Bull. Nov. Comp. Center, Comp. Science, 26 (2007), 111–132
© 2007 NCC Publisher

The C#-light project: solution of some verification challenges

A. V. Promsky

Abstract. The evolution of formal methods allowed us to overcome many obstacles in verification of procedural programs. However, wide spreading of object-oriented languages has brought new challenges, even in the case of sequential programs. These problems were thoroughly examined by ESC/Java and Spec# teams in their papers [10, 12], though in many cases they just state the presence of the challenge. This paper presents an overview of some problematic issues and a three-level approach to their solution in the C#-light project.

1. Introduction

The fully automatic verification of programs is a tempting and hardly accessible goal of modern programming. The active use of object-oriented languages has raised the difficulty to a new level. New formal methods and specification languages are required, because the classical Hoare approach and first-level logics are no more adequate for the task.

This paper purposes two aims. First, it gives an overview of some well-known problems of OO program verification [10, 12] and, second, it presents a three-level approach to C#-light program verification in which these problems were successfully solved.

The three-level approach is an attempt to reconcile two controversial goals. First of all, a wide coverage of C# is a requirement of practical (and, of course, commercial) interest. That is why C#-light includes a great part of C# [1], except for threads and some realization-dependent constructs. The formal definition of C#-light has the form of a structured operational semantics. Secondly, regardless of its limitations, the axiomatic approach [2, 9] is still the best choice. As a result, we chose a compact core of C#-light — the C#-kernel language and developed its Hoare-like logic. The first level of our approach consists in translation from C#-light into C#-kernel. The formally defined set of translation rules admits the proof of equivalence. At the second stage the lazy verification conditions are derived in C#-kernel axiomatic semantics. Laziness results from the fact that these conditions are not the final assertions of the specification language. Indeed, they can contain special terms related to the issues of control transfer and dynamic binding. Refinement of these terms is beyond the capacity of the Hoare logic and is performed at the third level.

In order to test our method, we verified most programs from the well-known collection [10] illustrating the verification challenges. Originally written in Java, they suit the C# syntax with minimal changes. We also examined some innovations of C#, such as delegates and events.

In the course of our experiments, we developed some strategies of verification condition simplification. In the case of OO languages, the size of verification conditions can overgrow all reasonable limits, so it becomes an additional problem.

The rest of the paper is organised as follows: Section 2 briefly describes the C#-light project. Some strategies suitable for simplification of verification conditions are considered in Section 3. Section 4 presents some verification challenges together with our comments on their verification. Section 6 is a conclusion.

This research has been partially supported by grant 04-01-00114a from RFBR and by grant N 14 for young scientists from SB RAS.

2. Preliminaries

Let us briefly describe the C#-light project and present information required for further reading. The details can be found in [5, 16, 17, 20].

2.1. The C#-light language

As it was said, C#-light is a sequential language. Besides the threads, the following constructs are forbidden: 1) attributes; 2) destructors; 3) the `using` statement; 4) the `checked` and `unchecked` operators; 5) unsafe code; 6) pre-processing directives.

It should be noted that constructs listed above either require the low-level knowledge about a concrete implementation of .Net platform, or their contribution to C# expressivity is not significant. However, the C#-light language is still a representative subset of C#. In comparison with Java, it includes properties, events, delegates and indexers.

In addition, C#-light proposes the syntax for annotations in the form of comments `/// <a> R </a>`, where `R` is an assertion (see the next Section).

2.2. Assertion language

The choice of an assertion language, which is used for pre-/postconditions and invariants, plays a crucial role in the Hoare method. Indeed, many properties of the Hoare logic depend on the expressiveness of assertions. The classical first-order language is not sufficient for modern programming languages and is a target for replacement/extension. As the examples we can mention JML [10] and HOL [18]. In our case, we added functors and some elements of λ-calculus.

Types. Types of the assertion language include a universal set $\mathcal{U}$, the set of C# identifiers $\mathcal{N}$, the set of abstract type names $\mathcal{T}$, functions $T \to T'$ and Cartesian products $T \times T'$, where T, T' are types. Let us note that $\mathcal{U}$ includes, at least, all C# literals, the set of memory sells $\mathcal{L}$, natural numbers and undefined value ω.

Expressions. Expressions of assertion languages include constants, variables, terms $f(s_1, \ldots, s_n)$ and λ-terms $\lambda(x, s)$ with a standard interpretation. Assertions are boolean expressions built by standard rules. We also fix the standard function $upd(f, c, v)$ which updates the function f at the argument c with the value v.

For convenience we will use the usual infix notation in our examples.

2.3. The C#-kernel language

Strictly speaking, C#-kernel is not a syntactic subset of C# as long as it contains a new class of constructions — *meta-instructions.* However, semantically we are still in the scope of C#-light. Let us briefly list the main features of C#-kernel:

- fully qualified names are used instead of namespaces and `using`-directives;
- the set of statements consists only of an expression-statement, an `if` statement, a `goto` statement and a block;
- the operators `||`, `&&`, `?:`, `new` and all assignments are forbidden;
- the sets of labels, local variable names, local constant names and type names are disjoint and do not contain duplicates within a program;
- a method and delegate invocation has a normalized form x`.`y`(`$z_1, ..., z_n$`)` or y`(`$z_1, ..., z_n$`)`, where x, y are names and z_1, ..., z_n are names or literals;
- instead of C# assignments, the metainstruction x `:=` e is used;
- objects are created by a low-level metainstruction `new_instance()`;
- types are explicitly initialized by metainstruction `Init(`C`)`;
- exceptions are handled by metainstructions `catch(`T`, `x`)` and `catch(`x`)`;
- declarations of fields and constants of classes and structs do not contain initializers. Instead, two methods `SFI` and `IFI` are reserved for each class declaration to perform static and instance field initialization, respectively.

The idea of metainstructions may seem too exotic. Indeed, in our approach of C program verification [15], the C-kernel language is a pure subset of the C-light language. To clarify the situation, the following analogy is proposed. You can insert the assembly language blocks into C programs. Also you can translate a C program into an assembly program which, in turn, is translated into binary commands. The binary-level semantics of assembly program also provides semantics for the original C program. In a manner of speaking, C#-kernel is the assembly language of a C#-light abstract machine. The procedural C-light did not require such an assembly language, but the object-oriented C#-light does.

This analogy with assembly language should not confuse the reader. C#-kernel is the intermediate language and its use is hidden from the user.

2.4. The C#-light abstract machine

In the classical approach [2], suitable for Pascal, a state of an abstract machine is a mapping from program variable names into values. Thus, variable names are treated as unique labels of memory cells, and the state takes the label as an argument and returns its contents. Two main reasons make this approach inconsistent in C#. First, there are local variables with a limited life time, so the domain of a state can dynamically change. Second, there are references, so every change of the state through a label must track all possible aliases.

To overcome this obstacle, we use such a notion as memory management. Remember that access by name only exists in the input code. The executable code implies access by address. We introduce this two-level "*name* → *address* → *value*" access into our states. We do not identify the addresses with any concrete type. The uninterpreted symbolic constants are sufficient. We only require that every address be unique.

Besides the two-level access, we also need to recognize object types, the internal structure of composite objects and thrown exceptions. Thus, a *state* is a mapping from the following abstract machine metavariables into their values:

Name	**Type**	**Description**
L	$\mathcal{N} \to \mathcal{L}$	Program variable addresses
V	$\mathcal{L} \to \mathcal{U}$	Program variable values
T	$\mathcal{N} \cup \mathcal{U} \to \mathcal{T}$	Program variable types
$L2$	$\mathcal{U} \times (\mathcal{N} \cup \mathcal{N}at) \to \mathcal{L}$	Composite object structure
$V0$	$\mathcal{U}$	Last evaluated value
E	$\mathcal{U}$	Last caught exception

We will often use the abbreviation *mvs* instead of the tuple $[L, V, T, L2, V0, E]$.

2.5. Translation from C#-light into C#-kernel

The basis for this translation was developed in our work on C program verification [15]. Complex C-light constructs were translated into the sequences of simple C-light constructs. The features of C# require translation of some simple constructs as well. The idea is as follows: if the interpretation of a construct in the operational semantics leads to a series of changes of metavariables, we can replace the original construct by an explicit sequence of these changes. That is where the connection between metavariables and metainstructions is established. Indeed, for every metainstruction of the form x := e, x is one of metavariables.

For example, the declaration `S x;` is replaced by

```
new_instance();
L := upd(L, x, V0);
T := upd(T, L(x), Loc(S));
```

The metainstruction `new_instance` results in allocation of a new memory cell, and the new address is implicitly stored in `V0`. In the second line, this new address is associated with the name `x`. The third line signals that the new cell contains an object of type `S`.

2.6. Axiomatic semantics of C#-kernel

The classical Hoare logic is a calculus of triples $\{P\}\,S\,\{Q\}$, where P and Q are assertions and S is a program. Such a formula is valid if the truth of the *precondition* P and termination of S implies the truth of the *postcondition* Q. The principal factor is that programs and assertions "share" the variables and assignments. Thus, the truth of assertions is expressed by the sets of states in which assertions are valid. For example, the formula $x = 3$ corresponds to all states where the memory cell x contains the value 3. The classical Hoare triple for an assignment looks like

$$\{P(x \leftarrow e)\}\,x := e\,\{P\} \tag{1}$$

or equivalently

$$\{P\}\,x := e\,\{\exists x'.(P(x \leftarrow x') \land x = e(x \leftarrow x'))\}\ , \tag{2}$$

where '$\leftarrow$' denotes the substitution.

At the first sight, the use of metavariables destroys this concept. However, not only the variables turned into metavariables. The assertion and programming languages changed as well. As a result, we have a classical case: a rather simple programming language, and every program has only six variables (now called metavariables); the assertions over these metavariables are written in the specification language. This leads to a decisive idea: **we can adapt the classical Hoare logic for C#-kernel**.

However, the simplicity of the logic structure is obtained by the price of complex assertions. For example, the formula $x = 3$ is now expressed as $V(L(x)) = 3$.

We defined the logic of C#-kernel as a forward Hoare logic, when the leftmost construct is interpreted. The disadvantage consists in the complex form (2) for an assignment. On the other hand, such a system allows us to discard the wittingly true triples. The proof environment, which signals about the current function, initialized classes and exceptions, also greatly reduces the proof.

The complete axiomatic semantics for C#-kernel can be found in [16]. As an example, let us consider the rule for the metainstruction `new_instance`.

$$\frac{\mathit{Env}^- \vdash \{\exists V0' \ \exists d(\mathit{newp}(d, L, V, L2) \wedge P(V0 \leftarrow V0') \wedge V0 = d)\}A\{Q\}}{\mathit{Env}^- \vdash \{P\}\, \texttt{new\_instance();}\ A\,\{Q\}}\ .$$

Here A is a sequence (possibly empty) of statements. The special predicate $\mathit{newp}(d, L, V, L2)$ guarantees that d is a new memory cell.

2.7. The lazy verification condition refinement

In practice, the axiomatic semantics is used in the form of a verification condition generator (VCG). It produces lemmas in the assertion language which must be proven to guarantee correctness of a program. In our approach, VCG can produce inconclusive or *lazy* verification conditions. It results from the problems of dynamic binding and loop invariants displacement.

Dynamic binding. When the user declares a virtual function in a class, he defines the interface to the whole set of overriding functions in the derived classes. The information which function is actually invoked cannot be resolved statically. However, the axiomatic semantics only expresses the symbolic computation without real modeling of the dynamic behaviour. Thus, the polymorph function invocation in the general case requires quantification over the set of overriding functions [18]. The resolution of such a quantifier, though finite for every finite program, complicates its verification.

Nevertheless, the forward Hoare logic proposes a partial solution. Indeed, in this case the information about object values (and, maybe, about the dynamic type) is accumulated in the precondition. When does the precondition "know" the object's dynamic type? First, when the object creation operation precedes the invocation point in one linear fragment. Second, when a user himself provided this information. In this case the invocation rule can use the specification of an appropriate function. The worst case can also be imagined. For example, if `T` is the base class containing a virtual function `f`, then for the procedure

```
    void g(T x){ ... x.f(); ... }
```

in the general case we cannot specify the concrete type of `x`. Then for the invocation `x.f();` we are forced to look through all descendants of type `T`.

We do not include the algorithms, which extract the dynamic type from the precondition, into axiomatic semantics. Instead, we use the lazy term $CALL()$ which results in a very simple axiomatic rule:

$$\frac{Env \vdash \{CALL(f, x, [args], mvs, \lambda(mvs, P))\}\, A\, \{Q\}}{Env^{-} \vdash \{P\}\, x\,\texttt{.}\,f\,\texttt{(}args\texttt{)}\,\texttt{;}\; A\, \{Q\}}\ .$$

The formal refining algorithm for $CALL$ is described in [16]. If f is a simple function or f is a virtual function and precondition P "knows" the dynamic type of x, then $CALL$ turns into a conjunction of P and of appropriate function specifications with some substitutes over metavariables. Otherwise, the quantifier over virtual functions will be added.

We use the analogous term $DELCALL$ to handle delegates [16].

Loop invariants. The problem is that we rewrite the loops `while`, `do`, `for` and `foreach` with the help of the `goto` statement. The treatment of `goto` in the Hoare logic is also based on the notion of invariant. But in the general case we cannot use the original loop invariant as a new label invariant. In addition, during translation various "forward gotos" can appear and they do not form loops at all.

And again, the lazy terms can help us. Every time the axiomatic semantics meets the statement `goto L` or a statement labeled by `L`, it introduces the symbolic formula $INV(mvs, L)$. The refining algorithms for INV can be found in [16], but in brief the idea is as follows. Among the verification conditions we look for formulas of the form $A \Longrightarrow INV(mvs, L)$, and we take the disjunction of all A. Obviously, this formula is true every time when the control reaches the label `L`. And then every occurrence of $INV(\overline{e}, L)$ is replaced by that disjunction with substitution of $\overline{e}$ instead of mvs.

3. Verification condition simplification

The two-level access model resolves the problems of composite objects and aliasing but leads to explosion in the size of verification conditions (VC). Let us consider a simple example in Figure 1.

Indeed, the original C#-light program is so simple that it could be verified in the classical Hoare logic [9]. However, in our approach we verify the source program through the implicit verification of an equivalent C#-kernel program in the right column. Let us compare the VCs obtained in each approach.

In each case we can use `true` as a precondition.

C#-light program	**C#-kernel program**

`{ int x, y, z;`
`  x = 1;`
`  y = 2;`
`  z = 3;`
`  z = 4;`
`}`
	`{ Init(int);`
	`  new_instance();`
	`  L := upd(L, x, V0);`
	`  T := upd(T, L(x), Loc(int));`
	
	`  new_instance();`
	`  L := upd(L, y, V0);`
	`  T := upd(T, L(y), Loc(int));`
	
	`  new_instance();`
	`  L := upd(L, z, V0);`
	`  T := upd(T, L(z), Loc(int));`
	
	`  V := upd(V, L(x), 1);`
	`  V := upd(V, L(y), 2);`
	`  V := upd(V, L(z), 3);`
	`  V := upd(V, L(z), 4);`
	`}`

Figure 1. A small example

The postcondition for the classical logic is $\mathtt{x} = 1 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 4$. Applying the rule (1) 4 times, we obtain the true assertion:

$$\mathsf{true} \Longrightarrow (1 = 1 \wedge 2 = 2 \wedge 3 = 3). \tag{3}$$

The two-level access in the C#-kernel logic requires the following precondition: $V(L(\mathtt{x})) = 1 \wedge V(L(\mathtt{y})) = 2 \wedge V(L(\mathtt{z})) = 4$.

Besides the more complex precondition, the number of VCs doubles because of the metainstruction `Init`. If a type is not yet initialized, this metainstruction triggers the process of static initialization, otherwise it is ignored. Let us consider the VC without initialization (Figure 2). For simplicity we omit the existential quantifiers binding the indexed variables. The predicate $SI(\mathtt{int}, \ldots)$ signals that the type `int` was actually initialized. The proof environment is also discarded because such a simple program does not depend on it.

The size of VC surprises. Obviously, verification of real programs can lead to immense assertions which can easily overcome the capacity of theorem provers. That is the price of the detailed memory model of C#-light/kernel. Thus, we need some simplification strategies which must precede the proof stage.

Strategy 1. Let us have a close look at the boxed conjuncts. The formula $E = \omega$ signals that the value of the metavariable E is undefined. In

$$
\left[\begin{array}{ll}
newp(d_3, L_3, V_1, L2) & \wedge \\
newp(d_2, L_2, V_1, L2) & \wedge \\
newp(d_1, L_1, V_1, L2) & \wedge \\
\boxed{E = \omega} & \wedge \\
\boxed{SI(\mathtt{int}, L_1, V_1, T_1, L2)} & \wedge \\
\boxed{V0_1 = d_1} & \wedge \\
L_2 = upd(L_1, \mathtt{x}, V0_1) & \wedge \\
\boxed{T_2 = upd(T_1, L_2(\mathtt{x}), Loc(\mathtt{int}))} & \wedge \\
\boxed{V0_2 = d_2} & \wedge \\
L_3 = upd(L_2, \mathtt{y}, V0_2) & \wedge \\
\boxed{T_3 = upd(T_2, L_3(\mathtt{y}), Loc(\mathtt{int}))} & \wedge \\
\boxed{V0 = d_3} & \wedge \\
L = upd(L_3, \mathtt{z}, V0) & \wedge \\
\boxed{T = upd(T_3, L(\mathtt{z}), Loc(\mathtt{int}))} & \wedge \\
V_2 = upd(V_1, L(\mathtt{x}), 1) & \wedge \\
V_3 = upd(V_2, L(\mathtt{y}), 2) & \wedge \\
V_4 = upd(V_3, L(\mathtt{z}), 3) & \wedge \\
V = upd(V_4, L(\mathtt{z}), 4) &
\end{array}\right]
\Longrightarrow
\left[\begin{array}{ll}
V(L(\mathtt{x})) = 1 & \wedge \\
V(L(\mathtt{y})) = 2 & \wedge \\
V(L(\mathtt{z})) = 4 &
\end{array}\right]
$$

Figure 2. The verification condition

other words, no exception was thrown during the execution. In this example the omitted environment also states the absence of exceptions, so this formula is true indeed. Analogously, handling of `Init(int)` adds the type `int` to the set of initialized types in environment. Thus, the formula $SI(\mathtt{int}, L_1, V_1, T_1, L2)$ is odd.

Further, the equalities of the form $variable_1 = variable_2$ are just renamings. We can simply prolong one of the variables.

Finally, all we need is to proof that the values of `x`, `y` and `z` are 1, 2 and 3, respectively. This proof requires the information about metavariables L and V, and the types are irrelevant. Moreover, the consequent does not contain any occurrence of the metavariable T. Thus, we can omit the conjuncts for T_2, T_3 and T. To justify such a discard, we have to show that the formula

$$
\begin{array}{ll}
T_2 = upd(T_1, L_2(\mathtt{x}), Loc(\mathtt{int})) & \wedge \\
T_3 = upd(T_2, L_3(\mathtt{y}), Loc(\mathtt{int})) & \wedge \\
T = upd(T_3, L(\mathtt{z}), Loc(\mathtt{int})) &
\end{array}
$$

is valid. This can be easily done.

Strategy 2. In the consequent of VC, the variables are matched against the constants. So, we can split VC into three formulas in accordance with the number of conjuncts in the consequent. Let us consider one of these new formulas (strategy 1 is supposed to be already applied):

$$\left[\begin{array}{ll} \mathit{newp}(d_3, L_3, V_1, L2) & \wedge \\ \mathit{newp}(d_2, L_2, V_1, L2) & \wedge \\ \mathit{newp}(d_1, L_1, V_1, L2) & \wedge \\ L_2 = \mathit{upd}(L_1, \mathtt{x}, d_1) & \wedge \\ L_3 = \mathit{upd}(L_2, \mathtt{y}, d_2) & \wedge \\ L = \mathit{upd}(L_3, \mathtt{z}, d_3) & \wedge \\ V_2 = \mathit{upd}(V_1, L(\mathtt{x}), 1) & \wedge \\ V_3 = \mathit{upd}(V_2, L(\mathtt{y}), 2) & \wedge \\ V_4 = \mathit{upd}(V_3, L(\mathtt{z}), 3) & \wedge \\ V = \mathit{upd}(V_4, L(\mathtt{z}), 4) & \end{array}\right] \Longrightarrow V(L(\mathtt{z})) = 4 \ . \tag{4}$$

Strategy 3. The type `int` is not a reference type, and C#-light does not support pointers. This guarantees that the object `z` does not have aliases, i.e. the assignments to `x` and `y` affect `z` only if the expressions over `x` and `y` are assigned to `z`. Let us begin to unfold the antecedent from the rightmost conjunct. If a term *upd* binds `z` with a value which is independent from `x` and `y`, then this *upd* can be substituted into the consequent. Otherwise, the *upd* is ignored. This unfolding takes into account the fact that the lower (righter) conjunct corresponds to the later assignment in the source program.

$$\begin{array}{l} \left[\begin{array}{ll} \mathit{newp}(d_3, L_3, V_1, L2) & \wedge \\ \mathit{newp}(d_2, L_2, V_1, L2) & \wedge \\ \mathit{newp}(d_1, L_1, V_1, L2) & \end{array}\right] \\ \Longrightarrow \mathit{upd}(V_4, \mathit{upd}(L_3, \mathtt{z}, d_3)(\mathtt{z}), 4)(\mathit{upd}(L_3, \mathtt{z}, d_3)(\mathtt{z})) = 4 \ . \end{array} \tag{5}$$

The standard semantics of *upd* (Section 2.2) provides the truth of this assertion. Though (5) is still bigger than (3), it is significantly simpler than VC from Fig. 2. Moreover, the proof does not depend on the fact that d_1, d_2 and d_3 are new addresses, so the whole antecedent can be discarded.

In the next Section we will demonstrate the VCs after the application of simplification strategies.

4. Verification challenges and their solution

The majority of examples are borrowed from [10]. Originally written in Java, they were translated into C#. Both languages are syntactically (and semantically) close, so the translation is trivial.

It should be noted that [10] contains two additional examples which represent the real verification challenges. The first one is the re-entrance problem which consists in potential violation of the class invariant during the nested method invocations. A common solution is to allow officially such a violation. However, we do not use the concept of a class invariant, so this problem is not crucial. The second one concerns the termination problem. Moreover, non-termination there resulted from overflowing. In our approach, neither total correctness notion, nor overflowing is modeled, so that example could not be verified.

4.1. Aliasing

Aliasing can make the classical Hoare logic fully inconsistent. For example, if `x` and `y` point to the same memory object, then the following Hoare triple becomes incorrect:

$$\{x = 1\}\,\texttt{y := 2}\,\{x = 1\}\ .$$

The explicit memory modeling allows us to find a solution. Consider the following program:

```
class C {
    C a;
    int i;

    C(){ a = null; i = 1; }
}

class Alias {
    int m(){
        C c = new C();
        c.a = c;
        c.i = 2;
        return c.i + c.a.i;
    }
}
```

Here, the field `i` is accessed via the object `c`, as well as via an aliased reference to `c` itself in the field `a`. The specifications for program methods are as follows:

Pre(`C.C`)	:	true
Post(`C.C`)	:	$V(L(\mathsf{a})) = \mathsf{null} \wedge V(L(\mathtt{i})) = 1$
Pre(`Alias.m`)	:	true
Post(`Alias.m`)	:	$V0 = 4$

Verification of the constructor `C()` is not of great interest, so let us consider the method `Alias.m()`. In our axiomatic semantics, treatment of the method invocation (here, `C()`) doubles the proof tree. In combination with the explicit exception propagation [20] this leads to generation of 16 verification conditions. However, 14 VCs are tautologies as long as they have form $\mathsf{false} \Longrightarrow \Phi$. Two remaining must be proved. Let us consider one of them (after simplifications):

$$\bigwedge \left[\begin{array}{lcl} V'' & = & upd(V', L'(\mathtt{c}), V'(L'(\mathtt{x}))) \\ V''' & = & upd(V'', L2(V''(L'(\mathtt{c})), \mathtt{a}), V''(L'(\mathtt{c}))) \\ V & = & upd(V''', L2(V'''(L'(\mathtt{c})), \mathtt{i}), 2) \\ L'' & = & upd(L', \mathtt{y1}, L2(V(L'(\mathtt{c})), \mathtt{i})) \\ L & = & upd(L'', \mathtt{y2}, L2(V(L2(V(L''(\mathtt{c})), \mathtt{a})), \mathtt{i})) \\ V0 & = & V(L(\mathtt{y1})) + V(L(\mathtt{y2})) \end{array}\right] \Rightarrow V0 = 4\ .$$

The standard semantics of *upd* guarantees its truth. The second VC is similar and can be checked by analogy.

4.2. Breaking out of a loop

The exit from a loop without a loop condition check is usually performed via a `break` statement (rarely, via `goto`). This complicates the underlying control flow semantics.

In our approach the `break` statements are replaced by `goto`s. This results in appearance of new labels. In Section 2.7 we discussed how to tame these labels and their invariants.

The input C#-light program looks like this:

```
class C {
    int[] ia;

    void NegateFirst(){
        for (int i = 0; i < ia.Length; i++){
            if (ia[i] < 0){ ia[i] = -ia[i]; break; }}
    }
}
```

When the first negative element is reached, its sign changes and the loop aborts.

The unconditional exit from a loop in combination with a potential array update leads to complication of specifications:

$$\begin{array}{lcl} \mathrm{Pre}(\mathtt{NF()}) & : & \exists old : \mathtt{int[]}.\ V(L2(V(L(\mathtt{this})), \mathtt{ia})) \neq \mathsf{null} \wedge \\ & & \qquad\qquad\qquad V(L2(V(L(\mathtt{this})), \mathtt{ia})) = old \end{array}$$

$$\begin{array}{lcl} \mathrm{Post}(\mathtt{NF()}) & : & \forall i.\ (0 \leq i \leq V(L2(V(L(\mathtt{this})), \mathtt{ia})).\mathtt{Length} \Longrightarrow \\ & & \quad ((old[i] < 0 \wedge (\forall j.\ 0 \leq j < i \Rightarrow old[j] \geq 0)) \Rightarrow \\ & & \qquad\qquad V(L2(V(L(\mathtt{this})), \mathtt{ia}))[i] = -old[i]) \wedge \\ & & \quad old[i] \geq 0 \Rightarrow V(L2(V(L(\mathtt{this})), \mathtt{ia}))[i] = old[i]) \end{array}$$

$$\begin{array}{lcl} \mathrm{Inv}(\mathtt{for}) & : & 0 \leq V(L(\mathtt{i})) \leq V(L2(V(L(\mathtt{this})), \mathtt{ia})).\mathtt{Length} \wedge \\ & & (\forall j.\ 0 \leq j < V(L(\mathtt{i})) \Rightarrow (V(L2(V(L(\mathtt{this})), \mathtt{ia}))[j] \geq 0 \wedge \\ & & \qquad\qquad\qquad V(L2(V(L(\mathtt{this})), \mathtt{ia})) = old[j]))\ . \end{array}$$

The original array content is stored in an auxiliary variable *old*.

Let us consider one of verification conditions. It corresponds to execution from the beginning of `C.NegateFirst()` up to the loop entry point.

$$\left[\begin{array}{l} CALL(\texttt{<},\texttt{int},[\texttt{i},\texttt{x0}],[L,V''',T,L2,V0,E],\lambda(mvs,\Phi))\wedge \\ V = upd(V''',L(\texttt{b0}),V0) \end{array}\right] \Rightarrow \mathrm{Inv}(\texttt{for})$$

where

$$\Phi \equiv \left[\begin{array}{l} CALL(\ \texttt{get\_Length},\texttt{x1},[],[L,V'',T,L2,V0,E], \\ \qquad \lambda(mvs,(\ INV([L,V',T,L2,V0',E],\texttt{L1})\wedge \\ \qquad\qquad V0 = V'(L2(V'(L(\texttt{this})),\texttt{ia}))\wedge \\ \qquad\qquad V = upd(V',L(\texttt{x1}),V0))) \\ )\wedge \\ V = upd(V'',L(\texttt{x0}),V0) \end{array}\right].$$

To unify the semantics of C#-kernel, the standard operations are used in the syntax of the method invocation. That is why there are the terms *CALL* for the relation `<` and array property `Length`. They assert that the value of the counter `i` is less than the array length. Note that this length is stored in a new program variable `x0` which appears during translation into C#-kernel. Also, note the presence of a lazy invariant for a new label `L1`. Its refined value is the conjunction of two assertions. The first one is the precondition Pre(`NF()`), where the program variable `i` is associated with the value 0. The second one is the loop invariant Inv(`for`) propagated (as a proof precondition) through the loop body. Note that if the end point of the loop body is reached, then for the current value of `i` the elements `ia[0]`, ..., `ia[i-1]` are not negative, i.e. the loop was not aborted via `break`. Thus, the VC antecedent states that either we have a zero-length array or the loop invariant is not affected by the counter increment. By induction, this implies the consequent Inv(`for`).

4.3. Static initialization

A static field/method of the class belongs to the class itself. In particular, it can be used even if there is no instance of the class. In C# (as well as in Java) static initialization is lazy. For example, in Java a class is initialized at its first *active* use. Thus, if initialization causes some nontrivial processes in a program, then the user must precisely know where it will take place.

C# makes the situation even worse as long as a class is initialized immediately at the first active use only if a static constructor exists. Otherwise, initialization takes place at an implementation-dependent time prior to the first active use [1, §17.4.5.1]. This results in nondeterminism of the class initialization order [7].

Let us consider the following example:

```
class C {
    static bool r1, r2, r3, r4;

    static void m(){
        r1 = C1.b1;
        r2 = C2.b2;
        r3 = C1.d1;
        r4 = C2.d2;
    }
}

class C1 {
    public static bool b1 = C2.d2;
    public static bool d1 = true;
}

class C2 {
    public static bool d2 = true;
    public static bool b2 = C1.d1;
}
```

In Java[1] the situation is absolutely deterministic though it can be a surprise for a user. The first assignment in the body of the method `m()` triggers initialization of the class `C1`, which in turn triggers initialization of the class `C2`. During initialization of `C2`, initialization of `C1` suspends and the default field values are used. As a result, static field `C2.b2` is set to false and all other fields are set to true. If the first two strings in the method body of `m()` are switched, the class `C2` will be initialized before the class `C1`, resulting in all fields getting the value true.

In C# the order of initialization is unknown even though the compilation in Visual Studio 2005 provides the Java's result. So, in the general case, this program is indeed a challenge, but we can use the static constructors mentioned above. Let us add the declarations `static C1(){}` and `static C2(){}` to classes `C1` and `C2` respectively. It will guarantee the needed behavior.

The program specifications are as follows:

$$\begin{array}{lll} \mathrm{Pre}(\texttt{C.m}) & : & \mathsf{true} \\ \mathrm{Post}(\texttt{C.m}) & : & V(L(\texttt{r1})) \wedge \neg V(L(\texttt{r2})) \wedge V(L(\texttt{r3})) \wedge V(L(\texttt{r4})) \end{array}$$

According to the definition of C#-kernel (see Section 2.3), the field declarations do not contain initializers. During translation from C#-light, the static field initializers move into the special methods `SFI` which are created for each class automatically. Because these methods cannot be specified a

[1] In [10] it was not verified since ESC/Java cannot handle static initialization.

$$\left[\begin{array}{ll}
L_2 = \mathit{upd}(L_1, \texttt{C1}, d_1) & \wedge \\
V_2 = \mathit{upd}(V_1, d_1, e_1) & \wedge \\
\neg \mathit{SI}(\texttt{C1}, L_1, V_1, T_0, L2_0) \wedge \neg \mathit{SI}(\texttt{C2}, L_4, V_4, T, L2_3) & \wedge \\
L_3 = \mathit{upd}(L_2, \texttt{field\_b1}, d_2) & \wedge \\
L2_2 = \mathit{upd}(L2_1, \langle V_2(\texttt{C1}), \texttt{b1}\rangle, L_3(\texttt{field\_b1})) & \wedge \\
L_4 = \mathit{upd}(L_3, \texttt{field\_d1}, d_3) & \wedge \\
L2_3 = \mathit{upd}(L2_2, \langle V_2(\texttt{C1}), \texttt{d1}\rangle, L_4(\texttt{field\_d1})) & \wedge \\
V_3 = \mathit{upd}(V_2, L2_3(\texttt{C1}, \texttt{b1}), \mathsf{false}) & \wedge \\
V_4 = \mathit{upd}(V_3, L2_3(\texttt{C1}, \texttt{d1}), \mathsf{false}) & \wedge \\
L_5 = \mathit{upd}(L_4, \texttt{C2}, d_4) & \wedge \\
V_5 = \mathit{upd}(V_4, d_4, e_2) & \wedge \\
L_6 = \mathit{upd}(L_5, \texttt{field\_d2}, d_5) & \wedge \\
L2_4 = \mathit{upd}(L2_3, \langle V_5(\texttt{C1}), \texttt{d2}\rangle, L_6(\texttt{field\_d2})) & \wedge \\
L = \mathit{upd}(L_6, \texttt{field\_b2}, d_6) & \wedge \\
L2 = \mathit{upd}(L2_4, \langle V_5(\texttt{C1}), \texttt{b2}\rangle, L(\texttt{field\_b2})) & \wedge \\
V_6 = \mathit{upd}(V_5, L2(\texttt{C2}, \texttt{d2}), \mathsf{false}) & \wedge \\
V_7 = \mathit{upd}(V_6, L2(\texttt{C2}, \texttt{b2}), \mathsf{false}) & \wedge \\
V_8 = \mathit{upd}(V_7, L2(\texttt{C2}, \texttt{d2}), \mathsf{true}) & \wedge \\
V_9 = \mathit{upd}(V_8, L2(\texttt{C2}, \texttt{b2}), V_8(L2(\texttt{C1}, \texttt{d1}))) & \wedge \\
V_{10} = \mathit{upd}(V_9, L2(\texttt{C1}, \texttt{b1}), V_9(L2(\texttt{C2}, \texttt{d2}))) & \wedge \\
V_{11} = \mathit{upd}(V_{10}, L2(\texttt{C1}, \texttt{d1}), \mathsf{true}) & \wedge \\
V_{12} = \mathit{upd}(V_{11}, L2(\texttt{C}, \texttt{result1}), V_{11}(L2(\texttt{C1}, \texttt{b1}))) & \wedge \\
V_{13} = \mathit{upd}(V_{12}, L2(\texttt{C}, \texttt{result2}), V_{12}(L2(\texttt{C2}, \texttt{b2}))) & \wedge \\
V_{14} = \mathit{upd}(V_{13}, L2(\texttt{C}, \texttt{result3}), V_{13}(L2(\texttt{C1}, \texttt{d1}))) & \wedge \\
V = \mathit{upd}(V_{14}, L2(\texttt{C}, \texttt{result4}), V_{14}(L2(\texttt{C2}, \texttt{d2}))) &
\end{array}\right] \Rightarrow \mathrm{Post}(\texttt{C.m})$$

Figure 3. The VC for the static initialization example

priori, their semantics consists of inline body substitution. As a result, the VCs can grow significantly (see Figure 3). However, the proof is straightforward.

4.4. Overriding and dynamic types

Consider the following program:

```
class C {
    virtual void m() { m(); }
}

class D : C {
    override void m() { throw new System.Exception(); }

    void test() { base.m(); }
}
```

At first sight, it looks like the method `test()` will loop forever. In reality it is not the case: the method `test()` calls the method `m()` from the class `C`, which calls the method `m()` from the class `D`, since the reference '`this`' has the runtime-type `D`. It should be noted that in Java the program behavior can be clearer for a reader, because exceptions thrown from the methods can be expressed explicitly. In particular, in [10] all three methods are accompanied by the note `throws Exception`.

The specifications are as follows:

$$
\begin{array}{lll}
\mathrm{Pre}(\texttt{D.m}) & : & \mathsf{true} \\
\mathrm{Post}(\texttt{D.m}) & : & T(E) = SE
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Pre}(\texttt{D.test}) & : & T(V(\texttt{this})) = D \\
\mathrm{Post}(\texttt{D.test}) & : & T(E) = SE
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Pre}(\texttt{C.m})(mvs, a) & : & prec(C.m, mvs)\wedge \\
 & & T(V(\texttt{this})) = TOT(a) \wedge mvs = mvs_0(a)\wedge \\
 & & (TOT(a) \neq \texttt{C} \Rightarrow \\
 & & \quad \forall f\ \forall mvs'\ ((invoker(f, \texttt{m}, \texttt{this}, [], mvs)\wedge \\
 & & \qquad\qquad\qquad subst(mvs', mvs, f, \texttt{this}, []) \Rightarrow \\
 & & \qquad\qquad\qquad pre(f)(mvs', rest(a))))\wedge \\
 & & (TOT(a) = \texttt{C} \Rightarrow \mathsf{true})
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{Post}(\texttt{C.m})(mvs, a) & : & (TOT(a) \neq \texttt{C} \Rightarrow \\
 & & \qquad \forall f\ (invoker(f, \texttt{m}, \texttt{this}, [], mvs_0(a)) \Rightarrow \\
 & & \qquad\qquad post(f)(mvs, rest(a))))\wedge \\
 & & (TOT(a) = \texttt{C} \Rightarrow \mathsf{false})\ .
\end{array}
$$

Here a stands for the tuple of specification parameters for `C.m()` and $TOT(a)$ denotes the type of `this`. Also $mvs_0(a)$ stores the initial values of meta-variables from mvs. For brevity, we use the name SE instead of `System_Exception`. The postcondition `D.Test()` states that E stores the uncaught exception of type SE. The specification of `C.m()` represents the case analysis. If the type of `this` is `C`, then the invocation `m()` leads to the endless loop ($TOT(a) = \texttt{C} \Rightarrow \mathsf{false}$). Otherwise, the invocation `m()` is the invocation of a proper implementation of `C.m()` from some derived class. The collection of all those implementations is quantified by the variable f which satisfies the predicate *invoker*. This predicate is a logical representation of the late binding algorithm.

The classes `C` and `D` do not contain the fields. As a consequence, the intermediate C#-kernel program is very short and can be represented as follows:

```
class C {
    virtual void m() { this.m(); }
}

class D : C {
    public void IFI_D() {
        Init(C);
        this.IFI_C();
    }

    override void m() {
        Init(System_Exception);
        new_instance();
        L := upd(L, x, V0);
        T := upd(T, L(x), Loc(System_Exception));
        new_instance();
        V := upd(V, L(x), V0);
        T := upd(T, V(x), System_Exception);
        x.IFI_System_Exception();
        x.System_Exception();
        E := V(x);
    }

    void test() { base.m(); }
}
```

Note that the initializing method `IFI` appears in `D` (the corresponding method in `C` turns to be empty). The name spaces are eliminated during translation, so the full name `System.Exception` is replaced by the global level name `System_Exception`.

Two VCs are generated for `D.test()` depending on the fact that the exception is/(is not) raised during the invocation `base.m()`. The VC without exception is tautology, since it has the form $\mathsf{false} \Rightarrow \Phi$. The VC with exception looks like this:

$$CALL(\mathtt{m}, \mathtt{base}, [\,], mvs, \lambda(mvs, P)) \land E \neq \omega \Rightarrow T(E) = SE \ ,$$

where

$$P \equiv prec(\mathtt{D.test}, mvs) \land T(V(\mathtt{this})) = \mathtt{D} \ .$$

The formula $prec()$ is automatically added by the generator for every method to represent some standard assumptions about the class and `this`. However, it does not affect the truth of this VC and we can omit its definition.

The details of refinement of this VC can be found in [20].

4.5. Delegates

A delegate is a class that encapsulates a function signature. In comparison with the C function pointers, the higher level of safety is provided. In the general case, a delegate points at a list of functions, though this feature is rarely used. Let us consider the following program:

```
class Minimum {
    public delegate bool Order(int e1, int e2);

    static public int Find(int[] arr, Order ord) {
        int min = arr[0];
        int xx = 0;

        while(xx < arr.Length){
            if(ord(arr[xx], min)) min = arr[xx];
            xx++;
        }

        return min;
    }
}

class C {
    static bool LessThan(int e1, int e2) { return e1 < e2; }

    static void Main(string[] args) {
        Minimum.Order order = new Minimum.Order(LessThan);
        int[] arr = new int[] {3, 5, 1, 7, 4};
        Minimum.Find(arr, order);
    }
}
```

The class `Mininum` declares the method `Find()`, which should find the minimal element in an array of integers. The method which defines the ordering relation is passed to the method `Find` through the delegate parameter (compare with the C standard library function `qsort`). Here we use the usual "<" relation.

The program specifications are as follows:

$$
\begin{array}{lll}
\mathrm{Pre}(\texttt{Find}) & : & 0 \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
 & & \exists (\mathit{somef} : \texttt{Order}).\ \lambda e_1 e_2.\ V(L(\texttt{ord})) = \lambda e_1 e_2 . \mathit{somef} \\
 & & \\
\mathrm{Post}(\texttt{Find}) & : & \exists j\ (\ 0 \le j \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
 & & \qquad V0 = V(L2(V(L(\texttt{arr})), j)) \wedge \\
 & & \qquad \forall i\ (i \ne j \Rightarrow \mathit{somef}(V0, V(L2(V(L(\texttt{arr})), i))))) \\
 & & \\
\mathrm{Inv}(\texttt{while}) & : & V(L(\texttt{xx})) \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
 & & \forall k\ (\ k \le V(L(\texttt{xx})) \Rightarrow \\
 & & \qquad\quad (\mathit{somef}(V(L(\texttt{min})), V(L2(V(L(\texttt{arr})), k))) \vee \\
 & & \qquad\quad\ V(L(\texttt{min})) = V(L2(V(L(\texttt{arr})), k)))) \\
 & & \\
\mathrm{Pre}(\texttt{LessThan}) & : & \mathsf{true}
\end{array}
$$

$$\text{Post}(\texttt{LessThan}):\ V(L(e_1)) < V(L(e_2)) \Longrightarrow V0 = \mathsf{true}$$

$$\begin{array}{lll}\text{Pre}(\texttt{Main}) & : & \mathsf{true}\\ \text{Post}(\texttt{Main}) & : & V0 = 1\end{array}$$

When we specify the method `Minimum.Find()`, we come across the same problem as for virtual methods. To avoid the limited applicability of `Minimum.Find()`, we should not specify the relation "<" as the only possible parameter. Thus, the delegate parameter is specified by the abstract logical function *somef* without any assumptions about its nature.

Among the dozens of VCs we can consider the following (after simplifications):

$$\left[\begin{array}{l} CALL(\texttt{++}, \texttt{int}, [\texttt{xx}], [L, V_4, T, L2, V0, E], \lambda(mvs, \Phi))\wedge \\ V = upd(V_4, L(\texttt{xx}), V0) \end{array}\right] \Rightarrow INV(\texttt{L}) \quad (6)$$

where

$$\Phi \equiv \left[\begin{array}{ll} DELCALL(V_2(L(\texttt{ord})), [\texttt{x0}, \texttt{min}], mvs, \lambda(mvs, \Phi_2)) & \wedge \\ V_3 = upd(V_2, L(\texttt{b}), V0) & \wedge \\ V_3(L(\texttt{b})) = \mathsf{true} & \wedge \\ V_4 = upd(V_3, L(\texttt{min}), V2(\texttt{arr}, \texttt{xx})) & \end{array}\right]$$

$$\Phi_2 \equiv \left[\begin{array}{ll} newp(d_2, L_2, V_1, L2) \wedge newp(d_1, L_1, V_1, L2) & \wedge \\ V_1(L_1(\texttt{xx})) \le V_1(L2(V_1(L_1(\texttt{arr})), \texttt{Length})) & \wedge \\ \forall k.(k \le V_1(L_1(\texttt{xx})) \Rightarrow & \\ \quad\quad some f(V_1(L_1(\texttt{min})), V_1(L2(V_1(L_1(\texttt{arr})), k)))\vee & \\ \quad\quad V_1(L_1(\texttt{min})) = V_1(L2(V_1(L_1(\texttt{arr})), k)) & \\ ) & \wedge \\ L_2 = upd(L_1, \texttt{b}, d_1) & \wedge \\ L = upd(L_2, \texttt{x0}, d_2) & \wedge \\ V_2 = upd(V_1, L(\texttt{x0}), V2(\texttt{arr}, \texttt{xx})) & \end{array}\right].$$

It corresponds to the body of the `while` loop and contains all variants of lazy terms: *CALL*, *DELCALL* and *INV*. The proof sketch is as follows. The enclosing term *CALL* corresponds to the fragment `xx++`. The assertion $V0 = V(L(arg)) + 1$ can be taken as a postcondition of increment. Note that the side effect of increment is expressed in the second conjunct in (6). During the refinement of *CALL*, the name V is replaced by V_4 and Φ filters outwards. Here, the delegate points at the single function, so the refinement of *DELCALL* resembles the refinement of *CALL*. Thus the adapted delegate specifications are added to Φ_2. Finally, it is easy to establish that $INV(\texttt{L}) = \text{Inv}(\texttt{while})(V0 \leftarrow V(L(\texttt{min})))$. As a result (6) asserts that the loop invariant is preserved, when the counter `xx` increases. The proof is by induction.

5. Related work

Interesting results on semantics formalization for Java and C# are described in [4, 18, 19]. As a rule, these papers either propose a detailed low-level semantics, inconvenient for verification, or verification is the initial goal but the language coverage is poor. The paper [7] deserves a special attention since some hidden controversies of the C# standard [1] were revealed there.

The LOOP project [11] was started in 1997 at the University of Nijmegen. Apart from threads and inner classes, the majority of Java features are covered. The LOOP tool is effectively a compiler, which takes a Java program and its JML specifications as input. As output, it generates several lemmas in the syntax of the theorem prover PVS. Semantics of objects and classes is based on the algebraic approach. Most of the case studies are so-called Java Card programs designed to run on smart cards. The main shortcoming is that the tool works effectively only for small programs.

Another well-known example is the ESC/Java project [13] which supports a wide Java subset. A simple contract language (a subset of JML) is used for specifications. A wp-calculus is used as the semantics description, and a theorem prover searches for potential bugs. However, the full functional correctness was not the aim here, and the developers have deliberately chosen an unsound and incomplete approach to maximize the number of typical bugs that the tool can spot fully automatically.

The Spec# system [3] looks very promising. It fully integrates into Microsoft Visual Studio and .NET Framework, so the complete infrastructure, including libraries, designing and editing tools, is provided. The Spec# language is a superset of C#. It provides user specifications, non-null types and some means for high-level data abstraction. The specifications become a part of a program and can be checked dynamically. The static check by means of the theorem prover Boogie is also supported.

The problem of verification condition complication affects the practicability of a verification system. In [14] Luckham and Suzuki considered the issue of the combinatorial size explosion, when the update function *upd* is used for arrays and pointers. Some algorithms for reducing the recursive *upd* invocations were proposed.

The two-stage VC generation algorithm has been implemented as a part of ESC/Java [6]. The first stage translates a source fragment into an assignment-free, or *passive*, intermediate form. The second stage uses a VC generation technique that is optimized to exploit the assignment-free nature of the passive form. This two-stage algorithm creates a VC, whose size in worst-case quadratic in the size of the source fragment, and in practice appears to be close to linear.

Another approach consists in simplification of VCs before their proof. Gribomont [8] proposed a strategy for the propositional case, corresponding to digital circuits and concurrent synchronization algorithms. Efficiently computable criteria allow one to detect and discard provably irrelevant parts of boolean VCs.

6. Conclusion

In this paper we have described some C# program verification challenges and proposed the solution techniques in the framework of the C#-light project. These techniques form the tree-level approach which extends our two-level approach to C program verification [15]. The advantages of C#-light and of the approach are as follows:

- The C#-light language supports the major part of sequential C#.
- The verification process is based on a simple Hoare-like logic. The simplicity results from translation of semantically difficult C#-light constructs into C#-kernel and postponement of handling some dynamic aspects until the refinement stage.
- Unambiguous inference of lazy verification conditions by means of forward proof rules considerably reduces the number of generated lemmas.

The complexity of verification conditions is not usually proclaimed as a challenge. However, our detailed memory model can lead to the combinatorial explosion in the length of terms. Therefore we proposed some strategies which considerably simplify the verification conditions.

The three-level approach is promising for applications. We are developing an experimental tool for C#-light program verification including C#-light to C#-kernel translator, lazy verification condition generator as well as lazy verification condition refiner. It is supposed to implement a static analyzer to check applicability of simplification strategies.

Our list of challenges is not complete. The problems such as termination, class invariance, low-level arithmetics, as well as some advanced features mentioned in [12], can form the framework of future research.

References

[1] C# Language Specification. Standard ECMA-334. — 2001.

[2] Apt K.R., Olderog E.R. Verification of sequential and concurrent programs. — Berlin etc.: Springer, 1991. — 450 p.

[3] Barnett M., Leino K.R.M., Schulte W. The Spec# Programming System: An Overview // Lect. Notes Comput. Sci. — 2004. — Vol. 3362. — P. 49–69.

[4] Börger E., Fruja N.G., Gervasi V., Stärk R. A High-Level Modular Definition of Semantics of C# // Theor. Comput. Sci. — 2004. — Vol 336, N 2/3.

[5] Dubranovsky I.V. C# program verification: the translation from C#-light into C#-kernel. — Novosibirsk, 2006. — 56 p. — (Prep. / IIS SB RAS; N 140).

[6] Flanagan C., Saxe J.B. Avoiding Exponential Explosion: Generating Compact Verification Conditions // ACM Press, January 2001. — P. 193–205.

[7] Fruja N.G. Specification and Implementation Problems for C# // Lect. Notes Comput. Sci. — 2004. — Vol. 3052. — P. 127–143.

[8] Gribomont P.E. Simplification of Boolean Verification Conditions // Theor. Comput. Sci. — 2000. — Vol. 239, N 1. — P. 165–185.

[9] Hoare C.A.R. An Axiomatic Basis for Computer Programming // Communs. ACM. — 1969. — Vol. 12, N 1. — P. 576–580.

[10] Jacobs B., Kiniry J.L., Warnier M. Java Program Verification Challenges // Lect. Notes Comput. Sci. — 2003. — Vol. 2852. — P. 202–219.

[11] Jacobs B., Poll E. Java Program Verification at Nijmegen: Development and Perspective // Lect. Notes Comput. Sci. — 2004. — Vol. 3233. — P. 134–153.

[12] Leavens G.T., Leino K.R.M., Müller P. Specification and verification challenges for sequential object-oriented programs // TR#06–14a, Dept. of Computer Science, Iowa State University, 2006.

[13] Leino K.R.M. Extended Static Checking: a Ten-Year Perspective // Lect. Notes Comput. Sci. — 2001. — Vol. 2000. — P. 157–175.

[14] Luckham D.C., Suzuki N. Verification of Array, Record and Pointer Operations in Pascal // ACM Trans. Progr. Lang., and Systems. — 1979. — Vol. 1, N 2. — P. 226–244.

[15] Nepomniaschy V.A., Anureev I.S., Mihailov I.N., Promsky A.V. Verification-Oriented Language C-light // System informatics. — Novosibirsk, 2004. — Vol. 9: Formal methods and models of informatics Proc. — P. 51–134.

[16] Nepomniaschy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V. Towards C# Program Verification: A Three-Level Approach // Programming and Computer Software. — 2007. — N 4. — P. 1–16. (English translation of Russian Journal "Programmirovanie".

[17] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards C# Program Verification: C#-Kernel and Its Axiomatic Semantics // Proc. CS&P'2006. — Humboldt University, Berlin, 2006. — Vol. 2: Specification. — P. 195-206.

[18] Oheimb D.v. Hoare Logic for Java in Isabelle/HOL // Concurrency and Computation. — 2001. — Vol. 13.

[19] Poetzsch-Heffter A., Muller P. A Programming Logic for Sequential Java // Lect. Notes Comput. Sci. — 1999. — Vol. 1576. — P. 162–176.

[20] Promsky A.V. Application of three-level approach to C#-light program verification. — Novosibirsk, 2006. — (Prepr. / IIS SB RAS; N 139).