Bull. Nov. Comp. Center, Comp. Science, 32 (2011), 1–18

Integrated approach to analysis and verification of imperative programs*

I. S. Anureev

Abstract. This paper describes a new approach to the analysis and verification of imperative programs, which allows us to integrate, unify and combine the methods and techniques of analysis and verification of imperative programs, accumulate and use knowledge about them. A feature of the approach is to use the domain-specific language Atoment, designed to develop tools for analysis and verification of programs, which allows us to represent both methods and techniques of analysis and verification and data for them (program models, annotations, inference rules, etc.) in a single unified format. The paper includes an introduction to the language Atoment, a description of a multilanguage system Spectrum of analysis and verification of programs, based on this language, and a methodology of applying this approach to verification of imperative programs by example of the C-light language.

Keywords: *program analysis, program verification, program model, domain-specific language, verification system, C-light, C-kernel.*

1. Introduction

Imperative programming languages such as C, C++, Java, C# play an important role in the software industry. Therefore, analysis and verification of programs in these languages is an actual problem in the theory and practice of programming. Let us consider the most interesting projects in this area.

In the project of Sharma, Dhodapkar, Ramesh et al. (Bhabba Atomic Research Centre, Indian Institute of Technology) [1], a deductive method for detecting runtime errors in programs written in the industrially sponsored safe subset of C called MISRA C was presented. The method is based on a special model of C programs: each C program is modeled as a typed transition system encoded in the specification language accepted by the PVS theorem prover. Since the specification is strongly typed, proof obligations are generated, for possible type violations in each statement in C, when loaded in the PVS theorem prover [2] which need to be discharged. The technique does not require execution of the program to be analysed and is capable of detecting runtime errors such as array bound errors, division by zero, arithmetic overflows and underflows, etc. Based upon the method, a tool was developed which converts MISRA C programs into PVS specifications automatically. The tool was used in checking runtime errors in several

*Partially supported by the RFBR under grant 11-01-00028-a and Integration Project RAN 14/12.

programs developed for real-time control applications. However, despite the fact that the language of MISRA C is extremely limited (for example, unions, pointer arithmetic, dynamic memory and recursion are forbidden), it has no formal syntax definition. Therefore, input programs should be examined for compliance with this standard. Also, the method is not a deductive verification in a usual sense and is intended only for a search for some classes (see above) of run-time errors.

The Verisoft project (University of Saarlandes, German Research Center for Artificial Intelligence) [3] is an example of ad-hoc verification project oriented mainly to embedded systems. Verification of the operating system kernel for a simple but annotated processor is one of the goals of the project. The C0 language, a simple subset of C, is used. Its semantics is modeled in the theorem-proving system Isabelle/HOL [4]. Because of a weak expressive power of the C0 language, the Assembler language is used in addition, which complicates the verification. However, the verified libraries of string and list processing were written in C0.

A promising approach to verification of C-programs was proposed in the project Why (France, INRIA) [5]. Why is a platform suitable for verification of many imperative languages. It defines an intermediate language of the same name to which programs of target programming languages are translated. The purpose of translation is generation of verification conditions in a form that does not depend on the specific theorem-proving system. The toolset Frama-C [6] was built on the basis of Why. It supports a static analysis of the full C language and deductive verification for a limited subset of C (the goto statements can not jump backward and inside blocks; function pointers, casting between integers and pointers, union, functions with variable parameter lists, and computation over real numbers are forbidden.). Also, a subset of the standard library, including the important functions of memory and files, was annotated using the specification language ACSL (ANSI/ISO C Specification Language Home Page) [7]. The list of verified programs includes quite simple sorting and search programs.

In 1997, the University of Nijmegen started the LOOP project (Logic of Object-Oriented Programming) [8]. It is oriented to automated verification of Java programs. Most of Java constructions, except for multi-threading and nested classes, are supported. In fact, LOOP is a compiler written in OCaml. Its input is a sequential Java program and its specification in the JML language (Java Modelling Language) [9, 10]. Its output is a set of files in the syntax of the theorem-proving system PVS which describe the meaning of the program and its specifications. The algebraic approach is used to define the semantics of objects and classes. The system was successfully applied to verification of programs in the JavaCard language which is used in the so-called smart cards. As a drawback, we note that the system works effectively only for small programs.

The project ESC/Java (Extended static checking for Java) [11, 12] is another example of Java program verification. It supports a broad subset of Java. The key idea of the project is that the system ESC/Java is aimed at finding common errors in programs rather than at proving their full functional correctness. This increases the power of automated proving at the cost of skipping some errors. The system uses a simple language of contracts, which is a subset of JML, as the specification language. The semantics is described by the weakest precondition calculus. Although this approach is criticized for its incompleteness and inconsistency, the system is recognized as impressive in deductive program verification. The system has also been used successfully for verifying JavaCard programs.

The Spec# programming system [13, 14] is a new attempt at a more cost-effective way to develop and maintain high-quality software. It fully integrates into the development environment Microsoft Visual Studio and .NET Framework and thus provides the complete infrastructure, including libraries, designing support, editing tools, etc. The system is oriented to the Spec# language, which is an extension of the object-oriented language C#. Spec# extends the type system to include non-null types and checked exceptions. It provides contracts in the form of pre- and postconditions, as well as object invariants. Specifications become a part of the program execution and are checked dynamically. A special component of the Spec# programming system, the Spec# static program verifier Boogie [15], allows static checking of specifications. It generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyzes the verification conditions to prove correctness of the program or find errors in it.

Analysis of the projects reviewed allows us to indicate a number of inherent disadvantages. First, their theories are often developed for a specific rather limited class of problems or for a single programming language. On the other hand, for a general-purpose system (eg. Why) quite simple examples of analysis and verification are considered, which demonstrates the limited power of the verification methods used in these systems. Second, in some projects little attention is paid to justification of the methods correctness. Finally, the "closed" approach to development dominates in the absolute majority of projects. The closeness in this context means that the methods developed and implemented by participants (experts and programmers) of a project are black boxes for users of a corresponding verification system. Users have to wait for new versions of the system or adding new methods in the current version. At the same time, the complexity of the approaches used and the lack of documentation prevents analysis and modification of the system by ordinary users even if the source code of the system is accessible.

We propose a new approach to verification of imperative programs which

unifies and integrates the methods and techniques of verification of imperative programs and allows us to overcome these shortcomings. The key idea of the approach is the use of unified models to describe programs (as well as annotations, logical formulas, deduction rules, etc.) and a domain-specific language (which is natural to both mathematicians and programmers) to describe the methods and techniques of analysis and verification of programs.

The approach uses the executable specification language Atoment [16, 17, 18] as a domain-specific language. It takes into account the specificity of this problem domain, namely:

- representation of data (programs, annotations, axioms, properties, inference rules, etc.) in the form of trees;
- application of analysis and verification methods to software models, which are labeled ordered directed graphs, instead of their application to original annotated program texts;
- a natural representation of many practical methods and techniques of analysis and verification (static analysis methods, methods based on transformational, operational and axiomatic semantics, model-checking techniques, automaton methods; bisimulation techniques) as conversions on these graphs;
- a complex conceptual structure of software systems and programming languages, including hundreds of concepts.

The Atoment language also ensures the fulfilment of a number of methodological principles, which, in our opinion, one needs to follow when developing the methods of analysis and verification.

First, transition from program texts (annotations, axioms, inference rules, etc.) to their models should satisfy the principle of structural identity. This means that each of the lexical and syntactic units of a text should match exactly one unit of the model. Fulfilment of this principle allows us to identify a text with its model, without proving the correctness of translation of the text into its model.

Second, the translation should satisfy the principle of naturalness. This means that the model should maintain the common terminology and notation. Fulfilment of this principle provides a comfortable conceptual environment for development of methods and techniques of analysis and verification.

Third, the language should have a compact syntax and transparent semantics. This allows us to not spend time learning the language itself, and concentrate on developing the analysis and verification methods.

The multilanguage system Spectrum of analysis and verification of programs, based on this language, allows a user:

- to describe techniques of analysis and verification in a natural notation,

- to transfer these techniques from one programming language to another (using model commonality),
- to analyze and verify algorithms in various object domains, adding (if necessary) new languages to represent them,
- to share methods and techniques of analysis and verification with other users and combine them.

Drawing the user into the development of methods for analysis and verification of its specific tasks should increase both the penetration of formal methods into software development and the quality of analysis and verification of programs. Benefits of a universal approach for teaching analysis and verification methods are also obvious.

The system Spectrum can be regarded both as a specialized development environment for analysis and verification tools and as an information system which accumulates experience in this field in the form of knowledge represented by Atoment specifications and provides access to them. In particular, the methods and techniques of analysis and verification of imperative programs are knowledge represented in this information system.

The paper has the following structure. In Section 2, a short review of the Atoment language is given. In Section 3, the conceptual framework of the extensible multilanguage analysis and verification system Spectrum and the specification of its kernel in Atoment are considered. Section 4 describes the methodology of application of our approach by the example of C-light program verification. In conclusion, the results of the paper are summarized and plans for future research are outlined.

2. The Atoment language

This section describes the basic concepts of the Atoment language to the extent necessary for understanding the examples of its use presented in the following sections.

The Atoment language has a compact unified syntax. All (both syntactic and semantic) units of the language are represented by expressions that are built of bricks of two types – atoms and elements (collectively called atoments) – with a single expression constructor `(...)`. Atoms are syntactic bricks which have a syntactic representation. Elements are the semantic bricks that have no syntactic representation. Formally, elements and atoms are two disjoint sets `Elem` and `At` and expressions are the set `Exp` which are defined as follows:

- if `A` is an atoment, then `A` is an expression;
- `()` is an (empty) expression;
- if $\mathtt{A}_1$, ..., $\mathtt{A}_n$ are slots, then $\mathtt{(A}_1 \ \ldots \ \mathtt{A}_n\mathtt{)}$ is a (compound) expression.

Slots are "places" for subexpressions. They define semantics of subexpressions in the context of the expression. Let `B` be an expression. Four types of slots (depending on their role in the expression) are distinguished: value slots and attribute slots which have the same form `B`, type slots of the form `B@`, and property slots of the form `B@@`. A value slot of the expression `A` is called an attribute slot, if it is in an odd position of the expression which is obtained from `A` by removing all type and property slots. The meaning of these roles is explained below.

Expressions consisting only of value slots are called lists. They are similar to the lists of the Lisp language.

The set of atoms `At` is defined as follows:

1. If `A` is a sequence of Unicode symbols and `A` does not contain whitespace symbols, reserved symbols `(`, `)`, `{`, `}`, `[`, `]`, `"`, and the reserved symbol sequence `'''`, then `A` is an atom.
2. If `A` is a sequence of Unicode symbols and `A` does not contain `"` and `'''`, then `"A"` is an atom. It is called a string.
3. If `A` is a sequence of Unicode symbols and `A` does not contain `'''`, then `'''A'''` is an atom. It is called a text and used to insert texts written in other languages.

The atoms of the first type, starting with `@` or terminating with this symbol, are used for special purposes (for example, to represent the type slots and property slots).

Expressions can be interpreted both as data and as computable entities. The first interpretation is used to represent the data (programs, annotations, axioms, properties, inference rules, etc.) for verification methods and the second one to represent verification methods themselves. Before describing the semantics of (computation of) expressions, we introduce the concepts of the state and the variable that define the context of the computation.

A state `s` is a partial function from elements to expressions. Let `St` be the set of all states. An element `E` is defined in the state `s`, if `s(E)` is defined. The expression `s(E)` is called the value (or the content) of the element `E`. If the expression `s(E)` contains an attribute slot `B`, then the expression `B` is called an attribute of `E`. If a value slot `V` follows `B`, then `V` is called a value of the attribute `B` of the element `E` in the state `s`. Let us note that, because the attribute slot `B` can appear in the expression `s(E)` more than once, the attribute `B` can have several values. If the expression `s(E)` contains a type slot `B@`, then the expression `B` is called a type of the element `E` in the state `s`. Similar to attributes, the element `E` can have more than one type, because there also can be a few type slots in the expression `s(E)`. If the expression `s(E)` contains a slot `B@@`, then the expression `B` is called a property of `E` in the state `s`. A slot `U` is called a slot of the element `E` in the state `s`, if `U` is a slot of the expression `s(E)`.

A state has a natural graph interpretation and represents a labeled ordered directed graph. Each element represents a node of the graph, and the slots of the expression, which is the value of this element in the current state, represent labeled ordered arcs. The element is the tail of these arcs, and subexpressions appearing in the slots are the heads of these arcs. The labels of the arcs are the types of corresponding slots. The order of the arcs is defined by the order of the slots of the expression. Thus, states allow us to describe models, and expressions allow us to describe model conversions.

Any element `E` can be regarded as a concept. The content of the concept `E` is the set of elements for which `E@` is a slot. These elements are called instances of the concept of `E`. For example, the element `E` with the value `(goto-statement@ label L)` is an instance of the concept `goto-statement`, which has the attribute `label` with the value `L`, i.e. `E` is a goto statement with the label `L`. The concepts provide the means for categorizing units of target languages (programming languages, annotation languages and so on). Properties also provide the means for categorizing but they do not define new concepts.

Concepts and attributes are used to represent the conceptual (ontological) framework for software systems and programming languages. For a software system they describe its domain ontology, and for a programming language they describe its categorial apparatus.

A variable meaning `vv` is a partial function from atoments to expressions. Let `VV` be the set of all variable meanings. If `vv(A)` is defined, then `A` is called a variable, and `vv(A)` is called the value of the variable `A`. Variables are used to store intermediate results of evaluation of expressions. We extend the function `vv` to expressions in the following way: `vv(U)` is the result of replacing all occurrences of variables defined by the meaning `vv` in the expression `U` with their values; `vv(U, V)` is the result of replacing all occurrences of variables, which are defined by the meaning `vv` and do not belong to the set `V`, in the expression `U` with their values.

Evaluation of the expression `E` returns an expression which is called the value of the expression `E`. Also, evaluation of an expression can change the state. The result of evaluation of an expression depends on the current state and the current variable meaning. Formally, the expression semantics `Sem` is a function from triples `(E, s, vv)`, where `E` is an expression to be evaluated, `s` is the current state, `vv` is the current variable meaning, to the set of triples `(V, s′, vv′)`, where `V` is the value of the expression `E` in the state `s` in the variable meaning `vv`, `s′` and `vv′` are the state and the variable meaning resulting from evaluation of the expression `E`. The semantics of simple expressions is determined by the rules:

- if `A` is an atoment and `vv(A)` is undefined or `A` is an empty expression, then `Sem(A, s, vv) = {(A, s, vv)}`;

- if A is an atoment and vv(A) is defined, then Sem(A, s, vv) = {(vv(A), s, vv)}.

The semantics of compound expressions is defined by semantic schemes. A semantic scheme is an element with the value of the form (sem A var C where B := D). The attribute sem defines the kind of expressions that satisfy the scheme. The expression A, which is the value of this attribute, is called a pattern of the scheme. The attribute var defines the variables of the pattern A, and the attribute where defines restrictions on the values of these variables. Variables are represented by atoments, and their values are represented by expressions. The expression C is either an atoment (representing one variable), or a list of atoments (representing a set of variables).

An expression E is said to match the pattern A or satisfy the scheme if E is obtained from A by replacing all occurrences of variables from C in A with their values subject to the condition that these values satisfy the condition B. In this case, the values of the pattern variables are substituted into the expression D and the resulting expression D′ is evaluated. The result of evaluating the target expression E is the result of evaluating the expression D′. Semantic schemes for the library expressions with the predefined semantics do not contain the body D. The body D can contain an expression (return U). This expression terminates the evaluation of D with the value equal to the value of U. The special case (return) is an abbreviation for (return ()). For example, the expression (one) satisfies the scheme (sem (one) := (return 1)) and the evaluation of this expression returns 1.

The expression (sem A var C where B := D), evaluated in the state s in the variable meaning vv, adds a new semantic scheme with the value (sem vv(A, V) var C where vv(B, V) := vv(D, V)), where V is a set of pattern variables defined by the attribute var.

Denotation. Let us introduce some denotations which we use below. Let A′, B′, A'_2,... denote the results of evaluation of expressions A, B, A_2, ... In the case of several expressions, the expression that is lower in the lexicographic order for letters and the numeric order for indexes is evaluated before. For example, the expression A is evaluated before the expression B, and the expression A_1 is evaluated before the expression A_2. Let s′ denote the state obtained after the evaluation of these expressions.

3. Multilanguage system Spectrum of program analysis and verification

In this section we consider the conceptual framework of the extensible multilanguage analysis and verification system Spectrum and the specification of its kernel in the Atoment language. The conceptual framework of the system is presented in Figure 1.

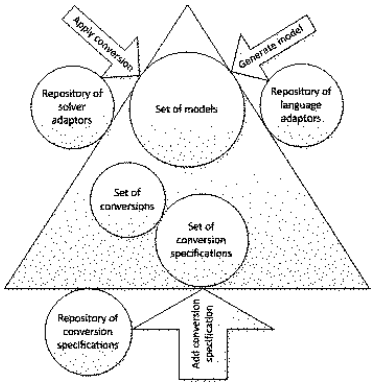

Figure 1. Conceptual framework of the Spectrum system

The logic of the system operation is based on the successive conversions of models, among which are program models, annotation models, models of logical formulas, models of deduction rules, etc.

The main cycle of operation of the system is sending a query to perform a model conversion and obtaining a result of this conversion. The query specifies the applicable conversion and the model to which it applies. The result of conversion of the model includes a model or a set of models (if the model is converted to several models) and additional information. In particular, this information can describe backward dependencies and the conversion status.

Backward dependencies (the correspondence between units of the input model and units of the output models) are used to adjust the result of successive conversions to the initial formulation of the target task of analysis or verification.

The conversion status describes a logical connection between the input model and the output model. For program models it can take the values "preserves correctness", "strengthens correctness", "weakens correctness". The status "preserves correctness" means that the input model is correct if and only if the output model is correct. The status "strengthens correctness" means that if the output model is correct, then the input model is correct. The status "weakens correctness" means that if the input model is correct, then the output model is correct. For formula models it can take the values "preserves validity" or "preserves satisfiability".

Additionally, a user can send queries "to generate a model" and "to add a conversion specification".

The query "to generate a model" specifies the text in a target language (a programming language, an annotation language, a language of inference rules, etc.) which is converted to the model. The unified format of the internal language of the system (the language of models) represented by Atoment

expressions provides a multilanguage character of the system. To generate a model, the system uses a repository of language adapters which translate the text in the target language into its model. The result of the translation is a model and additional information which, in particular, describes the backward dependencies (correspondence between units of the target language and units of the resulting model).

A special case of conversions is application of solvers to check formulas. To translate a formula model into the format of a solver, a repository of solver adapters is used. The result of the conversion in this case is a simplified formula (more properly, a model of the formula), the proof status (with the values "true", "false", "unproven") or a counterexample. The result can also contain information about the process of proving (derivation tree, statistical information).

The query "to add a conversion specification" specifies the conversion specification added. This specification defines a set of conversions that become available after executing the query by the system.

Thus, in the system Spectrum both models and their conversions are specified by Atoment expressions. Another important feature of the system is that the kernel of the system is also an Atoment specification that is executed by the interpreter of the Atoment language. The system kernel does not include the repository of language adapters and the repository of solver adapters.

The expression that launches the system Spectrum has the form `(include "spectrum")`. The library expression `(include A)` successively executes the expressions from the file with the name `A` as if these expressions were directly fed to the Atoment interpreter from the command line. The file `spectrum` describes the set of predefined variables and functions of the system, defined by variables and expressions of the Atoment language, respectively.

Since model conversions are Atoment expressions, a query to perform a model conversion is reduced to the execution of the appropriate expression by the interpreter of the Atoment language.

Conversion specifications are described by semantic schemes. The repository of conversion specifications is a set of files with the extension `cs` (conversion specification). A query to add a conversion specification has the form `(include "name.cs")`, where `name.cs` is the name of the file containing a set of semantic schemes. These schemes define expressions that serve as conversions. For example, the query `(include "C-kernel VC generator.cs")` adds the specification of the verification condition generator for the C-kernel language (see Section 4) to the set of accessible conversions of the system. The single conversion of this specification has the form `(generateVC A)`, where `A` is a list of Hoare triples. This expression returns the list of verification conditions for Hoare triples from the list `A`.

A query to generate a model has the form `(generate-model A lang`

`B)`, where `A` is the name of the file containing a text in an input language, `B` is the name of the input language. For example, the expression `(generate-model "sorting.c" lang c-kernel)` generates a model for the C-kernel program that is in the `sorting.c`, and the expression `(generate-model "safe-property.txt" lang mu)` generates a model for the formula in the language of μ-calculus that is in the file `safe-property.txt`.

In the case of conversion which applies a solver to check a formula, the query has the form `(check A solver B)`, where `A` is the model of the formula to be checked, `B` is the solver used. For example, `(check ((x + y) = z) solver Z3)` checks the formula `((x + y) = z)` in the solver Z3.

4. Methodology of applying the integrated approach by the example of the C-light language

In this section, we consider how to add a new programming language and the methods and techniques of program verification for it to the Spectrum system by the example of the C-light language. To add a language L, it is required to describe a model of L-programs in Atoment, implement L-adapter that translates an (annotated) L-program into its model, and specify methods and techniques of verification of L-programs in Atoment. They together form an L-component of the Spectrum system.

The C-light-component of the system includes the C-program model, C-adapter and specification of two-stage C-program verification method [19, 20] in Atoment. A two-stage method is applied to the representative subset C-light of the C language. In the first stage, an annotated C-light program is translated into the subset C-kernel of C-light [21, 22]. In the second stage, verification conditions for the resulting C-kernel program are generated based on the C-kernel axiomatic semantics [20].

Let us consider the examples of translation of some C-light constructions into their models in Atoment. Other C-light constructions are translated into their models in a similar way.

The annotated while statement with the condition `A`, body `B` and invariant `I` corresponds to the element with the value `(while-statement@ condition A body B inv I)`. Thus, this element is an instance of the concept `while-statement` and has the attributes `condition`, `body` and `inv`, which define the condition `A`, body `B` and invariant `I` of the while statement, respectively.

The if statement with the condition `A`, the then branch `B` and the else branch `C` corresponds to the element with the value `(if-statement@ condition A then B else C)`.

The break statement corresponds to the element with the value `(break-statement@)`, and the continue operator corresponds to the element

with the value `(continue-statement@)`.

The return statement with the expression `A` corresponds to the element with the value `(return-statement@ expression A)`, and the return statement without the expression corresponds to the element with the value `(return-statement@)`.

The switch statement with the governing expression `A` and the body `B` corresponds to the element with the value `(switch-statement@ expression A body B)`.

The block statement `{`A_1 ... A_n`}` corresponds to the element with the value `(block-statement@` A_1 ... A_n`)`.

The expression statement `A;` corresponds to the element with the value `(expression-statement@ expression A)`.

C-light expressions can use the same translation scheme as C-light statements. But to simplify a combination of expressions with formulas of the annotation language in the rules of the C-kernel axiomatic semantics, we use another translation scheme. In this scheme, C-light expressions are translated to the Atoment expressions that are structurally equivalent to them (up to parentheses). For example, the C-light expression `A + B * C` is translated to the Atoment expression `(A + (B * C))`.

It is important to note that the translation matches each construction of an annotated program in a target programming language with exactly one element in the Atoment language. Also, the translation preserves the terminology of the target language. Thus, the translation satisfies the principles of structural identity and naturalness. In addition, for each construction of the target language, its model has a number of reserved properties (attributes). In particular, these properties describe the position of the construction in the program in the target language and are used for reverse mapping of elements of the model to the corresponding constructions of the target program.

Models of programs in other programming languages are constructed in a similar way on the basis of this technique.

Let us consider the specification of the conversion which translates a C-light program into a C-kernel program [21, 22] by the example of elimination of the break statement. The conversion recursively analyses the statements of the C-light program until it reaches a break statement. All occurrences of this statement are replaced by goto statements in accordance with the following rules:

```
switch(e){A break; B} -> {switch(e){A goto L; B} L:}
while(e){A break; B} -> {while(e){A goto L; B} L:}
```

Here `A, B` are program fragments, L is a label.

First of all we define the concept `label-place`. It describes the place to which the control is transferred if we met a break statement. The attribute

`statement` of this concept points to the `... {A break; B}` if we have not met a break statement in `A`, and points to the block `{... L:}` otherwise. The label `L` is defined by the attribute `label` of this concept.

The conversion is described by the expression `(eliminate-break A label-place LP)` with the following semantics:

```
(sem (eliminate-break A label-place LP) var (A LP)
 where (((A is statement) or (A = ())) and
        (LP is label-place)) :=
 (seq@
  (if ((A is while-statement) or (A is switch-statement))
   then (seq@
         (eliminate-break (A . body)
          label-place (new (label-place@ statement A)))
         (return)))
  (if (A is break-statement)
   then
   (seq@
    (var@ Lab)
    (if (LP . label)
     then (Lab := (LP . label))
     else // create a new block {... L:}
     (seq@
      (Lab := (new))
      (var@ LabSt St St1)
      (LabSt := (new (labelled-statement@ label Lab)))
      (St := (LP . statement))
      (St1 := (new))
      (value St1 := (value St))
      (value St := (block-statement@ St1 LabSt))))
    (value A := (new (goto_statement@ label Lab)))
    (return)))
  (if (A is block-statement)
   then
   (seq@
    (foreach X in A do (eliminate-break X label-place LP))
    (return)))
  (if (A is if-statement)
   then
   (seq@
    (eliminate-break (A . then) label-place B)
    (eliminate-break (A . else) label-place B)
    (return)))
```

```
... // other statements
(if (A = ()) then (return))))
```

The expressions (A$_1$ and ... and A$_n$) and (A$_1$ or ... or A$_n$) define the conjunction and disjunction of the expressions A$'_1$, ..., A$'_n$, respectively. The false value is specified by the empty expression (), and the true value is specified by any other expression.

The expression (A is B), if it is not redefined by semantic schemas for specific values of A and B, returns true, if A$'$ is an instance of the concept B$'$, and () otherwise.

The expression (seq@ A$_1$ ... A$_n$) successively calculates the expressions A$_1$, ..., A$_n$ and returns A$'_n$.

The expression (if U then V else W) is similar to a usual conditional statement. The condition U$'$ is false if and only if it returns the false value ().

The expression (U . V) returns the value of the attribute V$'$ for the element U$'$.

The expression (U := V) sets the variable U to V$'$.

The expression (new U) evaluated in a state s generates a new element E with the value U$'$, where U$'$ is the result of evaluation of subexpressions of all slots of the expression U, and expands the domain of s on this element (s$'$(E) = U$'$). In our case, this expression is used to create new labels and new instances of the concepts label-place, labelled-statement, and goto-statement. A special case (new) is an abbreviation for (new ()).

The expression (var@ U$_1$ ... U$_n$) declares the variables U$_1$, ..., U$_n$, setting them to (). The existence domains of these variables are within the nearest parentheses surrounding this expression.

The expression (value A := B) sets the element A$'$ to the value B$'$. The expression (value A) evaluated in a state s returns the expression s(A$'$).

The expression (foreach X in Y do Z) goes over value slots of the expression s$'$(Y$'$) from left to right and for each of them evaluates the expression obtained by replacing all occurrences of the atoment X in vv$'$(Z, {X}) by this slot. Here (Y$'$, s$'$, vv$'$) is a result of evaluation of the expression Y.

Let us now consider the specification of the verification condition generator for the C-kernel language. We will describe the generic structure of the generator and the specification of the rule for the while statement. Rules for other statements are described in a similar way. The generator implements the forward strategy which is applied to the first operator S of the program fragment $S\,D$ of the Hoare triple $\{P\}\,S\,D\,\{Q\}$, where P and Q are pre- and postconditions. The rule for the while statement in this case has the form:

$$\frac{P \Rightarrow Inv \quad \{Inv \wedge B\}\,C\,\{Inv\} \quad \{I \wedge not(B)\}\,D\,\{Q\}}{\{P\}\,S\,D\,\{Q\}},$$

where Inv is the invariant of the while statement S of the form $while(B)\,C$.

The generator is described by the expression `(generateVC A)`, where `A` is a list of Hoare triples. This expression returns a list of generated verification conditions for Hoare triples from the list `A`:

```
sem (generateVC A) var A where (A is (list Hoare-triple)) :=
(seq@
 (if (A is empty-expression) then (return)
  else
  (seq@
   (var@ HT FRAG S HT-pre HT-post)
   (HT := (A . 1 right@@)) (FRAG := (HT . fragment))
   (HT-pre := (HT . pre)) (HT-post := (HT . post))
   (if (FRAG is empty-expression)
    then (return (exp (HT-pre implies HT-post))))
   (var@ S) (S := (FRAG . 1))
   (// the rule for the while statement
    if (S is while-statement)
    then
    (seq@
     (var Inv B) (Inv := (S . inv)) (B := (S . condition))
     (A +=
      (new (Hoare-triple@ pre (exp (Inv and B))
            fragment (S . body) post Inv))
     (delete FRAG . 1)
     (A +=
      (new (Hoare-triple@ pre (exp (Inv and (not B)))
            fragment FRAG post HT-post))))
    (return (add (exp (HT-pre implies Inv))
             to (generateVC A)))))
   // rules for other statements
   ... )))
```

The concept `Hoare-triple` describes Hoare triples. Instances of this concept have the obligatory attributes `pre`, `post`, and `fragment`. The attributes `pre` and `post` match P and Q and define pre- and postconditions, respectively. The attribute `fragment` matches the program fragment $S\,D$.

Instances of the concept `(list T)` are lists of the instances of the concept of `T`. The concept `empty-expression` has the only instance, the empty expression `()`.

The expression `(A . B right@@)` returns the `B`′-th element of the list `A`′ counting from the end. The property `right` means that elements are counted from right to left.

The expression `(exp A)` returns the expression `vv(A)` and does not change the current state and the current variable meaning.

The expressions `(A implies B)`, `(A and B)` and `(not A)` represent the corresponding propositional formulas of the annotation language.

The expression `(A += B)` adds the expression B' to the end of the list A'. The expression `(delete A . B)` deletes the B'-th element from the list A'. The expression `(add A to B)` adds the element A' to the end of the list B' and returns the resulting list.

In conclusion of this section, let us consider a typical user session. In this session the user applies the two-stage method to a sorting program written in C (for simplicity, we are restricted only to user queries):

```
> (include "spectrum")
> (var@ Model Hoare-Triple-list VC-list Result)
> (Model := (generate-model "sorting.c" lang c-light))
> (include "C-light to C-kernel translator.cs")
> (Model := (translate Model from C-light to C-kernel))
> (include "C-kernel VC generator.cs")
> (Hoare-Triple-list += Model)
> (VC-list := (generateVC Hoare-Triple-list))
> (foreach X in VC-list do (Result += (check X solver Z3)))
```

5. Conclusion

A modern trend in the field of program verification is transition from the development of verification methods which are applied to small programs in model languages to development of verification methods which are applied to big software systems in industrial programming languages. The tendency is to pick out program properties which are important in practice and to develop specialized methods of analysis and verification for these cases. Unification and formalization of descriptions of these properties and verification methods for them are an important open problem. The use of combinations of verification methods is also peculiar to industrial verification. Development of tools for accumulation, analysis and formalization of experience in the field of integration of different verification methods is another important open problem.

To solve these problems, the paper presents a new approach to analysis and verification of imperative programs, which allows us to integrate, unify and combine the methods and techniques of analysis and verification of imperative programs, accumulate and use knowledge about them. The components of this approach (the domain-specific language Atoment and the extensible multilanguage system Spectrum of analysis and verification of programs) have been considered. The methodology of application of the ap-

proach to verification of imperative programs by the example of the C-light language has been described.

We plan to apply this approach to development of

- program models of programming languages that are extensively used in practice (Java, C/C++, C# and so on),
- formal executable specifications of these languages on the basis of operational ontological approach [23],
- the methods of analysis and verification of program models and software systems.

References

[1] John A.K., Sharma B., Bhattacharjee A.K., Dhodapkar S.D., Ramesh S. Detection of runtime errors in MISRA C programs: A deductive approach // Proc. of Conf. SAFECOMP-2007. – Lect. Notes. Comput. Sci. – 2007. – Vol. 4680. – P. 491–504.

[2] PVS Home Page. – http://pvs.csl.sri.com/.

[3] Verisoft Home Page. – http://www.verisoft.de/.

[4] Isabelle Home Page. – http://www.cl.cam.ac.uk/research/hvg/isabelle/

[5] Why Home Page. – http://why.lri.fr/.

[6] Frama-C Home Page. – http://frama-c.com/.

[7] ACSL. – http://frama-c.com/acsl.html

[8] Berg J., Huisman M., Jacobs B., Poll E. A type-theoretic memory model for verification of sequential Java programs // Recent Trends in Algebraic Development Techniques. – Lect. Notes Comput. Sci. – 2000. – Vol. 1827. – P. 1–9.

[9] JML Home Page. – http://www.cs.ucf.edu/ leavens/JML/.

[10] David Cok. OpenJML: JML for Java 7 by Extending OpenJDK // NASA Formal Methods. – Lect. Notes. Comput. Sci. – 2011. – Vol. 6617. – P. 472–479.

[11] ESC/Java Home Page. – http://kind.ucd.ie/products/opensource/ESCJava2/.

[12] Chalin P., Kiniry J.R., Leavens G.T., Poll E. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In: FMCO 2005. – Lect. Notes. Comput. Sci. – 2006. – Vol. 4111. – P. 77–101.

[13] Spec# Home Page. – http://research.microsoft.com/en-us/projects/specsharp/.

[14] Barnett M., Leino R., Schulte W. The Spec# programming system: An overview. In CASSIS 2004. – Lect. Notes. Comput. Sci. – 2004. – Vol. 3362. – P. 49–69.

[15] Barnett M., Chang B., DeLine R., Jacobs B., Leino R Boogie: a modular reusable verifier for object-oriented programs // FMCO 2005. – Lect. Notes. Comput. Sci. – 2006. – Vol. 4111. – P. 364–387.

[16] Anureev I.S. The Atoment Language: Syntax and Semantics. – Novosibirsk, 2010. – 47 p. – (Rep./SB RAS. IIS; N 157). (In Russian).

[17] Anureev I.S. The Atoment Language: Standard Library. – Novosibirsk, 2010. – 47 p. – (Rep./SB RAS. IIS; N 158). (In Russian).

[18] Anureev I.S. Introduction to the Atoment language // Bull. Novosibirsk Comp. Center. Ser.: Comput. Sci. – Novosibirsk, 2010. – IIS Special Iss. 31. – P. 1–16.

[19] Nepomniaschy V. A., Anureev I.S., Mikhailov I.N., Promsky A.V. Towards verification of C programs: C-Light language and its formal semantics // Programming and Computer Software. – 2002. – N 6. – P. 314–323.

[20] Nepomniaschy V. A., Anureev I.S., Promsky A.V. Towards verification of C programs: axiomatic semantics of the C-kernel Language // Programming and Computer Software. – 2003. – N 6. – P. 338–350.

[21] Nepomniaschy V. A., Anureev I.S., Promsky A.V. Towards verification of C programs: C-Light language and its transformational semantics // Problem of Programming. – 2006. – N 2-3. — P. 359-368. (In Russian).

[22] Nepomniaschy V. A., Anureev I.S., Mihailov I.N., Promsky A.V. Verification-oriented C-light language // The collected scientific papers «Formal methods and models of informatics», Ser. System Informatics. – Novosibirsk: SB RAN Publishers, 2004. – N 9. – P. 51–134. (In Russian).

[23] Anureev I.S. Operational ontological approach to formal programming language specification // Programming and Computer Software. – 2009. – N 1. – P. 35–42.