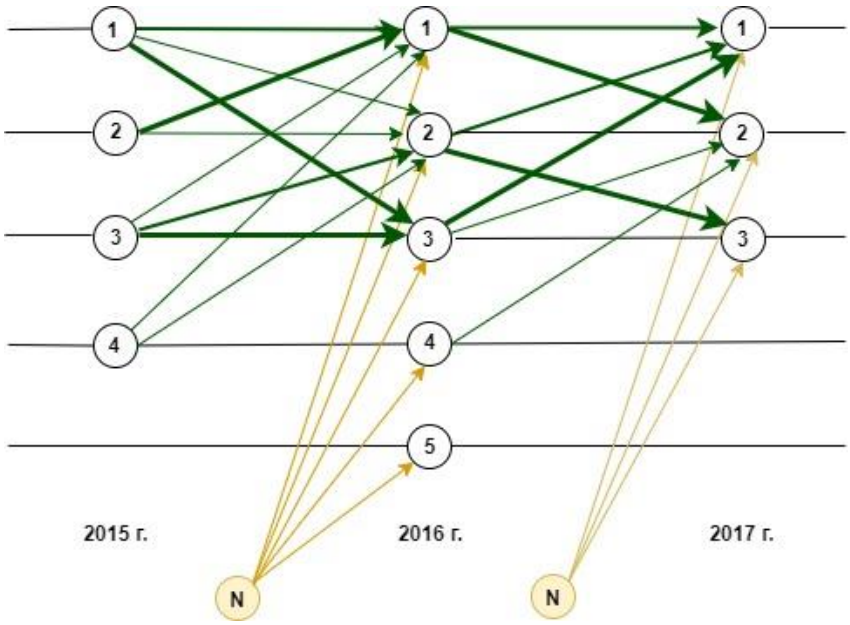


Рис.1. Граф развития тем за период 2015 – 2017.

Причинами нестабильности поведения кластеров, в частности, являются неустоявшаяся терминология ПО, отличающаяся наличием вариантов написания терминов (argument (-ative, -ation) structure; svm, support vector machine и др.), особенности жанра (доклады конференций, где смена участников, а следовательно, и обновление тем, неизбежны). Но, тем не менее, проследить наследование терминов одних кластеров другими, относящимися к следующему временному срезу, можно и на данных такого типа. Например, выявление аргументации, имеющей отношение к анализу мнений, графовый метод ее представления просматривается в кластерах 2 (2015), 1(2016), 2(2017), при этом можно заметить появление в методах 2016 г. позиционной контекстной классификации (context, stance classification).

Среди информативных вышедших из оборота терминов можно назвать следующие: wordnet, news article, large set, politic, original claim, trigram, adjective, adverb, main verb, class distribution, probability distribution (отсутствуют в 2016 г.); tree, baseline model, annotation guideline, discourse, elementary discourse unit, linguistic feature, controversial topic, semantic similarity, random forest, n-gram, fold cross validation, binary feature (отсутствуют в 2017 г.). Нельзя сказать, что вышедшие термины оказались менее актуальными, но они стали реже обсуждаться в данный момент времени.

Значительное обновление терминологии происходило в каждом временном периоде. В 2016 г. это: stance classification, baseline model, sentiment feature, n-gram, confusion matrix, persuasive essay, annotation guideline, elementary discourse unit, linguistic feature, fold cross validation, cohens kappa, f1 score, statistic, argumentation structure, argumentative microtext, controversial topic, parser, rebuttal, tree structure, main claim, svm, support vector machine;

2017 г.: ranking, supervised machine, web, logistic regression, pipeline, tf idf, wikipedia, word embedding, argumentative relation, argument scheme, argument unit.

На графе легко увидеть возникшие и исчезнувшие кластеры, содержащие практически новую лексику: 4-й и 5-й в 2016 г., один из которых в следующем году прекращает свое существование (5-й).

5. Заключение

Результатом проделанной работы является создание pipeline для проведения исследований по выявлению тематических кластеров ПО на основе коллекций текстов, а также отслеживанию изменений в их терминологическом составе в отдельные временные периоды.

Трансформацию терминологического состава кластеров предлагается анализировать с помощью ориентированных графов, построенных на основе критерия, который позволяет фиксировать наиболее значимые изменения. Терминологическая лексика выявленных тематических кластеров характеризует отдельные направления, в которых ведутся исследования, а трансформация терминологического состава кластеров во времени демонстрирует изменения, связанные с предпочтениями в выборе задач и методов.

Так, анализ предметной области Argument Mining выявил наличие неустоявшейся терминологии, стабильность в использовании методов на основе машинного обучения, а также конкурентоспособных методов на основе знаний. Отмечен переход от более простых моделей аргументации к более сложным. Данное исследование полезно при составлении обзоров ПО, выявления baseline, выборе актуальных методов и ресурсов для решения задач, связанных с анализом аргументации.

Планируется продолжить работы и провести расширенный эксперимент на коллекции текстов ПО Argument Mining, дополненной текстами других временных срезов.

Список литературы

1. Пархоменко П.А., Григорьев А.А., Астраханцев Н.А. Обзор и экспериментальное сравнение методов кластеризации текстов // Труды ИСП РАН, 2017. Т. 29, вып. 2. С. 161-200.
2. Callon M., Courtial, J.P. Laville. Co-word analysis as a tool for describing the network of interaction between basic and technological research: the case of polymer chemistry // Scientometrics. 1991. N 22. P. 155−205.
3. Small H. Tracking and predicting growth areas in science [Электронный ресурс]. URL: http://www.scimaps.org/exhibit/docs/small.pdf (дата обращения: 10.09.2019).
4. Van Eck N.J., Waltman L. Visualizing Bibliometric Networks [Электронный ресурс]. URL: https://link.springer.com/chapter/10.1007 (дата обращения: 10.09.2019).

5. Cobo M.J., López-Herrera A.G., Herrera-Viedma E., Herrera F. An approach for detecting, quantifying, and visualizing the evolution of a research field: A practical application to the Fuzzy Sets Theory field [Электронный ресурс]. URL: https://www.sciencedirect.com/science/article/pii/S1751157710000891 (дата обращения: 10.09.2019).
6. Kandilas V., Uphum, S. P., Ungar L. H. Analyzing knowledge communities using foreground and background clusters [Электронный ресурс]. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.3141&rep= rep1&type=pdf (дата обращения: 12.09.2019).
7. VOSviewer Homepage, URL: http://www.vosviewer.com/, (дата обращения: 12.09.2019).
8. Waltman, L., Van Eck, N.J., & Noyons, E.C.M. A unified approach to mapping and clustering of bibliometric networks // Journal of Informetrics. 2010. N 4(4). P. 629-635.
9. Сивоголовко Е.В. Методы оценки качества четкой кластеризации // Компьютерные инструменты в образовании. 2011. № 4. С. 14–31.
10. Chou C.H., Su M.C., E. Lai. A new cluster validity measure and its application to image compression // Pattern Analysis and Applications. 2004.
11. Calinski R.B., Harabasz J. A dendrite method for cluster analysis // Comm. in Statistics. 1974.
12. Kaufman L., Rousseeuw P. Finding Groups in Data. An Introduction to Cluster Analysis. Wiley, 2005.