Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

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

TOWARDS C# PROGRAM VERIFICATION: A THREE-LEVEL APPROACH

Preprint
128

Novosibirsk 2005

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. A Hoare-like logic is presented for C#-kernel. At the second stage, lazy verification conditions are generated by means of the Hoare-like logic. The generated verification conditions are lazy because they can include symbols representing postponed extractions of invariants of labelled statements as well as postponed invocations of methods and delegates. At the third stage, lazy verification conditions are refined using some algorithms of operational semantics. This approach allows us to simplify axiomatic semantics and to make unambiguous inference of lazy verification conditions. An example of verification of a C#-light program serves to illustrate this approach.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

В.А. Непомнящий, И.С. Ануреев,
И.В. Дубрановский, А.В. Промский

НА ПУТИ К ВЕРИФИКАЦИИ C#-ПРОГРАММ: ТРЕХУРОВНЕВЫЙ ПОДХОД

Препринт
128

Новосибирск 2005

Мы представляем новый трехуровневый подход к верификации последовательных объектно-ориентированных программ. Он применяется к выразительному подмножеству C#-light языка C#, которое включает все его основные последовательные конструкции. На первом этапе язык C#-light транслируется в промежуточный язык C#-kernel. На втором этапе порождаются ленивые условия корректности посредством аксиоматической семантики, разработанной для языка C#-kernel. Эти условия являются ленивыми, так как они могут включать специальные функциональные символы, представляющие отложенное уточнение инвариантов помеченных операторов, а также отложенные вызовы методов и делегатов. На третьем этапе эти условия уточняются с использованием алгоритмов операционной семантики. Этот подход позволяет упростить аксиоматическую семантику, а также однозначно выводить условия корректности. Пример верификации C#-light программы иллюстрирует данный подход.

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 C# [3]. 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, 15, 16, 18, 19]. 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 [15].

We develop a new three-level approach to verification of C# programs for a wide sequential subset called C#-light. The approach combines operational and axiomatic semantics. A preliminary version of this approach has been published in [14].

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. Based on the syntax of C#-light, C#-kernel only permits the `if` statement, block statement, `goto` statement, labeled statement, empty statement and expression statement. The translation to this restricted set of statements is achieved by introducing metainstructions into C#-kernel, which are used to handle metavariables encoding the states of C#-light programs.

At the second stage, lazy verification conditions are generated by means of forward rules of axiomatic C#-kernel semantics. These verification conditions are lazy because they can include special functional symbols representing the postponed extractions of invariants of labeled statements, as well as the postponed invocations of methods and delegates.

At the third stage, lazy verification conditions are refined using some

algorithms of operational semantics.

The main difference with [14] consists in a modification of C#-kernel axiomatic semantics to reduce the number of generated verification conditions.

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

This research has been partially supported by a gift from Microsoft Research within the ROTOR project in 2002–2003 and is partially supported by RFBR grant 04-01-00114a.

2. THE C#-LIGHT LANGUAGE

The C#-light language includes a considerable subset of C#. Unsupported constructs are as follows:

1. *Attributes.* At the first sight the concept of attributes is quite simple. It allows programmers to invent new kinds of declarative information, attach this information to various program entities, and retrieve this declarative information at run-time [1]. But their formalization must take into account the compilation process and various rules of ambiguities resolution. In most cases user attributes have no significant importance and the built-in ones, as a rule, are used in the cases which are out of the scope of our project, such as interaction of .NET packages and COM servers.
2. *Destructors.* The garbage collector underlying C# is invisible to most developers. Thus, in contrast to C++, we cannot predict precisely the moment of object deallocation and, consequently, the moment of destructor invocation.
3. *The* `using` *statement.* Similarly to destructors, this statement has implicit connection to the garbage collection mechanism.
4. *The* `checked` *and* `unchecked` *operators.* These constructs determine the reaction of a virtual machine to overflow for integral-type arithmetic operations and conversions. They could complicate the expres-

sion evaluation semantics, especially if we try to model how the high-order bits are discarded in an `unchecked` context.

5. *Unsafe code.* The security is one of the cornerstones of the .NET platform. Unsafe code can violate the requirements of security. In addition, such low-level memory manipulations depend on a concrete realization.
6. *Pre-processing directives.* In C#, pre-processing directives are processed as a part of the lexical analysis phase, so they cannot be unfolded by a separate tool [11]. Fortunately, the expressiveness of C# does not suffer in a high degree if these directives are discarded.

Thus, the C#-light language is a representative sequential subset of C#. In comparison with Java, it includes distinctive features of C# such as properties, events, delegates and indexers.

C#-light proposes the means to annotate programs. The annotations are comments of the form

```
/// <a> R </a>
```

where `R` is a formula of the specification language (see the next sect.)

3. THE SPECIFICATION LANGUAGE

The *specification language* is used to express the program annotations (pre- and postconditions, loop and label invariants). Speaking more generally, it is used to write assertions about program properties. Traditionally, the first-order language seems to be a good choice for a simple program language. It can become insufficient for complex concepts, such as aliasing. Some researchers were forced to use new logical constructs (the `if`-term, for example) or complex substitution definitions [2, 17]. Instead of this, we prefer to extend the first-order language by higher-order functions and some elements of λ-calculus. This approach is also widespread and used, for example, in [15, 16].

3.1. Types

The types of the specification language include the base types $\mathcal{U}$, $\mathcal{N}$, $\mathcal{T}$, functions $T \to T'$ and Cartesian products $T \times T'$, where

- $\mathcal{U}$ is a universal set which includes, at least, all C# literals, the set $\mathcal{L}$ of storage locations, the set $\mathcal{N}at$ of natural numbers (including zero) and undefined value ω,
- $\mathcal{N}$ is the set of C# identifiers,

- $\mathcal{T}$ is the set of type names.

The class and structure instances are denoted directly by the corresponding storage locations [1, §8.3]. Let $Loc(T)$ denote storage locations represented by C# variables of type T.

3.2. Expressions

Expressions of the specification language are defined by induction as follows:

- the variables and constants of type T are expressions of type T;
- if $s_1, \ldots, s_n$ are expressions of type $T_1, \ldots, T_n$, respectively, and s is an expression of type $T_1 \times \ldots \times T_n \to T$, then $s(s_1, \ldots, s_n)$ is an expression of type T;
- if x is a variable of type T and s is an expression of type T', then $\lambda(x, s)$ is an expression of type $T \to T'$, called λ-term. As usual, it defines a function which, being applied to its argument a, produces the value of the expression s, where every occurrence of x is substituted by a.

We fix a special function name *upd* with the following standard interpretation: if s is an expression of type $T \to T'$, e_1 and e_2 are expressions of type T and T', respectively, then $upd(s, e_1, e_2)$ is an expression of type $T \to T'$, which is equal to s everywhere except, maybe, e_1 and $upd(s, e_1, e_2)(e_1) = e_2$.

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

Logical expressions, called *annotations*, are built from expressions of type *bool* with the help of logical connectives $\wedge$, $\vee$, $\neg$, $\Rightarrow$ and quantifiers $\exists$ and $\forall$ in a usual way.

3.3. Metavariables and states

The interpretation of annotations is based on the notion of states. We define a *state* as a mapping of the specification language variables into their values.

In fact, the classical approach [2] treats states as mappings of program variables, because these variables are denoted by the same names in the expression language, which is a first-order language. For example, execution of the assignment $x := y$ in a state σ leads to the modified state $upd(\sigma, x, y)$. In axiomatic semantics this situation is modeled by well-known Hoare axiom

$$\{P(x \leftarrow y)\}\, x := y\, \{P\}.$$

This axiom may become unsound in the case of aliasing. In [17] the complicated redefinition of substitution was proposed to solve this problem.

In our approach, the program variables are denoted by *constants* of type $\mathcal{N}$ with the same names. The link between such "variables" and values in the corresponding storage locations is established by the fixed the specification language variables, called *metavariables*[1]. They include:

1. the metavariable L of type $\mathcal{N} \to \mathcal{L}$, which maps the program variable names into the corresponding storage locations;
2. the metavariable V of type $\mathcal{L} \to \mathcal{U}$, which produces the values stored in storage locations;
3. the metavariable T of type $\mathcal{N} \cup \mathcal{U} \to \mathcal{T}$, which produces the types of program variables from $\mathcal{N}$ and the types of literals from $\mathcal{U}$;
4. the metavariable $L2$ of type $\mathcal{U} \times (\mathcal{N} \cup \mathcal{N}at) \to \mathcal{L}$, which maps the object fields and array elements into storage locations;
5. the metavariable $V0$ of type $\mathcal{U}$, which contains the value of the last evaluated expression;
6. the metavariable E of type $\mathcal{U}$, which contains the value of the catched exception.

Note. This fixation of the metavariable names leads to an implicit restriction: to avoid ambiguity, the set of program identifiers cannot contain the names L, V, T, $L2$, $V0$, E. For example, if there is an integer program variable `L`, then what is the type of the specification language expression L: $\mathcal{N}$ or $\mathcal{N} \to \mathcal{L}$? But it is not necessary to state such a restriction in the definition of C#-light, because it is sufficient just to rename the identifiers in a program.

In C#-light, the assignment considered above leads to the modified state

$$upd(\sigma, V, upd(\sigma(V), \sigma(L)(x), \sigma(V)(y))),$$

and the corresponding Hoare triple takes the following form:

$$\{P(V \leftarrow upd(V, L(x), V(y))\}\, x \texttt{ = } y\texttt{;}\, \{P\}.$$

The metavariables describe precisely the state of the abstract C#-light machine. The main advantage of this approach is that we can use the simplest definitions of state and substitution.

[1] Most of them are mappings of some type. That's why we use the prefix "meta" to reveal their higher-order nature and to separate them from program variables.

3.4. Notational conventions

Nevertheless, such a meta-approach makes expressions of the specification language considerably cumbersome. As we have seen, the term x turns into $V(L(x))$. Active use of "upd" term cause a combinatorial explosion in the length of expressions, which makes them totally unreadable. We develop some strategies of optimization in our verification tool. For a while, in this paper we shall use the following abbreviations:

Original term	Short form
$V(L(x))$	V(x)
$L2(V(x), y)$	$L2(x, y)$
$V(L2(x, y))$	$V2(x, y)$

We will also use the name mvs to denote the tuple $[L, V, T, L2, V0, E]$. The combination of mvs with some index (or dash) means that the components of the tuple have the same index (dash). For example, mvs_i stands for $[L_i, V_i, T_i, L2_i, V0_i, E_i]$.

4. THE C#-KERNEL LANGUAGE

The C#-kernel language is an object-oriented language that is based on the C#-light's subset S defined by the following restrictions:

- S does not contain namespaces and `using`-directives;
- S does not contain events;
- 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`, iteration statements, declaration statements;
- S contains only those `if` statements that have an `else`-branch and have a boolean variable as a conditional expression;
- an invocation of a static function member is located only in those program 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 within a program;
- the sets of labels, local variable names, local constant names and type names are disjoint.

The subset S is extended by metainstructions, and restrictions are applied to the expression statements and the class and struct declarations.

4.1. Metainstructions

Metainstructions are used to handle metavariables. In C#-kernel, there are five metainstructions:

1. `x := e` assigns the expression `e` in the annotation language to the metavariable `x`.
2. `new_instance()` allocates a new storage location and puts it to `V0`.
3. `Init(C)` performs static initialization of the class/struct `C` if the type `C` has not yet been initialized.
4. `catch(T, x)` returns `true` if `E` stores a value of the type `T`, and `false` otherwise. Additionally, in the former case the value of the variable `x` is set to the exception object located in `E`, and the value of the metavariable `E` is set to ω to indicate that the exception has been caught. This metainstruction can be used only as a conditional expression within an `if` statement.
5. `catch(x)` returns `true` if `E` $\neq \omega$, and `false` otherwise. Additionally, in the former case the value of the variable `x` is set to the exception object located in `E`, and the value of the metavariable `E` is set to ω. This metainstruction can be used only as a conditional expression within an `if` statement to model the generic `catch` section of a `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 on C#-light expressions:

- a normalized expression has the form `x.y(`z_1`, ..., `z_n`)` or `y(`z_1`, ..., `z_n`)`, where `x` is a variable name or a type name, `y` is a method name, a delegate name, a constructor name, or an operator, z_1, ..., z_n are variable names (possibly with the `ref` and `out` modifiers), literals or metavariables `E` and `V0`;
- in normalized expressions, the logical operators `||` and `&&`, conditional operator `?:`, operator `new` and all assignment operators are not permitted;

- in normalized expressions, the function members can be invoked only in their normal form [1].

4.3. The class and struct declarations

Declarations of fields and constants of classes and structs do 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() { ... }
```

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 of a class `C` takes place in the method

```
public void IFI_C() { ... }
```

These methods extend the context for direct assignments to a `readonly` field [1].

5. TRANSLATION FROM C#-LIGHT INTO C#-KERNEL

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

Let us briefly describe the rules of translation from C#-light into C#-kernel. The complete description can be found in [4].

5.1. Expression normalization

The expression-statement normalization is performed by the Norm transformation that is defined with the help of C# expression production rules. For each type of expression, Norm specifies how to split the expression into several normalized expression statements.

Invocations of function members in the expanded form are replaced by invocations of function members in the normal form. To achieve this, in each

invocation expression, arguments that correspond to the parameter array are replaced by an array creation expression that allocates memory for the array of these arguments. For example, for a functional member of the form

```
public void F(int p, param int[] a);
```

the expression `F(1, 2, 3, 4);` is transformed into the expression

```
F(1, new int[] 2, 3, 4);
```

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

```
if (x + y) { ... }
```

is transformed into the fragment

```
bool z = x + y;
if (z) { ... }
```

The requirement that an invocation of a static function member is located only in those program points 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 with the help of the `if` statement. For example, the expression

```
x ? e1 : e2;
```

is replaced by the fragment

```
if (x)
    { Norm[(T) e1]; }
else
    { Norm[(T) e2]; }
```

where `T` is the type of the conditional expression.

The assignment operators, as well as the field and array access, are expressed with the help of the metainstruction `:=`. The operator `new` is expressed with the help of the metainstructions `:=`, `new_instance` and the `IFI` method. For example, if `c` is a variable of a class `C`, then the fragment

```
c = new C();
```

is translated as follows:

```
Init(C);
new_instance();
L := upd(L, x, V0);
T := upd(T, x, C);
```

```
T := upd(T, L(x), Loc(C));
new_instance();
V := upd(V, L(x), V0);
T := upd(T, V(x), C);
x.IFI_C(); x.C();
V := upd(V, L(c), V(x));
```

Note that a helper local variable `x` is used in this example.

Events are reduced to delegates.

The property, indexer and operator declarations are duplicated in the method declarations according to the reserved member names [1]. After that, the property and indexer access, as well as operator invocations, are transformed into invocations of appropriate methods. For example, for an event of the form

```
public int P
{
    get { ... }
    set { ... }
}
```

declared in a class `C`, the methods

```
public int get_P();
```

and

```
public void set_P(int value);
```

are added to `C`, and the expressions `P = 5;` and `x = P;` are translated into the expressions `set_P(5);` and `x = get_P();`, respectively.

The field and array access, as well as field and array element updates, are expressed with the help of the metainstruction `:=`. For example, if a field of the type `int` is declared in a class `C`, then

```
c.a = c;
```

is transformed into

```
V := upd(V, L2(c, a), V(c));
```

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

5.2. Statement elimination

The declaration statements are translated into the metainstructions `:=` and `new_instance`. For example, the declaration of a struct

```
S x;
```

has the following form in C#-kernel:

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

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 metavariable `V0` are used to set the return value of the function member. So if `x` is a local variable that holds the result of evaluation of `e`, then

```
V0 := V(x);
```

sets the return value of the function member.

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

The selection statement `switch` is expressed with the help of 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.

The `try` statement elimination is considered as a part of the exception handling elimination described further.

5.3. Exception handling elimination

In this section, we will detail our translation method for exception handling elimination.

The main steps are the following.

1. The statements `goto case` and `goto default` are replaced by the `goto` statement.
2. When we eliminate the statement

   ```
   throw e;
   ```

   the metainstruction `:=` is used to raise an exception which is the result of evaluation of `e`. The exception is stored in the metavariable `E`.
3. The `try` block is replaced by its content.
4. The `catch` blocks are replaced by the `if` statements with conditional expressions of the form `catch(T, x)` or `catch(x)`.

For the purpose of elimination, we introduce labeled `finally`-blocks. To transform general `goto`, `goto case` and `goto default` into the goto statements that transfer control to labeled `finally` blocks (in the sequel, this process is called normalization), a special algorithm that is described further has been elaborated. This algorithm uses the rule of normalization of labels of the switch-statement.

Rule of switch-label normalization (SLN). A fragment of the form

switch-labels statement-list

where *statement-list* does not start with a labeled statement of the form *identifier*`:;` produced by this rule, is replaced by

switch-labels `L:;` *statement-list*

where `L` is a new unique label.

Algorithm of the `goto` statement normalization. For brevity, let us consider normalization of the `goto case` and `goto default` statements omitting normalization of the `goto` statement. Let $Lab(B)$ be the set of all top-level labels in a block or statement B. Let $\mathtt{L}_1(B)$, ..., $\mathtt{L}_K(B)$ denote the elements of the set $Lab(B)$. Let V be the set of all values of the constant expressions permitted in switch-statements.

1. All switch-statements are enumerated in a textual order.
2. The **SLN** rule is applied.
3. Let Lab^{new} be the set of all labels produced by the **SLN** rule at the stage of normalization of the `switch` statements, $Lab^{new}(S) = \{\mathtt{L}_1(S), ..., \mathtt{L}_{n(S)}(S)\}$ be the set of such labels in a `switch`-statement S. Let us define correspondences

$$CaseL : V \times \mathcal{N} \to Lab \qquad \text{and} \qquad DefL : \mathcal{N} \to Lab$$

   between the labels of `switch`-statements and the new labels. For all labels of the form `case E:` that get to *switch-labels* of the statement S_i, $CaseL(\mathtt{E}, i)$ `:=` `L`, where `L` is a label produced by the **SLN** rule for a `switch`-section containing `case E:`. For all labels of the form `default:` that get to *switch-labels* of the statement S_i, $DefL(i)$ `:=` `L`.
4. New different elements $\mathtt{l}_1, ..., \mathtt{l}_N$, $N = |Lab^{new}|$ are added to the enumeration `GT_LABELS`. Let the function $v : Lab^{new} \to$ `GT_LABELS` defines a one-to-one correspondence between the elements of Lab^{new} and $\mathtt{l}_1, ..., \mathtt{l}_N$.

5. For all `switch`-statements S, for all blocks B in S, excluding the bodies of nested `switch`-statements, for all `finally`-blocks in B at the top level, if a `finally`-block `finally {A}` is higher than all other `finally`-blocks of S, then it is replaced by

   ```
   finally(f_0) {
       A
       if (GT.gt == GT_LABELS.v(L_1(S))) goto L_1(S);
       else if (GT.gt == GT_LABELS.v(L_2(S))) goto L_2(S);
       ...
       else if (GT.gt == GT_LABELS.v(L_n(S)(S))) goto L_n(S)(S);
   }
   ```

   otherwise, it is replaced by

   ```
   finally(f_{j+1}) {
       A
       if (GT.gt != GT_LABELS.none) goto f_j;
   }
   ```

   where $\mathtt{f}_j$ are new different labels, $j \geq 0$. The blocks B are enumerated recursively starting from the body of the statement S.

6. For all switch-statements S_i, for all statements `goto case E;` in the statement S_i, if `goto case E;` jumps out of a `try`-statement with a `finally`-block labeled with a label $\mathtt{f}_j$, then `goto case E;` is replaced by

   ```
   GT.gt = GT_LABELS.v(CaseL(E,i));
   goto f_j;
   ```

   Otherwise, `goto case E;` is replaced by `goto` $CaseL(\mathtt{E}, i)$`;`.

7. For all switch-statements S_i, for all statements `goto default;` in the statement S_i, if `goto default;` jumps out of a `try`-statement with a `finally`-block labeled with a label $\mathtt{f}_j$, then `goto default;` is replaced by

   ```
   GT.gt = GT_LABELS.v(DefL(i));
   goto f_j;
   ```

   Otherwise, `goto default;` is replaced by `goto` $DefL(i)$`;`.

Note that in the points 5–7, the quantifier application strategy has the form of a recursive traversal of blocks, moving from the outermost to the innermost block. However, the statements `goto case` and `goto default` are handled in a textual order starting from the topmost statement.

So the above algorithm transforms the statements `goto case` and `goto default` to the ordinal `goto`. Additionally, in the resulting program the `goto` statements do not jump out of the `try` statements. In this case, the elimination of `try` is permitted. The deletion of `try` is a local transformation performed by a set of rules that are applied non-deterministically. Let us consider an example of the mentioned rules.

Rule ETRY1. The fragment of the form

```
try {A} finally(L) {B}
```

is replaced by

```
{A} L:;
    if (catch(x))
    {
        B
        MD := upd(MD, Exc, MD(MeM(x)));
        goto M;
    }
    {B}
    M:;
```

where `M` is a new label.

Rule ETRY2. The fragment of the form

```
try {A} catch (T1 x) {C1}
        catch (T2 x) {C2}
        ...
        catch (Tn x) {Cn}
        catch        {Cg}
```

is replaced by

```
{A} if      (catch(T1, x)) { T1 eSaved; eSaved = x; C1 }
    else if (catch(T2, x)) { T2 eSaved; eSaved = x; C2 }
    ...
    else if (catch(Tn, x)) { Tn eSaved; eSaved = x; Cn }
    else if (catch(eSaved)) { Cg }
```

The remaining rules are similar to ETRY1, ETRY2 and are described in [4].

5.4. `Using` directive and namespace elimination

`Using` directive elimination is performed in two steps. All names are replaced by their appropriate qualified names. After that, the using directives are removed.

Namespace elimination is performed as follows. All qualified names are replaced with their fully qualified names [1]. In this case, 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. C#-KERNEL AXIOMATIC SEMANTICS

The base formulas of C#-kernel axiomatic semantics are constructs of the form $\{P\}\,A\,\{Q\}$ (called *Hoare triples*), where A is a (possibly empty) sequence of statements and metainstructions, P and Q are annotations called a precondition and postcondition, respectively. The truth of a Hoare triple $\{P\}\,A\,\{Q\}$ is defined in the usual way: if execution of the program fragment A starts in a state in which the precondition P is true and terminates, then the postcondition Q is true in the final state. Note only that here termination means either normal termination or termination with throwing out an exception.

Let us introduce some definitions. The function members [1], standard library operators and property and indexer accessors will be called function components. Each function component in a program has a unique name called its identifier.

Access to the structure of a function component M with the identifer f is provided by the functions *class*, *body*, *SP*, *pre* and *post* such that $class(f)$ is the name of the class which contains M, $body(f)$ is the body of M, $SP(f)$ is the tuple of variables called 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 formulas of the annotation language called pre- and postconditions of M, respectively.

We denote the inheritance relation on types by $\subseteq$. Let SI be a predicate such that the formula $SI(C, L, V, T, L2)$ is true iff the class or struct C is initialized in the state defined by metavariables L, V, T and $L2$. Note that the metavariables E and $V0$ have no influence on type initialization.

Let us denote the formula

$$L(\texttt{this}) \notin \{\omega, \texttt{null}\} \wedge V(\texttt{this}) \neq \omega \wedge$$
$$T(V(\texttt{this})) \subseteq class(f) \wedge$$
$$SI(T(V(\texttt{this})), L, V, T, L2),$$

where f is an identifier of a function component, by $thiscond(f, mvs)$. This formula imposes usual restrictions on `this`, that hold for an invocation of any nonstatic function component with the identifier f. Let $thiscond(f, mvs)$ be equal to $true$ if f is an identifier of a static function component.

We suppose below that for each function component f the formula

$$pre(f)(mvs, SP(f)) \Rightarrow prec(f, mvs),$$

where $prec(f, mvs)$ means that

$$thiscond(f, mvs) \wedge E = \omega \wedge SI(class(f), L, V, T, L2)$$

is true. This formula imposes usual restrictions on `this`, E and $class(f)$ which hold before an invocation of f.

6.1. Program

The program rule reduces the proof of correctness of an annotated C#-kernel program Prg to the proofs of correctness of bodies of its function members. The proof of correctness of the body of the function member M is performed in the environment $Env = (Prg, f, EI, ICS)$, where f is an identifier of M, ICS is a set of initialized classes and struct of the program Prg. The exception indicator EI takes on the values $true$, $false$ and ω for cases when an exception has been thrown, no exception has been thrown and it is not known whether an exception has been thrown or not, respectively. Below, the superscripts $+$ and $-$ of the environment Env means that EI sets to $true$ and $false$, respectively. If there is no index, then suppose that $EI = \omega$.

Let us denote the truth of a Hoare triple $\{P\}\, A\, \{Q\}$ for the program fragment A in the environment Env by $Env \vdash \{P\}\, A\, \{Q\}$.

The environment (Prg, f, EI, ICS) is called conformed with the triple $\{P\}\, A\, \{Q\}$ if the following formulas are true:

- $EI = true \wedge P \Rightarrow E \neq \omega$,
- $EI = false \wedge P \Rightarrow E = \omega$,
- $P \Rightarrow SI(C, L, V, T, L2)$ for each class or struct $C \in ICS$.

The components EI and ICS of a conformed environment duplicate information of the precondition of the Hoare triple. They are used only to optimize the inference system (to reduce the number of the resulting verification conditions). Below we consider only conformed environments and design the inference system such that its rules preserve the conformance property.

Let f_i, where $1 \leq i \leq n$, be a set of all identifiers of constructors and methods of the program Prg except for initializing methods `SFI` and `IFI`. We do not verify initializing methods because they appear at the stage of translation from C#-light into C#-kernel and, therefore, have no specifications. Let us remind that declarations of properties, indexers and operators are duplicated in method declarations after the translation. Denote the constructs $(Prg, f_i, false, \{class(f_i)\})$, $pre(f_i)(mvs, SP(f_i))$ and $post(f_i)(mvs, SP(f_i))$ by Env_i^-, P_i and Q_i, respectively.

The rule for the program Prg with an entry point given by the method with the identifier f (this method is named `Main`) has the form:

$$\frac{\begin{array}{l} Env_i^- \vdash \{P_i\}\, body(f_i)\, \{Q_i\} \quad (1 \leq i \leq n) \\ (Prg, \omega, \omega, \emptyset) \vdash \begin{array}{l} \{P\} \\ \texttt{Init(}class(f)\texttt{);} \\ class(f)\texttt{.Main(V0);} \\ \{Q\} \end{array} \end{array}}{\{P\}\, Prg\, \{Q\}}\,, \tag{1}$$

The value of the metavariable $V0$ is an array of application parameters. If f does not contain parameters, the metavariable $V0$ is omitted in this rule. The value ω of the second component of the environment of the second branch of the rule means that the sequence of statements which contains no function component of the program Prg is executed.

6.2. Elimination of uncertainty in exception indicator

When defining the statement semantics, we consider separately two cases — an exception has been thrown and no exception has been thrown to the moment of the statement execution. As stated above, this information is contained in the exception indicator EI of the environment. The following rule removes uncertainty ($EI = \omega$), when it is not known whether an exception has been thrown or not to the moment of the statement execution,

reducing definition of the statement semantics to these two cases:

$$\frac{Env^- \vdash \{P \wedge E = \omega\}\, A\, \{Q\} \quad Env^+ \vdash \{P \wedge E \neq \omega\}\, A\, \{Q\}}{Env \vdash \{P\}\, A\, \{Q\}}\ . \tag{2}$$

6.3. Exception propagation

For most statements, execution of a statement in the case when an exception has been thrown is reduced to that the statement is ignored, and the exception is propagated to the next statement. To avoid writing similar rules, the rules of such kind are combined into one scheme of exception propagation.

Let S be a statement which is not a labelled statement and a statement `if` with metainstructions `catch(`T`,`x`)` or `catch(`x`)` as the governing expression. The exception propagation rule has the form:

$$\frac{Env^+ \vdash \{P\}\, A\, \{Q\}}{Env^+ \vdash \{P\}\, S\ A\, \{Q\}}\ . \tag{3}$$

6.4. The statement `if`

The rule for the statement `if` performs case analysis depending on whether the governing expresssion x is true or false. Remind that in C#-kernel the expression x can be only a variable. Therefore the value of this expression is $V(x)$:

$$\frac{Env^- \vdash \{P \wedge V(x) = true\} S_1\ A\{Q\} \quad Env^- \vdash \{P \wedge V(x) = false\} S_2\ A\{Q\}}{Env^- \vdash \{P\}\ \texttt{if(}x\texttt{)}\ S_1\ \texttt{else}\ S_2\ A\, \{Q\}}\ . \tag{4}$$

6.5. Block

The algorithm of garbage collection, and, in particular, the moment of deletion of local variables defined in a block depends on its implementation. Therefore local variables are not deleted in our semantics after exiting the block, and the rule for the block only eliminates braces:

$$\frac{Env^- \vdash \{P\}\, S_1 \ldots S_n\ A\, \{Q\}}{Env^- \vdash \{P\}\, \{S_1 \ldots S_n\}\ A\, \{Q\}}\ . \tag{5}$$

Note that no collision of names of local variables appears because all local variables of C#-kernel programs are unique.

6.6. The goto statement and the labelled statement

The rules for the goto statement and the labelled statement usually use special annotations called invariants which are attached to labels. In our approach invariants are not attached to labels but can be placed between any statements of a program. The only restriction is that any cyclic path in the program contains at least one invariant. In particular, this approach allows us to avoid the problem of attaching invariants to new labels which appear after translation from C#-light to C#-kernel. Instead of invariants, the rules for these two statements use special expressions $INV(e, L)$ called lazy invariants, where e is a tuple of expressions and L is a label. Lazy invariants are replaced by real invariants at the stage of refinement of verification conditions (see Section 7 below).

Let A and B be program fragments and A contains no labelled statements at the top level. Then the rules are as follows:

$$\frac{Env^- \vdash P \Rightarrow INV(mvs, L)}{Env^- \vdash \{P\}\,\texttt{goto}\ L;\ A\,\{Q\}}, \tag{6}$$

$$\frac{\begin{array}{l} Env^- \vdash \{P\}A\{INV(mvs, L)\} \\ Env \vdash \{INV(mvs, L)\}B\{Q\} \end{array}}{Env^- \vdash \{P\}\,A\ L{:}\ B\,\{Q\}}, \tag{7}$$

$$\frac{\begin{array}{l} Env^+ \vdash \{P\}A\{INV(mvs, L)\} \\ Env \vdash \{INV(mvs, L)\}B\{Q\} \end{array}}{Env^+ \vdash \{P\}\,A\ L{:}\ B\,\{Q\}}. \tag{8}$$

6.7. The expression statement

Semantics of an expression statement is defined by four rules.

The rule of invocation for constructors and methods, except for initializing methods from user-defined classes and structs, uses the predicate $CALL$. This predicate defines the resulting values of metavariables of mvs by the name x of the method or constructor, by the expression y referencing to an object or type, by the argument list $[z]$ and by the set of initial values f metavariables specified by the precondition P:

$$\frac{Env \vdash \{CALL(x, y, [z], mvs, \lambda(mvs, P))\}\,A\,\{Q\}}{Env^- \vdash \{P\}\,y.x(z);\ A\,\{Q\}}. \tag{9}$$

To preserve the initial values of metavariables invariable, i. e. as they were before the invocation, λ-expression $\lambda(mvs, P)$ is used in $CALL$ instead of P. Expressions of the form $CALL(\ldots)$ called lazy method invocations are defined at the refinement stage (see Section 7 below).

Initializing methods `SFI` and `IFI` for user-defined classes and structs have their own rules which replace invocations of these methods by their bodies because these methods appear after the translation and have no annotations:

$$\frac{Env^- \vdash \{P\}\, LocVarRen(S)\ A\,\{Q\}}{Env^- \vdash \{P\}\, C\texttt{.SFI(); }\ A\{Q\}}\ , \tag{10}$$

$$\frac{Env^- \vdash \{P\}\, LocVarThisRen(S)\ A\,\{Q\}}{Env^- \vdash \{P\}\, y\texttt{.IFI\_C(); }\ A\{Q\}}\ , \tag{11}$$

where S denotes the body of the invoked method, the function $LocVarRen$ renames local variables of S, the function $LocVarThisRen$, in addition to renaming, replace `this` by y in the body S.

The rule of the delegate invocation uses the predicate $DELCALL$. It defines the resulting values of metavariables of mvs by the delegate $V(x)$ to which the variable x refers, the argument list $[z]$ and the set of initial values of metavariables specified by the precondition P:

$$\frac{Env \vdash \{P'\}\, A\,\{Q\}}{Env^- \vdash \{P\}\, x\texttt{(}z\texttt{); }\ A\,\{Q\}}\ , \tag{12}$$

where P' stands for

$$DELCALL(V(x), [z], mvs, \lambda(mvs, P)).$$

Expressions of the form $DELCALL(\ldots)$ called lazy delegate invocations are defined at the refinement stage (see Section 7 below).

6.8. Annotations

Annotations placed in certain control points of a program are interpreted as invariants of these points:

$$\frac{Env^- \vdash P \Rightarrow R \qquad Env^- \vdash \{R\}\, A\,\{Q\}}{Env^- \vdash \{P\}\, \texttt{///}\ \texttt{<a>}R\texttt{}\ A\{Q\}}\ , \tag{13}$$

$$\frac{Env^+ \vdash P \Rightarrow R \qquad Env^+ \vdash \{R\}\, A\,\{Q\}}{Env^+ \vdash \{P\}\, \texttt{///}\ \texttt{<a>}R\texttt{}\ A\{Q\}}\ . \tag{14}$$

6.9. The empty statement and empty program

The rules for an empty statement and empty program are defined in the usual way:

$$\frac{Env^- \vdash \{P\}\, A\, \{Q\}}{Env^- \vdash \{P\}\ ;\ A\, \{Q\}}\ , \tag{15}$$

$$\frac{Env^- \vdash P \Rightarrow Q}{Env^- \vdash \{P\}\ \{Q\}}\ , \tag{16}$$

$$\frac{Env^+ \vdash P \Rightarrow Q}{Env^+ \vdash \{P\}\ \{Q\}}\ . \tag{17}$$

6.10. Metainstructions of exception catching

Let us remind that the metainstruction `catch(`t`,`x`)` can appear in a C#-kernel program only as a governing expression of a statement `if`. The case when no exception has been thrown and, therefore, this metainstruction returns the value $false$ is defined by the rule

$$\frac{Env^- \vdash \{P\}\, S_2\ A\, \{Q\}}{Env^- \vdash \{P\}\texttt{if(catch(}t\texttt{,}x\texttt{))}\ S_1\ \texttt{else}\ S_2\ A\{Q\}}\ . \tag{18}$$

The case when an exception has been thrown is defined by the rule with two premises:

$$\frac{\begin{array}{l} Env^- \vdash \{P'\}\, S_1\ A\, \{Q\} \\ Env^+ \vdash \{P \wedge T(E) \not\subseteq t\}\, S_2\ A\, \{Q\} \end{array}}{Env^+ \vdash \{P\}\texttt{if(catch(}t\texttt{,}x\texttt{))}\ S_1\ \texttt{else}\ S_2\ A\{Q\}}\ , \tag{19}$$

where the formula P' has the form

$$\begin{array}{l} \exists V'\ \exists E' (P(V \leftarrow V')(E \leftarrow E') \wedge \\ \qquad V = upd(V', L(x), E') \wedge \\ \qquad E = \omega \wedge T(E') \subseteq t)\ . \end{array}$$

The first premise meets the case when a type of the thrown exception is derived from the type t ($T(E') \subseteq t$). In this case, the exception is catched ($E = \omega$) and becomes the value of the variable x ($V = upd(V', L(x), E')$), and the metainstruction `catch` returns the value $true$. The second premise meets the case when the type of the thrown exception is not derived from the type t ($T(E) \not\subseteq t$) and the metainstruction returns the value $false$.

The rule for the metainstruction `catch(`x`)` is not a special case of the above rule for $t =$ `object` because this metainstruction can catch exceptions of the types which are not derived from `object`:

$$\frac{Env^- \vdash \{P\}\, S_2\ A\,\{Q\}}{Env^- \vdash \{P\}\, \texttt{if(catch(}x\texttt{))}\ S_1\ \texttt{else}\ S_2\ A\,\{Q\}}\ , \tag{20}$$

$$\frac{Env^- \vdash \{P''\}\, S_1\ A\,\{Q\}}{Env^+ \vdash \{P\}\, \texttt{if(catch(}x\texttt{))}\ S_1\ \texttt{else}\ S_2\ A\,\{Q\}}\ , \tag{21}$$

where P'' denotes

$$\begin{array}{l} \exists V'\ \exists E'\ (\\ P(V \leftarrow V')(E \leftarrow E') \wedge \\ V = upd(V', L(x), E') \wedge E = \omega)\ . \end{array}$$

6.11. The assignment metainstruction

The metainstruction x `:=` e in the case of $x \neq E$ is defined by the rule:

$$\frac{Env^- \vdash \{\exists x'\ P(x \leftarrow x') \wedge x = e(x \leftarrow x')\}\, A\,\{Q\}}{Env^- \vdash \{P\}\, x\ \texttt{:=}\ e;\ A\,\{Q\}}\ . \tag{22}$$

The assignment E `:=` e has a separate rule because it can change the environment (the value of exception indicator EI):

$$\frac{\begin{array}{l} Env^- \vdash \{P \wedge e = \omega\}\, A\,\{Q\} \\ Env^+ \vdash \{P'\}\, A\,\{Q\} \end{array}}{Env^- \vdash \{P\}\, E\ \texttt{:=}\ e;\ A\,\{Q\}}\ , \tag{23}$$

where P' denotes the formula

$$\exists E'\ P(E \leftarrow E') \wedge E = e(E \leftarrow E') \wedge E \neq \omega\ .$$

6.12. The metainstructions of static initialization and storage allocation

Let us denote by $newp(d, L, V, L2)$ the formula

$$\forall x \forall y (d \notin \{L(x), V(x), L2(x, y)\})$$

defining a new location d.

In the case when information about initialization of a class or struct C is contained in the environment, i. e. $C \in ICS$, the rule for the metainstruction `Init(`C`)` has the form:

$$\frac{Env^- \vdash \{P\}\, A\, \{Q\}}{\texttt{Env}^- \vdash \{\texttt{P}\}\, \texttt{Init(C); A}\, \{\texttt{Q}\}} \ . \tag{24}$$

If $C \notin ICS$, the case analysis is performed. The first premise meets the case when the type C has been initialized and information about this fact is contained in the precondition P. The second premise meets the case of initialization of the type C which includes initialization of static fields `C.SFI()` and execution of the static constructor `C.C()`:

$$\frac{\begin{array}{l} upd(Env^-, ICS, ICS \cup \{C\}) \vdash \\ \quad \{P \wedge SI(C, L, V, T, L2)\}\, A\, \{Q\} \\ upd(Env^-, ICS, ICS \cup \{C\}) \vdash \\ \quad \{P'\}\, C\texttt{.SFI(); } C\texttt{.C(); } A\, \{Q\} \end{array}}{\texttt{Env}^- \vdash \{\texttt{P}\}\, \texttt{Init(C); A}\, \{\texttt{Q}\}} \ . \tag{25}$$

Here P' stands for

$$\begin{array}{l} \exists L'\ \exists V'\ \exists d\ \exists e\ (\\ newp(d, L', V', L2) \wedge \\ newp(e, L', V', L2) \wedge \\ P(L \leftarrow L')(V \leftarrow V') \wedge \\ L = upd(L', C, d) \wedge \\ V = upd(V', d, e) \wedge \\ \neg SI(C, L', V', T, L2)). \end{array}$$

We treat the static part of a type (a class or struct) as an object in memory, which is accessible through the name of this type. The element d denotes storage location to which the name C refers and where the reference to the static part e of the type C is stored.

Note that rule (24) is auxiliary because, for a conformed environment, the information that the class C has been initialized is also contained in the precondition P. This rule optimizes the inference system and reduces the number of verification conditions by eliminating the case analysis which takes place in rule (25).

The metainstruction of storage allocation `new_instance(`x`)` is defined

by the rule:

$$\frac{Env^{-} \vdash \left\{ \begin{array}{l} \exists V0' \; \exists d \; (\\ newp(d, L, V, L2) \wedge \\ P(V0 \leftarrow V0') \wedge \\ V0 = d) \end{array} \right\} A \; \{Q\}}{Env^{-} \vdash \{P\} \texttt{new\_instance(); } A \; \{Q\}} \; . \tag{26}$$

The new location d is allocated and becomes the value of the variable $V0$.

7. VERIFICATION CONDITION REFINEMENT

Verification condition refinement is based on the algorithm which replaces lazy invariants of labelled statements by real invariants, as well as on axiomatization of the functions $CALL$ and $DELCALL$.

7.1. Lazy invariant refinement

Let X be a set of formulas. We denote the set of all formulas occuring in X of the form

$$A \Rightarrow INV(mvs, L)$$

by X_L and the set

$$\{A | A \Rightarrow INV(mvs, L) \in X_L\}$$

by $I_L(X)$.

The following algorithm refines lazy invariants $INV(\ldots)$ in the set Φ of the generated lazy verification conditions:

While there exists a label L such that the set Φ_L is not empty, replace Φ by Φ', where Φ' is the result of replacement of all occurences of expressions of the form $INV(\bar{e}, L)$ in the formulas of the set $(\Phi \setminus \Phi_L)$ by $\bigvee_{A \in I_L(\Phi)} A(mvs \leftarrow \bar{e})$.

For each path of program execution that leads to the program point that precedes the statement labelled by L there is a precondition accumulated in this point. The lazy invariant $INV(mvs, L)$ is replaced by disjunction of all such preconditions.

To illustrate the idea of lazy invariant refinement, consider the following example. Let A be a C#-light program fragment of the form

```
L: if (x<0) {
/// <a> x<0 </a>
x = x+1;
goto L;}
```

The annotation $x < 0$ is an invariant which breaks a cyclic path into linear parts. For simplicity we do not rewrite this program in terms of metavariables and use the usual Hoare rules for the statement if and the assignment. Let us prove the Hoare triple

$$\{x < 0\}\, A\, \{x = 0\}.$$

The set Φ of the generated lazy verification conditions consists of the following formulas:

$$\begin{array}{ll} x < 0 \Rightarrow INV(x, L), & (1) \\ INV(x, L) \wedge x < 0 \Rightarrow x < 0, & (2) \\ INV(x, L) \wedge \neg(x < 0) \Rightarrow x = 0, & (3) \\ x' < 0 \wedge x = x' + 1 \Rightarrow INV(x, L). & (4) \end{array}$$

The set Φ_L includes formulas (1) and (4). Then Φ is replaced by Φ' which consists of the formulas

$$\begin{array}{ll} (x < 0 \vee x' < 0 \wedge x = x' + 1) \wedge x < 0 \Rightarrow x < 0, & (2') \\ (x < 0 \vee x' < 0 \wedge x = x' + 1) \wedge \neg(x < 0) \Rightarrow x = 0 & (3') \end{array}$$

obtained as the result of the replacement of $INV(x, L)$ by

$$x < 0 \vee x' < 0 \wedge x = x' + 1$$

in formulas (2) and (3), respectively.

The proof of these formulas is straightforward.

7.2. Lazy invocation refinement

Unlike lazy invariants, lazy invocations are not eliminated at the refinement stage. Instead of that, axioms are defined for the functions $CALL$ and $DELCALL$. They allow us to prove verification conditions involving these functions.

The function $CALL$ is defined by the axiom

$$\begin{aligned} CALL(x,y,z,u,\lambda(mvs,P)) \Leftrightarrow \\ \exists v\ \exists f\ \exists w\ \exists a(P(mvs \leftarrow v) \wedge \\ invoker(f,x,y,z,v) \wedge \\ subst(w,v,f,x,z) \wedge \\ pre(f)(w,a) \wedge post(f)(u,a)). \end{aligned}$$

The tuple a defines specification parameters of the invoked function member with the identifier f. The tuple v contains the value which the metavariables have before invocation of this function member. The logical function `subst` defines whether the tuple w of metavariables is the result of argument substitution in the function member. The logical function $invoker$ defines whether f is an identifier of the function member. In turn, these functions are axiomatized in accordance with the specification [1].

The value of a delegate is a list of pairs of the form $[a,b]$ where a is the name of an invoked function member, b is the object or type on which the function member is invoked. The delegate invocation is reduced to sequential invocations of function members which belong to this list. Therefore the axioms for the function $DELCALL$ are inductive definitions, where induction is performed on the first argument of the function $DELCALL$ which is the delegate value.

$$\begin{aligned} DELCALL(rcons([x1,y1],x),z,u,\lambda(mvs,P)) \Leftrightarrow \\ CALL(x1,y1,z,u, \\ \lambda(mvs,DELCALL(x,z,u,\lambda(mvs,P)))). \end{aligned}$$

The induction base is defined by the axiom

$$DELCALL([\,],z,u,\lambda(mvs,P)) \Leftrightarrow P.$$

We use the usual list operations: $first$ (returns the first element of a list), $last$ (returns the last element of a list), $tail$ (returns a list except for the first element), $head$ (returns a list except for the last element), $cons$ (adds an element as the first element of a list) and $rcons$ (adds an element as the last element of a list).

8. ILLUSTRATIVE EXAMPLE

As an example we adopted one of the programs [7, Fig. 9] known as verification challenges. The program addresses the issues of overriding and

dynamic types. Originally written in Java, it is presented as the following C#-light program:

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

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

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

At the first sight the method `test()` seems to loop forever. But the *late binding* prevents this. The method `test()` calls the method `m()` from the class `C`, which calls the method `m()` from the class `D`, since 'this' has the runtime-type `D`.

It should be noted that the syntax of Java has not the keywords `virtual` and `override`. Thus, Java programs require careful reading to recognize the virtualization of methods. At the same time Java allows us to express the exceptions raised by methods.

8.1. Program annotations

Let us introduce some useful functions over tuples. The term $x[i]$ will stand for the i-th element of x, $subtuple(x,i,j)$ will denote the slice $x[i..j]$, and $len(x)$ returns the length of x.

Let a denote the tuple of specification parameters for `C.m`. We assume that the element $a[1]$ (denoted as $TOT(a)$) stores the type of `this`, and the tuple $subtuple(a,2,7)$ (denoted as $mvs_0(a)$) stores the initial values of metavariables from mvs. The resting parameters (i.e. $subtuple(a,8,len(a))$) are denoted as $rest(a)$. For brevity, we use the name SE instead of `System_Exception`.

Now we can describe the specifications for program methods:

$$pre(D.m)(mvs,[\,]):\ prec(D.m,mvs)$$

$$post(D.m)(mvs,[\,]):\ T(E)=SE$$

$$pre(D.Test)(mvs,[\,]):\ prec(D.Test,mvs)\wedge T(V(this))=D$$

$$post(D.Test)(mvs,[\,]):\ T(E)=SE$$

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

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

The postcondition of the method `D.Test` states that the metavariable E stores the uncaught exception of type SE raised by the method `D.m`. The specification of the virtual method `C.m` describes the case analysis. If the type of `this` is `C`, then the invocation `m();` is the recursive invocation of `C.m` itself, which leads to the endless loop $(TOT(a)=C\Rightarrow false)$. Or else, 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 result of invocation satisfies the postcondition of actual f.

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

The result of translation is the following C#-kernel program:

```
class C {
   public static void SFI() {}
   public void IFI_C() {
       Init(object);
       this.IFI_object();
   }

   virtual void m() { this.m(); }
}

class D : C {
   public static void SFI() {}
   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, x, System_Exception);
       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(); }
}
```

It should be noted that our axiomatic semantics allows us to optimize the resulting C#-kernel program. As we have seen, the proofs for method bodies are carried out under assumptions that the corresponding classes are

already initialized. Thus, we only use the explicit `Init` metainstructions in those contexts, where they cannot be omitted.

The example also illustrates the inconvenience of our approach for manual verification. The program grows in size. For example, a single string in the body of `D.m` turned into a dozen of strings. On the other hand, the process of verification condition generation can be fully automated, so the user is not forced to inspect this intermediate code at all. And the main argument is that it is considerably simpler to justify small and clear proof rules for metainstructions than one huge and confusing rule for the original operator `new`.

Note that the method `SFI` is empty for every class because the types `C` and `D` do not contain static fields. The implicit usage of the reference `this` in `IFI`-methods and in `C.m` also deserves attention.

According to translation rules, the namespace `System` is eliminated and every occurrence of the qualified name `System.Exception` is replaced by the global level name `System_Exception`.

8.3. Verification condition generation

Let us consider, for example, VC generation for the method *Test*. According to rule (1), we have to prove the triple

$$\{P_0\}\texttt{base.m();}\{T(E)=SE\},$$

with P_0 equal to

$$prec(D.Test,mvs)\wedge T(V(\texttt{this}))=D,$$

in the environment $Env^- = (Prg, D.Test, true, \{D\})$.

Rule (9), applied to the invocation `base.m()`, transforms the precondition P_0 into precondition P_1 of the form

$$CALL(m, base, [\,], mvs, \lambda(mvs, P_0))$$

and replaces the environment Env with Env^-.

By applying rules (2), (16) and (17) to the empty program with the precondition P_1, we obtain two verification conditions:

$$\begin{array}{lcl} VC_1 & : & Env^+ \vdash P_1 \wedge E \neq \omega \Rightarrow T(E)=SE, \\ VC_2 & : & Env^- \vdash P_1 \wedge E = \omega \Rightarrow T(E)=SE. \end{array}$$

8.4. Refinement and proof of verification conditions

After elementary logical simplifications, the condition VC_1 is rewritten as

$$\begin{array}{l}(CALL(m, base, [\,], mvs, \\ \qquad \lambda(mvs, prec(D.Test, mvs) \wedge \\ \qquad\qquad T(V(\texttt{this})) = D)) \wedge \\ E \neq \omega) \Rightarrow \\ T(E) = SE\end{array} \tag{27}$$

By $CALL$'s definition, the term $CALL(\ldots)$ in (27) transforms into

$$\begin{array}{l}\exists mvs_1\ \exists f\ \exists mvs_2\ \exists a \\ (T_1(V_1(\texttt{this})) = D \wedge prec(D.Test, mvs_1) \wedge \\ E_1 = \omega \wedge \\ SI(D, L_1, V_1, T_1, L2_1) \wedge \\ invoker(f, m, base, [\,], mvs_1) \wedge \\ subst(mvs_2, mvs_1, f, base, [\,]) \wedge \\ pre(f)(mvs_2, a) \wedge post(f)(mvs, a))\end{array} \tag{28}$$

By *invoker*'s definition, we have $f = C.m$ in Env^+. Then the precondition $pre(f)(mvs_2, a)$ transforms into

$$\begin{array}{l}T_2(V_2(\texttt{this})) = TOT(a) \wedge \\ mvs_2 = mvs_0(a) \wedge \\ (TOT(a) \neq C \Rightarrow \\ \quad \forall g\ \forall mvs' \\ \qquad ((invoker(g,\ m,\ \texttt{this},\ [\,],\ mvs_2) \wedge \\ \qquad\ subst(mvs', mvs_2, g, \texttt{this}, [\,])) \Rightarrow \\ \qquad pre(g)(mvs',\ rest(a))) \wedge \\ (TOT(a) = C \Rightarrow true)\end{array} \tag{29}$$

According to the definition of *subst*, the formula

$$subst(mvs_2, mvs_1, f, base, [\,])$$

can be rewritten as

$$\begin{array}{l}\exists d \\ (newp(d, L_1, V_1, L2_1) \wedge \\ L_2 = upd(L_1, \texttt{this}, d) \wedge \\ V_2 = upd(V_1, d, V_1(\texttt{this})) \wedge \\ T_2 = upd(upd(T_1, \texttt{this}, C), d, Loc(C))).\end{array} \tag{30}$$

The condition $T_1(V_1(\texttt{this})) = D$ implies $T_2(V_2(\texttt{this})) \neq C$. Then (29) turns into

$$\begin{array}{l} TOT(a) = D \wedge mvs_2 = mvs_0(a) \wedge \\ \forall\, g\ \forall mvs' \\ \quad ((invoker(g,\ m,\ \texttt{this},\ [\,],\ mvs_2) \wedge \\ \quad\ subst(mvs', mvs_2, g, \texttt{this}, [\,])) \Rightarrow \\ \quad\ pre(g)(mvs',\ rest(a))) \end{array} \tag{31}$$

By *invoker*'s definition, we have $g = D.m$ in Env^+. Then (31) transforms into

$$\begin{array}{l} TOT(a) = D \wedge mvs_2 = mvs_0(a) \wedge \\ \forall mvs'(subst(mvs', mvs_2, D.m, \texttt{this}, [\,]) \Rightarrow \\ \quad prec(D.m, mvs')). \end{array}$$

Analogously, the posccondition $post(f)(mvs, a)$ is rewritten as

$$\forall h\ (invoker(h,\ m,\ \texttt{this},\ [\,],\ mvs_2) \Rightarrow post(h)(mvs,\ [\,]))\ . \tag{32}$$

By *invoker*'s definition, we have $h = D.m$ in Env^+. Consequently, postcondition (32) turns into $T(E) = SE$. Then (28) becomes

$$\begin{array}{l} \exists mvs_1\ \exists mvs_2\ \exists a \\ (T_1(V_1(\texttt{this})) = D \wedge prec(D.Test, mvs_1) \wedge \\ prec(C.m, mvs_2) \wedge TOT(a) = D \wedge \\ \forall mvs'(subst(mvs', mvs_2, D.m, \texttt{this}, [\,]) \Rightarrow \\ \quad prec(D.m, mvs')) \wedge \\ mvs_2 = mvs_0(a) \wedge T(E) = SE)\ . \end{array}$$

This implies $T(E) = SE$ and, consequently, the verification condition VC_1 is true. The truth of VC_2 is established by analogy.

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 [11, 12, 13]. The three-level approach is applied to the language C#-light that includes all principal sequential C# constructs.

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 forward proof rules that simplifies automatic generation of verification conditions;
- the number of generated verification conditions can be considerably reduced using the environment information obtained by applying forward inference rules.

The nontrivial example presented in Section 8 illustrates these advantages of our approach.

It is suggested to work out theoretical justification of this approach including proving soundness of the axiomatic semantics with respect to operational semantics, as well as correctness of translation from C#-light into C#-kernel.

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 for C#-kernel programs and lazy verification condition refiner. It is suggested to implement a static analyzer of C#-light programs and a visualization environment.

However, verification of large C#-light programs is still a challenge. The combination of our approach with the modular approach [10] and extended static checking method [9] can be promising for this difficult problem.

REFERENCES

1. C# Language Specification. Standard ECMA-334. (2001) Web pages at `http://www.ecma-international.org/`.
2. Apt K.R., Olderog E.R. Verification of Sequential and Concurrent Programs. — Berlin a.o.: Springer-Verlag, 1991. — 450 p.
3. Börger E., Fruja N.G., Gervasi V., Stärk R. A High-Level Modular Definition of Semantics of C# // Theor. Comput. Sci. — 2004. — N 336(2/3).
4. Dubranovsky I.V. C# program verification: translation from C#-light into C#-kernel. — Novosibirsk, 2004. — (Prepr. / IIS SB RAS; N 120) (in Russian).
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. Jacobs B., Kiniry J.L., Warnier M. Java Program Verification Challenges // Proc. FMCO 2002. — Lect. Notes Comput. Sci. — 2003. — Vol. 2852. — P. 202–219.
8. Jacobs B., Poll E. Java Program Verification at Nijmegen: Development and Perspective // Lect. Notes Comput. Sci. — 2004. — Vol. 3233. — P. 134–153.
9. Leino K.R.M. Extended Static Checking: a Ten-Year Perspective // Lect. Notes Comput. Sci. — 2001. — Vol. 2000. — P. 157–175.

10. Müller P. Modular Specification and Verification of Object-Oriented Programs // Lect. Notes Comput. Sci. — 2002. — Vol. 2262.
11. Nepomniaschy V.A., Anureev I.S., Promsky A.V. Verification-Oriented Language C-light and its Structural Operational Semantics // Proc. PSI 2003. — Lect. Notes Comput. Sci. — 2003. — Vol. 2890. — P. 103–111.
12. Nepomniaschy V.A., Anureev I.S., Michailov I.N, Promsky A.V. Towards C program verification. The C-light language and its formal semantics // Programmirovanie. — 2002. — N 6. — P. 19–30 (in Russian).
13. Nepomniaschy V.A., Anureev I.S., Promsky A.V. Towards C program verification. Axiomatic semantics of the C-kernel language // Programminrovanie. — 2003. — N 6. — P. 65–80 (in Russian).
14. Nepomniaschy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V. A three-level approach to C#-light program verification // Joint NCC & IIS Bull. Ser.: Comp.Sci. — 2004. — Is. 20. — P. 61–85.
15. Oheimb D.v. Hoare Logic for Java in Isabelle/HOL // Concurrency and Computation: Practice and Experience. — 2001. — Vol. 13.
16. 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.
17. Pierik C., de Boer F.S. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts // Lect. Notes Comput. Sci. — 2003. — Vol. 2884. — P. 64–78.
18. Poetzsch-Heffter A., Muller P. A Programming Logic for Sequential Java // Proc. ESOP'99. — Lect. Notes Comput. Sci. — 1999. — Vol. 1576. — P. 162–176.
19. 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.

В.А. Непомнящий, И.С. Ануреев,
И.В. Дубрановский, А.В. Промский

НА ПУТИ К ВЕРИФИКАЦИИ C#-ПРОГРАММ: ТРЕХУРОВНЕВЫЙ ПОДХОД

Препринт
128

Рукопись поступила в редакцию 21.11.2005
Рецензент Ф. А. Мурзин
Редактор А. А. Шелухина

Подписано в печать 28.11.2005
Формат бумаги 60×84 1/16 Объем 2,3 уч.-изд.л., 2,4 п.л.
Тираж 60 экз.

ЗАО РИЦ "Прайс-курьер" 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6,
тел. (383) 330 72 02