Joint NCC & IIS Bull., Comp. Science, 20 (2004), 61–85

A three-level approach to C# program verification*

V. A. Nepomniaschy, I. S. Anureev,
I. V. Dubranovsky, A. V. Promsky

Abstract. A new three-level approach to sequential object-oriented program verification is presented. It is applied to a significant C# subset called C#-light that includes all principal sequential C# constructs. At the first stage, C#-light is translated into an intermediate language C#-kernel that allows us to simplify axiomatic semantics. At the second stage, lazy verification conditions are generated by means of backward rules of axiomatic C#-kernel semantics. Lazy verification conditions can include special functional symbols which are refined at the third stage. An example of verification of a C# light program serves to illustrate this three level approach.

1. Introduction

Verification of programs presented in widely-used object-oriented programming languages, such as C++, C#, Java, is a subject of much current interest. An essential prerequisite for a programming language to be suitable for verification is compact transparent formal semantics. The most extensively employed approach to formalization of semantics is the operational one using such notions as transition systems and abstract machines. For example, formal operational semantics has been developed for Java [2]. However, the verification process in operational semantics is, as a rule, much more complicated as compared with axiomatic semantics based on Hoare-like logics.

Difficulties of developing compact and transparent axiomatic semantics of object-oriented programming languages are connected with such constructs as overloading, dynamic binding of methods, exception handling, static initialization of classes. Axiomatic semantics has been proposed for different sequential Java subsets in [5, 6, 11, 12, 13, 14]. However, compact and transparent axiomatic semantics has been developed for separate difficult Java constructs, whereas it turned out to be cumbersome and inconvenient for the practical use in the case of a wide sequential Java subset [11].

We introduce a wide sequential C# subset called C#-light, and develop a new three-level verification approach combining operational and axiomatic

*This research was partially supported by Microsoft Research in 2002 and is partially supported by grant 04-01-00114a from RFBR.

semantics, to C#-light program verification. At the first stage, C#-light is translated into an intermediate language C#-kernel in order to eliminate some C#-light constructs difficult for axiomatic semantics such as, for example, the `try` statement, as well as to design axiomatic semantics in more compact and transparent form. At the second stage, lazy verification conditions are generated by means of backward rules of axiomatic C#-kernel semantics. These verification conditions are lazy because they can include special functional symbols which are refined at the third stage.

The paper consists of 9 sections. The language of program annotations is described in Section 2. The C#-light language is described in Section 3. The C#-kernel language is defined in Section 4. A method for translation of C#-light into C#-kernel is considered in Section 5. Axiomatic semantics of C#-kernel is described in Section 6. A verification condition refinement technique is considered in Section 7. An illustrative example of verification of a C#-light program is presented in Section 8. The results and perspectives of development of our approach are discussed in Section 9.

2. Annotation language

Let us introduce an *annotation language* which is used for description of program annotations (pre-, postconditions and invariants).

Types and alphabet. The admissible types of the annotation language are as follows:

- base: `CT#`, `Names`, `Locations`, `TypeSpecs`
- functions: $\mathtt{T}_i \rightarrow \mathtt{T}$
- Cartesian products: $\mathtt{T}_1 \times \mathtt{T}_2$

where `CT#` is the union of all admissible C#-light types, `Names` is the set of object identifiers in the program, `Locations` is the set of storage locations, `TypeSpecs` is the set of abstract names of types. The type `Location` includes the special storage locations `Val` and `Exc` that are used for storing the value of the last evaluated (sub)expression and raised exception, respectively.

The alphabet of the annotation language consists of the following symbol classes:

- *variables* and *constants*;
- *functional* and *predicate* symbols;
- *parentheses* and comma;
- *operation* symbols: standard T#-light operators and $\cup$.

Expressions. The variables can only be of the admissible types. The constants can be of every type except `void`. In addition to usual values, the base C# types can contain undefined value denoted as `Om`.

The expressions of the annotation language are defined by induction:

- a variable `v` of type `T` is an expression of type `T`;
- a constant `c` of type `T` is an expression of type `T`;
- if s_1, ..., s_n are expressions of types T_1, ..., T_n, respectively, and `s` is an expression of type $T_1 \times ... \times T_n \to T$, then $s(s_1, ..., s_n)$ is an expression of type `T`.

Logical expressions, or simply *annotations*, are built from expressions of the `bool` type with the help of logical connectives and quantifiers in a usual way. Since the expressions have a prefix syntax, the standard logical connectives are denoted as the following function-constants: `and(,)`, `or(,)`, `not()`, `imply(,)`, `exist(,)`, `any(,)`. The equality predicate is denoted by `eq(,)`. It should be noted that C#-light operators are also written in the prefix form. For example, the expressions `a[3]` and `x+3` will be written as `[ ](a, 3)` and `+(x, 3)`, respectively. For reading convenience, in some cases we prefer the usual syntax.

Among the function names we distinguish the name upd with the following fixed interpretation. If `s` is an expression of type $T \to T'$, and expressions e_1, e_2 are of types `T` and `T'`, respectively, then the term `upd(s,` e_1, e_2) is an expression s' of type $T \to T'$, which agrees with `s` on all arguments except, maybe, e_1 and $s'(e_1) = e_2$.

Let the term $s(u \leftarrow t)$ denote the substitution of the expression `t` for all free occurrences of the variable `u` in the expression `s`.

Interpretation of types. First, we fix, for each type `T`, a set of values called the *domain* of `T` and denoted by D_T. The boundaries of the base type domains (i.e., the types from `CT#`) are denoted by constants from the standard class definitions.

- $D_{bool} = \{$`false, true`$\}$;
- $D_{byte} = \{$`System.byte.MinValue ... System.byte.MaxValue`$\}$;

  ...
- $D_{void} = \emptyset$;

  ...
- $D_{T_1 \times ... \times T_n \to T} = D_{T_1} \times ... \times D_{T_n} \to D_T$, the set of all functions from the Cartesian product of the sets D_{T_1}, ..., D_{T_n} into the set D_T;
- D_{Names} is the set of all identifiers allowed by C# syntax;
- $D_{Locations}$ is the set of uninterpreted constants (storage locations);

- $\mathtt{D}_{\mathtt{TypeSpecs}}$ is the set of abstract type names allowed by C# syntax.

The semantic domain is defined as a disjoint union: $\mathtt{D} = \bigcup_{\mathtt{T}} \mathtt{D}_{\mathtt{T}}$.

Metavariables and states. The definition of a state shows a distinction of our approach from the classical one [1]. In that approach, the names of program variables were identified with variables in the annotation language. We stipulate that the variable names are just constants in the set $\mathtt{D}_{\mathtt{Names}}$. The modifications of program variables are handled via the following metavariables:

1. MeM is a variable of the type Names ∪ CT# → Locations ∪ CT#, i.e. it is **Me**mory **M**anagement;
2. MD is a variable of the type Locations ∪ CT# → CT#. This is abstract **Me**mory **D**ump;
3. TP is a variable of the type Names ∪ Locations ∪ CT# → TypeSpecs. It defines the types of identifiers, storage locations and constants of the C# light types.

For example, the fact that a program variable x is equal to 3 in the classical approach can be presented as an annotation eq(x, 3) and in our approach it is represented by eq(MD(MeM(x)), 3), where x is already a constant.

A function from metavariables to the semantic domain D is called a *state*. Let States denote a set of all states.

Interpretation of expressions. The *semantics* $\mathcal{I}[\![\mathtt{s}]\!]$ of an expression s of type T is a mapping

$$\mathcal{I}[\![\mathtt{s}]\!] : \mathtt{States} \to \mathtt{D}_{\mathtt{T}}$$

defined by induction on the structure of s:

- if s is a variable, then $\mathcal{I}[\![\mathtt{s}]\!](\sigma) = \sigma(\mathtt{s})$;
- if s is a constant denoting the value d, then $\mathcal{I}[\![\mathtt{s}]\!](\sigma) = \mathtt{d}$;
- if $\mathtt{s} \equiv \mathtt{op}(\mathtt{s}_1, \ ..., \ \mathtt{s}_n)$ for some expression op, then

$$\mathcal{I}[\![\mathtt{s}]\!](\sigma) = \mathcal{I}\|\mathtt{op}\|(\sigma)(\mathcal{I}[\![\mathtt{s}_1]\!](\sigma), ..., \mathcal{I}[\![\mathtt{s}_n]\!](\sigma)).$$

Using the standard method [1], we define the *truth* of an annotation p in a state σ, written as $\sigma \models \mathtt{p}$. We also introduce the *meaning* of an annotation defined by

$$\|\mathtt{p}\| = \{\sigma |\ \sigma \text{ is a state and } \ \sigma \models \mathtt{p}\}.$$

3. The C#-light language

The C#-light language is a significant sequential subset of the C# language. Let us define C#-light programs and annotated C#-light programs.

C#-light programs. C#-light programs include all C# constructs except attributes, destructors, lock and using statements, checked and unchecked constructs (both expressions and statements), unsafe code and preprocessor directives.

Annotated C#-light programs. Comments of a C#-light program `Prg` of the form

```
/// <annotation> Annotation </annotation>
```

are called annotations of `Prg`. A program which has annotations is called an annotated program.

Let us introduce some definitions.

Function components are program parts that contain executable statements. They include function members [4], standard library operators, and accessors of properties, indexers and events. Each function component has a unique name called a function component identifier that points to it.

Let `M` be a function component, `f` be a function component identifier which points to `M`.

An annotation `P` is called a precondition of `M` if the body of `M` starts with `P`. An annotation `Q` is called a postcondition of `M` if the body of `M` finishes with `Q`. The pair `(P, Q)`, where `P` and `Q` are pre- and postconditions of `M`, is called a specification of `M`.

An access to the structure of function components through their identifiers is provided by the functions `class`, `body`, `pre` and `post` such that `class(f)` is the name of the class containing `M`, `body(f)` is the body of `M`, `SP(f)` is the tuple of specification parameters of `M`, `pre(f)` and `post(f)` are functions such that `pre(f)(mvs, SP(f))` and `post(f)(mvs, SP(f))` are pre- and postconditions of `M`, respectively. Here and below, `mvs` denotes the tuple `[MeM, MD, TP]` of metavariables.

4. The C#-kernel language

The C#-kernel language is an imperative object-oriented language that is based on the syntax of C#-light. First, a C#-light subset `S` is built by the following restrictions:

- `S` does not contain namespaces and using-directives.
- `S` does not contain the following statements:

  - the jump statements `break`, `continue`, `return`, `goto case`, `goto default` and `throw`;
  - the `try` statement;
  - the selection statement `switch`;
  - all iteration statements;
  - declaration statements.

- `S` does not contain `if` statements, in which the boolean expression is not a boolean variable.
- An invocation of a static function member is located only in the places where the static initialization of an appropriate class or struct has already been performed.
- All labels, local variable names and local constant names must be unique.
- The sets of labels, local variable names, local constant names and type names are disjoint.
- All comments are annotations.

Second, C#-kernel extends the subset `S` by metainstructions, modified expression statements, and modified class and struct declarations.

4.1. The metainstructions

Metainstructions are used to handle metavariables. There are five metainstructions:

1. `x := e` assigns the expression `e` in the annotation language to the metavariable `x`.
2. `new_instance(x)` associates the identifier `x` with a new storage location.
3. `Init(C)` performs static initialization of the class `C`, if this class has not yet been initialized, and does nothing, otherwise.
4. `catch(T, x)` returns `true` if the storage location `Exc` stores a value of the type `T`, `false` otherwise. If the return value is `true`, the current exception object located in `Exc` is written to the variable `x` and is deleted from `Exc` to indicate that the exception object has been caught and no further exception handling is needed. This metainstruction occurs only as a conditional expression within an `if` statement.

5. `catch(x)` returns `true` if the value of the storage location `Exc` is defined, `false` otherwise. Like `catch(T, x)`, the current exception object is written to the variable `x` and is deleted from `Exc`. This metainstruction occurs only as a conditional expression within an `if` statement without the `else` branch and models a general catch clause of the `try` statement.

4.2. The expression statement

The expression statement in C#-kernel is a *normalized expression* or metainstruction followed by a semicolon. *Normalized expressions* are defined by the following restrictions to C#-light expressions:

- A normalized expression has the form $\mathtt{x.y(z_1, ..., z_n)}$ or $\mathtt{y(z_1, ..., z_n)}$, where `x`, `y` are names, $\mathtt{z_1}, \ldots, \mathtt{z_n}$ are names or literals.
- In normalized expressions, function members can be invoked only in their normal form [4].
- Logical operators `||` and `&&`, the conditional operator `?:`, the operator `new` and assignment operators are not permitted in normalized expressions.

4.3. The class and struct declaration

Declarations of fields and constants of classes and structs does not contain initializers. Instead, two methods `SFI` and `IFI` called initializing methods are reserved for each class declaration to perform static and instance field initialization, respectively. Initialization of constants and static fields takes place in the method

```
public static void SFI() {
    <constant-default-initialization>
    <static-field-default-initialization>
    <constant-initialization>
    <static-field-initialization>}
```

where each static field and constant is initially set to the default value of its type with the following optional initialization of fields and mandatory initialization of constants. Initialization of instance fields takes place in the method

```
public void IFI() {
    <instance-field-default-initialization>
    <instance-field-initialization>}
```

These methods extend the context for direct assignments to a `readonly` field.

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

Translation from C#-light into C#-kernel is a sequential application of translation algorithms. Translation algorithms are constructed on the basis of transformations. Certain algorithms are defined by a set of transformations which are non-deterministically applied to the program. Other algorithms have an imperative form, and transformations are used as elementary actions when defining these algorithms. Each transformation specifies a program fragment to be transformed and conditions under which the transformation can be applied.

All algorithms are split into two classes:

- normalization algorithms;
- elimination algorithms.

Normalization algorithms prepare constructs of the translated language for elimination algorithm application. Some of them modify C#-light constructs to conform to C#-kernel syntax. Elimination algorithms replace C#-light constructs that are prohibited in C#-kernel by those that are permitted in C#-kernel.

Let us sketch the main ideas of the translation algorithm from C#-light into C#-kernel.

5.1. Expression normalization

An invocation of a function member in the expanded form is replaced by an invocation of the function member in the normal form. To achieve this, in a given invocation expression, arguments that correspond to the parameter array are replaced by an array creation expression that creates an array of these arguments.

The boolean expression in the `if` statement is placed to a separate expression statement.

The requirement that an invocation of a static function member is located only in the places where the static initialization of the appropriate class or struct has already been performed is achieved by inserting the appropriate metainstruction `Init`.

The conditional operator `?:` and the logical operators `||` and `&&` are expressed via the `if` statement. The assignment operators are expressed via the metainstruction `:=`. The operator `new` is expressed via the metainstructions `:=` and `new_instance` and the `IFI` method.

The property, indexer, event and operator declarations are duplicated in the method declarations according to the reserved member names [4]. The property, indexer and event access, event assignment and operator invocation are transformed into the invocations of appropriate methods.

The field and array access, as well as field and array updates, are expressed via the metainstruction `:=`.

The constant and field initializers are moved to the `SFI` and `IFI` methods according to the definition of C#-kernel.

The detailed description of Norm transformation that normalizes C#-light expression statements is given in [3]. This transformation is defined using C# grammar rules for expression derivation. For each kind of expressions, Norm specifies how to split the expression into several normalized expression statements. This specification can also include calls to Norm, what makes the transformation recursive.

5.2. Statement elimination

The declaration statements are rewritten to the metainstructions `:=` and `new_instance`.

The jump statements `break`, `continue` and `return` are replaced by the `goto` statement. In this case, when we eliminate the statement `return e;` contained by a function member, the metainstruction `:=` and the special storage location `Val` are used to set the return value of the function member.

The `try` statement is replaced by several other statements. The `try` block is replaced by its content; the `catch` blocks are replaced by nested `if` statements with boolean conditions of the form `catch(T,x)` or `catch(x)`.

The statements `goto`, `goto case` and `goto default` are replaced by the `goto` statement so that all labels become unique. When we eliminate the statement `throw e;`, the metainstruction `:=` is used to raise the exception produced by the `e` expression. The exception is stored in the storage location `Val`. In this case, the special type `Exc` of `Val` signals about exception raising.

Iteration statements are transformed into the `if` and `goto` statements.

The selection statement `switch` is expressed via the conditional statement `if`. In fact, each `switch`-section is transformed into an `if` statement in which the `else` branch contains all subsequent `switch` sections. In a translated program, an innermost `else` branch will correspond to a `default` section.

5.3. Using directive and namespace elimination

The `using` directives are eliminated so that all names are replaced by their appropriate qualified names and the `using` directives are removed.

The namespace elimination is performed as follows. All qualified names are replaced with their fully qualified names. Further, all types are placed to the global namespace. In this case, the type names are changed by simple type names that are unique in the global namespace. Finally, all references to objects of these types are directed to the global namespace.

6. Axiomatic semantics of C#-kernel

The basic formulas of our Hoare-like axiomatic logic are constructs of the form $\langle P \rangle A \quad S \langle Q \rangle$ (called *Hoare's triples*), where A is a statement sequence (called *context*), S is a statement, P and Q are annotations called precondition and postcondition, respectively.

6.1. General rules

The starting rule. The starting rule reduces the proof of an annotated C#-kernel program `Prg` to the proof of bodies of its function members in the context called a proof environment. The proof environment `Env` is a pair `(Prg, f)`, where `f` is a function component identifier.

Let $\mathtt{f_i}$, where $\mathtt{1 \le i \le n}$, be a set of all function component identifiers pointing to constructors and methods of a program `Prg` except for methods `SFI` and `IFI`[1]. Let us remind that the property, indexer, event and operator declarations are duplicated in the method declarations at the stage of translation. Let $\mathtt{Env_i}$, $\mathtt{P_i}$ and $\mathtt{Q_i}$ denote $(\mathtt{Prg},\ \mathtt{f_i})$, $\mathtt{pre(f_i)(mvs,\ SP(f_i))}$ and $\mathtt{post(f_i)(mvs,\ SP(f_i))}$, respectively.

The rule has the form

$$\frac{\mathtt{Env_i} \vdash \langle\, \mathtt{P_i} \,\rangle\, \mathtt{Init(class(f_i))};\ \mathtt{body(f_i)}\, \langle\, \mathtt{Q_i} \,\rangle \text{ where } \mathtt{1 \le i \le n} \qquad \mathtt{Env_j} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{Init(class(f_j))}\, \langle\, \mathtt{Q'} \,\rangle}{\langle\, \mathtt{P} \,\rangle\, \mathtt{Prg}\, \langle\, \mathtt{Q} \,\rangle}\ , \qquad (1)$$

where $\mathtt{f_j}$ points to the entry point method of the program `Prg`, `entry_point_args` is a tuple of arguments of this method,

$$\mathtt{Q'} = \mathtt{CALL(Main,\ class(f_j),\ entry\_point\_args,\ mvs,\ Q)}.$$

Here `CALL` is a logical function such that the expression $\mathtt{CALL(x, y, [z], u, Q)}$ called a *lazy invocation* is true in the state σ if execution of the invocation expression `y.x(z)` in the state σ', such that $\sigma'(v) = \mathcal{I}[\![u]\!](\sigma)$ for each metavariable `v`, results in the state in which the formula `Q` is true, and the precondition of the invoked function member is true in the state σ'. Lazy invocations are specified at the refinement stage.

The exception propagation rule. We use the following rule scheme for exception propagation:

[1] We do not verify these function members because they appear at the stage of translation from C#-light to C#-kernel and there are no specifications.

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} \neq \mathtt{Om?}\, \langle\, \mathtt{Q} \,\rangle \\ \mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{TR_1(Q,\ S)} \\ \ldots \\ \mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{TR_n(Q,\ S)}\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{S}\, \langle\, \mathtt{Q} \,\rangle} \tag{2}$$

where S is a C#-kernel statement. The storage location Exc stores an unhandled exception. The fragments $\mathtt{TR_1(Q,S)}, ..., \mathtt{TR_n(Q,S)}$ are defined according to the semantics of the statement S. They have the form $\mathtt{B}\ \langle \mathtt{Q'} \rangle$, where B is a sequence of the C#-kernel constructs and intermediate forms generated by the rules of axiomatic semantics and Q′ is a postcondition. For simplicity, we omit the first branch in the scheme instances marked with (*).

The intermediate form R? called guard is defined by the rule:

$$\frac{\mathtt{Env} \mid\ \langle\, \mathtt{P} \,\rangle\, \mathtt{A}\, \langle \mathtt{R} \rightarrow \mathtt{Q} \rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{R?}\, \langle\, \mathtt{Q} \,\rangle}\ . \tag{3}$$

6.2. Statements

The if statement.

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{MD(MeM(x))} = \mathtt{true?} \quad \mathtt{S_1}\, \langle\, \mathtt{Q} \,\rangle \\ \mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{MD(MeM(x))} = \mathtt{false?} \quad \mathtt{S_2}\, \langle\, \mathtt{Q} \,\rangle\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{if\ (x)\ S_1\ else\ S_2}\, \langle\, \mathtt{Q} \,\rangle}\ (*). \tag{4}$$

The block statement.

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{S_1} \ldots \mathtt{S_n}\, \langle\, \mathtt{Q} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \{\mathtt{S_1} \ldots \mathtt{S_n}\}\, \langle\, \mathtt{Q} \,\rangle}\ (*). \tag{5}$$

The goto and labelled statements.

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{INV(mvs,\ L)} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{goto\ L;}\, \langle\, \mathtt{Q} \,\rangle}\ (*). \tag{6}$$

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A}\, \langle\, \mathtt{INV(mvs,\ L)} \,\rangle \\ \mathtt{Env} \vdash \langle\, \mathtt{INV(mvs,\ L)} \,\rangle\, \mathtt{S}\, \langle\, \mathtt{Q} \,\rangle\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{L:\ S}\, \langle\, \mathtt{Q} \,\rangle}\ . \tag{7}$$

We use the expression INV(mvs, L) called a **lazy invariant** of the label L because the additional labels that appear after the translation from C#-light to C#-kernel is not associated with invariants. The lazy invariant is replaced by the invariant at the refinement stage.

The expression statement. Expression statements are defined by the four rules.

The rule of an invocation of a function member has the form:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{CALL(x,\ y,\ [z],\ mvs,\ Q)} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{y.x(z);}\, \langle\, \mathtt{Q} \,\rangle} \ (*). \tag{8}$$

if y is a function member name and y.x(z) is not an invocation of an initializing method.

The initializing methods SFI and IFI have their own rules replacing their invocations by their bodies:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{LocValRen(S)}\, \langle\, \mathtt{Q} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{T.SFI();}\, \langle\, \mathtt{Q} \,\rangle} \ (*), \tag{9}$$

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{LocVal\&ThisRen(S)}\, \langle\, \mathtt{Q} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{o.IFI();}\, \langle\, \mathtt{Q} \,\rangle} \ (*), \tag{10}$$

Here S is the body of the invoked method, the function LocValRen substitutes the fresh names for all local variables in body(f), the function LocVal&ThisRen, in addition to local variables renaming, also substitutes o for the name this.

The rule of an invocation of a delegate has the form:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{DELCALL(MD(MeM(x)),\ [z],\ mvs,\ Q)} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{x(z);}\, \langle\, \mathtt{Q} \,\rangle} \ (*), \tag{11}$$

if x is a delegate name. The function DELCALL is defined in the refinement stage.

Annotations.

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A}\, \langle\, \mathtt{R} \,\rangle \qquad \mathtt{Env} \models \mathtt{R} \Rightarrow \mathtt{Q}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{///\ <annotation>\ R\ </annotation>}\, \langle\, \mathtt{Q} \,\rangle}\ . \tag{12}$$

The empty statement and empty program.

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A}\, \langle\, \mathtt{Q} \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{;}\, \langle\, \mathtt{Q} \,\rangle} \qquad \frac{\mathtt{Env} \models \mathtt{P} \Rightarrow \mathtt{Q}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \langle\, \mathtt{Q} \,\rangle}\ . \tag{13}$$

6.3. Metainstructions

Let τ denote a substitution

$$(\mathtt{MeM} \leftarrow \mathtt{upd(MeM, x, MD(Exc))},\ \mathtt{MD} \leftarrow \mathtt{upd(MD, Exc, Om)},\ \mathtt{TP} \leftarrow \mathtt{upd(TP, Exc, Om)}).$$

The metainstruction catch(T,x) has the following semantics:

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad (\mathtt{MD(Exc)} \neq \mathtt{Om} \wedge \mathtt{TP(Exc)} \subseteq \mathtt{T})?\quad \tau!\quad \mathtt{S}_1\, \langle\, \mathtt{Q} \,\rangle \\ \mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad (\mathtt{MD(Exc)} = \mathtt{Om} \vee \mathtt{TP(Exc)} \not\subseteq \mathtt{T})?\quad \mathtt{S}_2\, \langle\, \mathtt{Q} \,\rangle\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{if(catch(T,\ x))\ S_1\ else\ S_2}\, \langle\, \mathtt{Q} \,\rangle}\ , \tag{14}$$

where $\subseteq$ denotes the type inheritance relation. The intermediate form $\tau!$ is defined by the rule:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A}\, \langle\, \mathtt{Q}\tau \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \tau!\, \langle\, \mathtt{Q} \,\rangle} \ . \tag{15}$$

The metainstruction `catch(x)` has the following semantics:

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} \neq \mathtt{Om?} \quad \tau! \quad \mathtt{S}\, \langle\, \mathtt{Q} \,\rangle \\ \mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{Q} \,\rangle\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{if(catch(x))}\, \mathtt{S}\, \langle\, \mathtt{Q} \,\rangle} \ . \tag{16}$$

The metainstruction $\mathtt{X}$ `:=` $\mathtt{E}$ is defined by the rule:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{Q}(\mathtt{X} \leftarrow \mathtt{E}) \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{X} := \mathtt{E};\, \langle\, \mathtt{Q} \,\rangle}\ (*), \tag{17}$$

if $\mathtt{X}$ is a metavariable, $\mathtt{E}$ is an expression in the annotation language.

The metainstruction `Init(C)` has the following semantics:

$$\frac{\begin{array}{c}\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{MeM(C)} \neq \mathtt{Om?}\, \langle\, \mathtt{Q} \,\rangle \\ \mathtt{Env} \ \vdash \ \langle \mathtt{P} \rangle \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?} \quad \mathtt{MeM(C)} = \mathtt{Om?} \quad \tau! \quad \mathtt{C.SFI();} \\ \mathtt{C.C()};\, \langle \mathtt{Q} \rangle\end{array}}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{Init(C)};\, \langle\, \mathtt{Q} \,\rangle}\ (*), \tag{18}$$

where τ denotes a substitution $(\mathtt{MeM} \leftarrow \mathtt{upd}(\mathtt{MeM}', \ \mathtt{C}, \ \mathtt{nc}))$ and `nc` is a new location. The static part of the class `C` is treated as an object in the memory which is accessed via the name `C`.

The metainstruction `new_instance(x)` is defined by the rule:

$$\frac{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{MD(Exc)} = \mathtt{Om?}\, \langle\, \mathtt{Q}(\mathtt{MeM} \leftarrow \mathtt{upd}(\mathtt{MeM}, \ \mathtt{x}, \ \mathtt{nc})) \,\rangle}{\mathtt{Env} \vdash \langle\, \mathtt{P} \,\rangle\, \mathtt{A} \quad \mathtt{A\ new\_instance(x)};\, \langle\, \mathtt{Q} \,\rangle}\ (*), \tag{19}$$

where `nc` is a new location.

7. Verification condition refinement

At the refinement stage, the invariants `INV(...)` of labelled statements are refined and the logical functions `CALL(...)` and `DELCALL(...)` are defined according to the invocation rules [4] for function members and delegates, respectively.

Refinement of lazy invariants. For formulas `A` and `B`, let $\mathtt{F_{I.}(A)}$ denote

$$\mathtt{INV(mvs, L)} \Rightarrow \mathtt{A}.$$

For a set `X` of formulas, let $\mathtt{I_L(X)}$ denote $\{\mathtt{A} | \mathtt{F_L(A)} \in \mathtt{X}\}$, $\mathtt{X_L}$ denote

$$\{\mathtt{F_L(A)}|\mathtt{A} \in \mathtt{I_L(X)}\}.$$

The algorithm of refinement of lazy invariants `INV(...)` in the set of lazy verification conditions Φ generated is as follows:

> While there exists a label L such that Φ_L is not empty, replace Φ by $(\Phi \setminus \Phi_L)\tau$ where the substitution τ has the form:
>
> $$\{\mathtt{INV}(\bar{e}, \mathtt{L}) \leftarrow (\wedge_{\mathtt{A} \in \mathtt{I_L}(\Phi)} \mathtt{F_L(A)}(\mathtt{mvs} \leftarrow \bar{e}))|\mathtt{INV}(\bar{e}, \mathtt{L}) \in (\Phi \setminus \Phi_L)\}.$$

Refinement of lazy invocations. The lazy invocations are refined during the proof of verification conditions according to the following axioms, which define the functions `CALL` and `DELCALL`:

$$\begin{aligned}
\mathtt{ENV} \models\ & \mathtt{FCN(x)} \Rightarrow \\
& (\mathtt{CALL(x,\ y,\ z,\ u,\ Q)} \Leftrightarrow \\
& \forall \mathtt{f}\ (\mathtt{FCI(f)} \wedge \mathtt{invoker(f,\ x,\ y,\ z,\ u)} \Rightarrow \\
& \quad \exists \mathtt{a}\ (\mathtt{pre(f)(subst(u,\ f,\ y,\ z),\ a)} \wedge \\
& \quad\quad \forall \mathtt{v}\ (\mathtt{post(f)(v,\ a)} \Rightarrow \mathtt{Q(mvs} \leftarrow \mathtt{v)})))))
\end{aligned}$$

$$\begin{aligned}
\mathtt{ENV} \models\ & \mathtt{DELCALL(cons([x1,\ y1],\ x),\ z,\ u,\ Q)} \Leftrightarrow \\
& \mathtt{CALL(x1,\ y1,\ z,\ u,\ DELCALL(x,\ z,\ u,\ Q))}
\end{aligned}$$

$$\mathtt{ENV} \models \mathtt{DELCALL([\,],\ z,\ u,\ Q)} \Leftrightarrow \mathtt{TRUE}$$

The logical function `FCI` returns `true` if its argument is a function component identifier. The function `subst` modifies metavariables in `P`, performing argument substitution and returning the tuple of values of modified metavariables. The function `invoker` checks whether the function member `f` satisfies the invocation with parameters `x`, `y`, `z`, `u`, `v`. These functions are axiomatized in accordance with the specification [4].

The value of a delegate is a list of pairs of the form `[a,b]`, where `a` is the name of a function member invoked and `b` is an object or type on which it is invoked, with standard operations `car`, `cdr`, `cons` and `empty`.

For the function member identifer `f` pointing to the methods `SFI` and `IFI`, the function `pre(f)` is defined as follows:

$$\begin{aligned}
& \mathtt{pre(f)([MeM,\ MD,\ TP],\ [MeM_0,\ MD_0,\ TP_0])} = \\
& \mathtt{and(MeM} = \mathtt{MeM_0,\ MD} = \mathtt{MD_0,\ TP} = \mathtt{TP_0,\ MD_0(Exc)} = \mathtt{Om}, \\
& \mathtt{not(MeM_0(class(f))} = \mathtt{Om)).}
\end{aligned}$$

The function `post(f)` is generated automatically as a result of applying the rules of axiomatic semantics to `body(f)`.

8. Illustrative example

Let us see how the three-level verification approach can be used. The program defines two classes. The `IdManager` class provides the means of generating unique identifiers. For simplicity, these 'identifiers' are values of the `byte` type. The `Example` class defines the `Main` method that uses the static member of the `IdManager` class to initialize the array `ids`. The algorithm is intended to raise an exception when it tries to initialize the last element of `ids`. The annotated C#-light program is as follows:

```
/// <annotation>  entry_points_args = [ ]  </annotation>

using System;

public class IdManager {
    private IdManager() {
        /// <annotation>  P  </annotation>
        /// <annotation>
        ///    and(MeM = MeM0, MD = MD0, TP = TP0)
        /// </annotation>
    }
    private static IdManager instance = new IdManager();
    public static IdManager Instance {
        get{/// <annotation>  P  </annotation>
            return instance;
            /// <annotation>  Q:getIdManager  </annotation>
        }
    }

    private byte last_id = 0;

    public byte Id{
        get{/// <annotation>  P  </annotation>

            if((last_id+1) > byte.MaxValue){
                throw new Exception("Error");}
            else return last_id++;

            /// <annotation>  Q:getId  </annotation>
        }
    }
}

public class Example {
```

```
    public static void Main() {
        /// <annotation>  P  </annotation>

        const int MAX_SIZE = byte.MaxValue + 1;
        int[] ids = new int[MAX_SIZE];
        try{
            for(int i=0; i<MAX_SIZE; i++) {
                /// <annotation>  INV  </annotation>
                ids[i] = IdManager.Instance.Id;
            }
        }
        catch {}
        /// <annotation>  Q:Main  </annotation>
    }
}
/// <annotation>  true  </annotation>
```

where the annotations have the form:

```
P:

and(MeM = MeM0, MD = MD0, TP = TP0, MD0(Exc) = 0m)

Q:getInstance:

and(MeM = MeM0,
 MD = upd(MD0, Val, member(MD0(MeM0(this)), instance)),
 TP = TP0)

Q:getId:

and(
 imply(
   member(MD0(MeM0(this)), last_id) + 1 >
    member(MD(MeM(byte)),MaxValue),
   and(MeM = MeM0, MD = upd(MD0,Exc), TP = TP0)),
 imply(
  member(MD0(MeM0(this)), last_id) + 1 <=
   member(MD(MeM(byte)),MaxValue),
  and(MeM = MeM0,
   MD = upd(upd(MD0, MD0(MeM0(this)),
              upd(MD0(MeM0(this)), last_id,
               member(MD0(MeM0(this),last_id)))),
              MD0(Val),
```

```
            member(MD0(MeM0(this),last_id))),
   TP = TP0)))

Q:Main:

any(j,imply(and(0<=j, j<member(MD(MeM(byte)),MaxValue) + 1),
       member(MD(MeM(ids)),j)= j))

INV:

and(any(j,imply(and(0 <= j, j < MD(MeM(i))),
     member(MD(MeM(ids)),j) = j)),
 0 <= MD(MeM(i)),
 MD(MeM(i)) <= member(MD(MeM(byte)),MaxValue) + 1)
```

This program is translated into the following C#-kernel program:

```
public class IdManager {
    public static void SFI() {
        MD := upd(MD, MeM(IdManager),
                  upd(MD(MeM(IdManager)), instance, null));
        new_instance(x5);
        TP := upd(TP, MeM(x5), IdManager);
        x5.IFI();
        x5.IdManager();
        MD := upd(MD, MeM(IdManager),
                  upd(MD(MeM(IdManager)), instance, MeM(x5)));
    }
    public void IFI() {
        MD := upd(MD, MD(MeM(this)), upd(MD(MeM(this)), last_id, 0));
    }

    private IdManager(){}
    private static IdManager instance;
    public static IdManager get_Instance() {
        {   MD := upd(MD, Val, member(MD(MeM(this)), instance));
            goto a2;
        }
        a2: ;
    }
    private byte last_id;
    public byte get_Id() {
        Init(int);
        new_instance(x0);
        TP := upd(TP, x0, int);
        MD := upd(MD, MeM(x0), member(MD(MeM(this)), last_id));
```

```
        int.+(x0, 1);
        new_instance(x0);
        TP := upd(TP, x2, int);
        MD := upd(MD, MeM(x2), MD(Val));
        new_instance(x12);
        TP := upd(TP, x12, int);
        Init(byte);
        MD := upd(MD, MeM(x12), member(MD(MeM(byte)),MaxValue));
        Init(bool);
        new_instance(x1);
        TP := upd(TP, x1, bool);
        int.>(x2, x12);
        MD := upd(MD, MeM(x1), MD(Val));
        if(x1) {
                Init(System_Exception);
                new_instance(x4);
                TP := upd(TP, MeM(x4), System_Exception);
                x4.IFI();
                x4.System_Exception("Error");
                MD := upd(MD, Exc, MeM(x4));
        }
        else {
            new_instance(x6);
            TP := upd(TP, x6, int);
            MD := upd(MD, MeM(x6), member(MD(MeM(this)), last_id));
            new_instance(x4);
            TP := upd(TP, x4, int);
            int.+(x6, 1);
            MD := upd(MD, MeM(x4), MD(Val));
            MD := upd(MD, MD(MeM(this)),
                   upd(MD(MeM(this)), last_id, MD(MeM(x4))));
            MD := upd(MD, MD(Val), MD(MeM(x6)));
            goto a1;
        }
        a1: ;
    }
}

public class Example {
    public void SFI() {}
    public void IFI() {}

    public static void Main()
    {   Init(int);
        new_instance(MAX_SIZE);
        TP := upd(TP, MAX_SIZE, int);
        new_instance(x16);
```

```
TP := upd(TP, x16, int);
Init(byte);
MD := upd(MD, MeM(x16), member(MD(MeM(byte)), MaxValue));
new_instance(x17);
TP := upd(TP, x17, int);
int.+(x16,1);
MD := upd(MD, MeM(x17), MD(Val));
MD := upd(MD, MeM(MAX_SIZE), MD(MeM(x17)));

new_instance(ids);
TP := upd(TP, ids, int[]);
new_instance(x11);
TP := upd(TP, MeM(x11), int[]);
x11.IFI();
x11.int[](MAX_SIZE);
MeM := upd(MeM, ids, MeM(x11));
{
    {
        new_instance(i);
        TP := upd(TP, i, int);
        MD := upd(MD, MeM(i), 0);
        {
            a3:
            Init(bool);
            new_instance(x7);
            TP := upd(TP, x7, bool);
            int.<(i, MAX_SIZE);
            MD := upd(MD, MeM(x7), MD(Val));
            if (x7) {
                {
                    /// <annotation>  INV  </annotation>
                    Init(IdManager);
                    new_instance(x8);
                    TP := upd(TP, x8, IdManager);
                    IdManager.get_Instance();
                    MeM := upd(MeM, x8, MD(Val));
                    new_instance(x9);
                    TP := upd(TP, x9, byte);
                    x8.get_Id();
                    MD := upd(MD, MeM(x9), MD(Val));
                    MD :=
                     upd(MD, MeM(ids),
                      upd(MD(MeM(ids)), MD(MeM(i)),
                       MD(MeM(x9))));
                }

                int.+(i, 1);
```

```
                        MD := upd(MD, MeM(i), MD(Val));
                        goto a3;
                    }
                    else {}
                }
                if (catch(x10)) {}
        }
    }
}
```

We omit the annotations in the resulting C#-kernel program except `INV` for simplicity.

The axiomatic proof system generates lazy VCs for the C#-kernel program. We consider one verification condition, generation of which uses most of the rules of the proof system. Let

$$(\mathtt{I})\ \mathtt{results\ in\ Q}:\ \mathtt{A}$$

denote that the application of the rule `I` of the proof system results in the postcondition `A` hereinafter referred to as `Q`.

The verification condition `Q` is associated with the body of the method `Main` in the case when the `if` statement condition `x7` is true, and static initialization and exception throwing is not performed. It is obtained by the following updates of the postcondition `Q:Main`:

```
Q1:
 Q:Main

(5) results in Q2:
 Q1

(i3) results in Q3:
 Q4

(5) results in Q4:
 Q3

(3) results in Q5:
 imply(MD(Exc) = Om, Q1)

(4) results in Q4:
 Q3

(5) results in Q5:
 Q4
```

```
(6) results in Q6:
 INV([MeM, MD, TP], a3)

(3) results in Q7:
 imply(MD(Exc) = Om, Q6)

(14) results in Q8:
 Q7(MD <- upd(MD, MeM(i), MD(Val)))

(3) results in Q9:
 imply(MD(Exc) = Om, Q8)

(8) results in Q10:
 CALL(+, int, [i, 1], [MeM, MD, TP], Q9)

(3) results in Qii:
 imply(MD(Exc) = Om, Q10)

(5) results in Q12:
 Q11

(14) results in Q13:
 Q12(MD <- upd(MD,MeM(ids),
           upd(MD(MeM(ids)),MD(MeM(i)),MD(MeM(x9)))))

(3) results in Q14:
 imply(MD(Exc) = Om, Q13)

(14) results in Q15:
 Q14(MD <- upd(MD, MeM(x9), MD(Val)))

(3) results in Q16:
 imply(MD(Exc) = Om, Qi5)

(8) results in Q17:
 CALL(get_Id, x8, [], [MeM, MD, TP], Q16)

(3) results in Q18:
 imply(MD(Exc) = Om, Q17)

(14) results in Q19:
 Q18(TP <- upd(TP, x9, byte))
```

```
(3) results in Q20:
 imply(MD(Exc) = Om, Q14)

(i4) results in Q2i:
 Q20(MeM <= upd(MeM, x9, nc1))

(3) results in Q22:
 imply(MD(Exc) = Om, Q21)

(14) results in Q23:
 Q22(MeM <- upd(MeM, x8, MD(Val)))

(3) results in Q24:
 imply(MD(Exc) = Om, Q23)

(8) results in Q25:
 CALL(get_Instance, IdManager, [], [MeM, MD, TP], Q24)

(3) results in Q26:
 imply(MD(Exc) = Om, Q25)

(14) results in Q27:
 Q26(TP <- upd(TP, x8, IdManager))

(3) results in Q28:
 imply(MD(Exc) = Om, Q27)

(i4) results in Q29:
 Q28(MeM <- upd(MeM, x8, nc2))

(3) results in Q30:
 imply(MD(Exc) = Om, Q29)

(15) results in Q31:
 Q30

(3) results in Q32:
 imply(not(MeM(IdManager) = Om), Q31)

(3) results in Q33:
 imply(MD(Exc) = Om, Q32)
```

```
(9) results in Q:
 imply(INV, Q33)
```

The condition `Q` contains one lazy invariant

```
INV([MeM, upd(MD, MeM(i), MD(Val)), TP], a3),
```

which is resolved according to the refinement algorithm.

The verification condition `Q` contains three lazy invocations. The lazy invocation `CALL(+, int, [i, 1], mvs1, Q9)` is refined to the pre- and postconditions of the standard library operator `+`. The lazy invocation `CALL(get_Id, x8, [], mvs2, Q16)` is refined to the precondition `P` and the postcondition `Q:get_Id`. The lazy invocation `CALL(get_Instance, IdManager, [], mvs3, Q24)` is refined to the precondition `P` and the postcondition `Q:get_Instance`.

9. Conclusion

In this paper, we present the three-level approach to the sequential object-oriented program verification that extends our two-level approach to C program verification [10]. The three-level approach is applied to a significant C# subset called C#-light that includes all principal sequential C# constructs.

The C#-light program verification process includes

- translation from C#-light into C#-kernel,
- backward generation of lazy verification conditions by means of the symbolic execution of Hoare-like logic rules developed for C#-kernel,
- refinement of lazy verification conditions.

The advantages of the approach are as follows:

- Essential simplification of the Hoare-like logic by means of translation of some semantically difficult C#-light constructs into C#-kernel, and postponement of handling some dynamic constructs until the refinement stage.
- Unambiguous inference of lazy verification conditions in the Hoare-like logic by means of backward generation rules that simplifies automatic generation of verification conditions similar to the Pascal program case [7].

Theoretical justification of our approach requires the proof of both the soundness of the Hoare-like logic w.r.t. operational C#-light semantics and the correctness of translation from C#-light into C#-kernel.

The three-level approach is promising for applications. It is suggested to develop an experimental tool for C#-light program verification based on this approach, and to apply it to important parts of library programs. However, verification of large C#-light programs is still a challenge. It should be noted that two approaches applied to Java programs are promising for this difficult problem: the modular approach [9] and extended static checking [8]. For some applications it is useful to develop a static analyzer of C#-light programs in addition to the verification tool for them.

References

[1] Apt K.R., Olderog E.R. Verification of Sequential and Concurrent Programs. — Berlin a.o.: Springer Verlag, 1991.

[2] Börger E., Schulte W. A programmer friendly modular definition of the semantics of Java // Lect. Notes Comput. Sci. — 1999. — Vol. 1523. — P. 353–404.

[3] Dubranovsky I.V. The translation from C#-light into C#-kernel. — Novosibirsk. — (Prep. / IIS SB RAS; to appear).

[4] Standard ECMA-334 — C# Language Specification. — December 2001. — `http://www.ecma.ch`

[5] Huisman M., Jacobs B. Java program verification via a Hoare logic with abrupt termination // Proc. FASE 2000. — Lect. Notes Comput. Sci. — 2000. — Vol. 1783. — P. 284–303.

[6] Huisman M., Jacobs B. Inheritance in higher order logic: modeling and reasoning // Proc. TPHOLs 2000. — Lect. Notes Comput. Sci. — 2000. — Vol. 1869. — P. 301–319.

[7] Igarashi S., London R.L., Luckham D.C. Automatic program verification I: a logical basis and its implementation // Acta Informatica. — 1975. — Vol. 4. — P. 145–182.

[8] Leino K.R.M. Extended static checking: a ten-year perspective // Lect. Notes Comput. Sci. — 2001. — Vol. 2000. — P. 157–175.

[9] Muller P. Modular specification and verification of object-oriented programs // Lect. Notes Comput. Sci. — 2002. — Vol. 2262.

[10] Nepomniaschy V.A., Anureev I.S., Promsky A.V. Verification-oriented language C-light and its structural operational semantics // Proc. of PSI'03. — Lect. Notes Comput. Sci. — 2003. — Vol. 2890. — P. 103–111.

[11] Oheimb D. v. Hoare logic for Java in Isabelle/HOL // Concurrency and Computation: Practice and Experience. — 2001. — Vol. 13, N 13. — P. 1173–1214.

[12] Oheimb D. v., Nipkow T. Hoare logic for NanoJava: auxiliary variables, side effects, and virtual methods revisited // Proc. FME 2002. — Lect. Notes Comput. Sci. — 2002. — Vol. 2391. — P. 89–105.

[13] Poetzsch-Heffter A., Muller P. A programming logic for sequential Java // Proc. ESOP'99. — Lect. Notes Comput. Sci. — 1999. — Vol. 1576. — P. 162–176.

[14] Reus B., Wirsing M., Hennicker R. A Hoare calculus for verifying Java realizations of OCL-constrained design models // Proc. FASE 2001. — Lect. Notes Comput. Sci. — 2001. — Vol. 2029. — P. 300–317.