Bull. Nov. Comp. Center, Comp. Science, 34 (2012), 23–42

Two-level mixed verification method of C-light programs in terms of safety logic*

I. S. Anureev, I. V. Maryasov, V. A Nepomniaschy

Abstract. In this paper, a formalization of the two-level mixed verification method of C-light programs, based on program specific transition systems, is suggested. Two kinds of such systems are used to formalize the method. The first kind, operational semantics specific transition systems, is used to specify the C-light mixed operational semantics. The second kind, safety logic specific transition systems, is used to specify the mixed axiomatic semantics of C-kernel into which C-light programs are translated. The formalization makes this method more technological.

1. Introduction

The two-level approach is a trend in recent verification projects. This approach suggests translation of a source program into an intermediate language program and deductive verification of the obtained program.

CompCert [12] is a verified compiler for a large subset of C called Clight. Source programs are translated into Cminor language.

Why [10] is a software verification platform which contains an intermediate language of the same name, a general-purpose verification conditions generator, Krakatoa tool for verification of Java programs, and Frama-C tool for verification of C programs. The generator can be used with many existing provers (PVS, Z3, Coq, Isabelle/HOL, and others).

Dafny [11] is a programming language and a verifier. It includes specification statements and can be considered as a modern version of Pascal or a safe version of C. Dafny programs are translated into the intermediate language Boogie [8] from which verification conditions are generated. At the proof stage, the SMT solver Z3 is used.

VCC [9] is an industrial-strength verification environment for a low-level concurrent system code written in C. VCC extends C with the design-by-contract features, like pre- and postconditions, as well as type invariants. Boogie is also used as an intermediate language for annotated C programs.

The Verisoft project [1] aims at the pervasive formal verification from the application level over the system level software down to the hardware. In this project, a subset of C called C0 is used. The Hoare logic environment is built on the top of Isabelle/HOL for a quite generic model of a sequential

*Partially supported by RFBR under Grant No.11-01-00028-a and Interdisciplinary Integration Project of SB RAS No.3.

imperative programming language called Simpl. C0 is embedded into Simpl and uses its verification environment to conduct C0 program proofs.

Our two-level C program verification method [15, 16, 17] is applied to the C-light language which is a powerful subset of the ISO C language [18]. The peculiarity of C-light is that it has formal operational semantics [16, 17]. To verify C-light programs, we first translate them to C-kernel programs. The C-kernel language is a subset of C-light. Then we generate verification conditions based on the C-kernel axiomatic semantics [15]. Our method has theoretical justification. Theorems of correct translation of C-light into C-kernel and of soundness of the C-kernel axiomatic semantics were proven [15]. Also, the algorithm of loop invariants translation from C-light programs to C-kernel ones was suggested and its correctness was proven [6].

The two-level method specifies the C memory model in sufficient detail, which leads to cumbersome verification conditions. Therefore, it requires the development of verification condition simplification techniques. In [6, 13], the separation of variables into shared and non-shared was suggested. Non-shared variables are the variables, whose values are accessible (both for reading and writing) only through their names in a program. Shared variables are the variables which are not non-shared. In correspondence with this separation, the rules of operational and, respectively, axiomatic semantics were divided into two groups: common rules for shared variables and special simplified rules for non-shared ones. The operational and axiomatic semantics, that provide rule variants for the same C statement depending on the used variable kind, got the name mixed.

This work represents further development of the mixed semantics approach.

Firstly, we suggest more precise classification of variables depending on their address access, in which three kinds of variables are sorted out: address-independent, partially address-independent, and address-dependent. Address-independent variables correspond to non-shared variables. Address-dependent variables correspond to shared variables. Partially address-independent variables correspond to an intermediate case. These are variables, whose values, obtained by special rules for non-shared variables in those program points, where access through their names takes place, coincide with values obtained by common rules for shared variables. At the same time, access to their addresses is allowed not only through their names.

Secondly, we transfer this classification on expression templates that define the expression classes. For example, we can assume that all expressions satisfied the template `*a[_]`, where `a` is a variable of the type pointer to array, are address-independent.

Thirdly, the mixed axiomatic semantics rules are replaced by the safety logic rules based on safety logic specific transition systems (SL-STS) [4], which allow us to extend the class of proved properties. For example, in the

safety logic, one can prove safety properties of non-terminating programs, whereas axiomatic semantics assumes that the program, whose partial correctness is proven, terminates.

Fourthly, the mixed operational semantics rules and, respectively, C-program states are represented by a new formalism — operational semantics specific transition systems (OS-STS) [4].

Fifthly, an algorithm is suggested which allows us to check expression templates membership of three classes mentioned above. The feature of this algorithm is that the membership condition is formulated as a safety property and to check it the deductive inference based on SL-STS is used.

2. The states of C-light programs

The definition of an operational semantics of a programming language L with the help of OS-STS consists of two stages. At the first stage, a one-to-one correspondence between L statements and expressions of OS-STS is defined. This correspondence specifies a kind of denotational semantics of L. The language of expressions which corresponds to L statements is called a projection of L on expressions and is denoted as L-exp. At the second stage, the OS-STS which specifies the operational semantics of L-exp is built. Then `L` semantics, which is called a denotational-operational semantics, is defined as a combination of L denotational semantics, which maps L programs to their projections, and L-exp operational semantics. In our case, we define the C-light denotational-operational semantics. As the corresponding C-light statements are restored immediately from C-light-exp statements, we limit ourselves to a description of C-light-exp operational semantics only.

In this section, we describe the state of OS-STS, which specifies the C-light-exp operational semantics. According to the definition of such systems [4], a state is an algebraic system of a special kind and is characterized by a set of predefined and modifiable signature symbols.

Let us define symbols which specify memory models used in this semantics. There are two such memory models, common and simplified. The common memory model is used in the common rules and the simplified one is used in the special rules.

Let `aa` and `bb` stand for `(value of a in s)` and `(value of b in s)`, respectively.

Let us define symbols which describe the common memory model. The value `v`, returned by C-light-exp expressions, has the form `(value a address b)`, where `a` is the returned value itself, `b` is its address. In the case when a C-light-exp expression is not an lvalue [18], `b = undef`. The predefined symbols `(value of _)` and `(address of _)` are used to access the components `a` and `b` of the expression `v` and have the semantics `(value (value of v) in s) = a` and `(value (address of v) in s) = b`. The

predefined symbol `(value _ address _)` has the semantics `(value (value a address b) in s) = (value aa address bb)`. The modifiable logical symbol `(_ is address)` defines the property "to be an address". The expression `(a is address)` returns `true` if and only if `aa` is an address. The modifiable symbol `(value of _)` defines the value of an address. The expression `(value of a)` returns the value stored by the address `aa`. The modifiable symbol `(type of _)` defines the type of the address value. The expression `(type of a)` returns the type of the value which is stored by the address `aa`. The predefined symbols `(address of __)`, `(shift _ by _)`, and `(shift _ to __)` define an abstract memory map. The expression `(address of a)` returns the address of the variable `a`. The expression `(shift a by b)` returns the address obtained by shift of the address `aa` by `bb` cells. The expression `(shift a to b)` returns the address of the field `b` of the structure with the address `aa`.

Let us define the symbols which describe the simplified memory model. The modifiable symbol `(value of __)` defines the variable value directly without the address access. The expression `(value of a)` returns the value of the variable `a`. The modifiable symbol `(_ [ _ ])` defines the array element value. The expression `(a [ b ])` returns the value of the element with the index `bb` of the array `aa`. The modifiable symbol `(_ . __ )` defines the structure element value. The expression `(a . b)` returns the value of the field `b` of the structure `aa`.

The next group of symbols consists of the symbols parsing the expressions in compliance with their form. The modifiable logical symbol `(__ is address-independent)` defines address-independent expressions. Usually this symbol is defined by a set of templates for address-independent expressions: `(val (a is address-independent) s) = true` if and only if the expression `a` satisfies even one of templates for address-independent expressions (is its instance). Thus the template `x`, where `x` is a simple variable, has the single instance, the variable `x` itself, and the instances of the template `(a [ _ ])`, where `a` is a variable of array type, are the expressions of the element access to the array `a`. The modifiable logical symbol `(__ is partially address-independent)` defines partially address-independent expressions. It is also defined by a set of templates. The modifiable logical symbol `(__ is address-dependent)` defines address-dependent expressions. It is defined by two previous symbols as follows: `(value (a is address-dependent) in s) = true` if and only if `(value of (a is address-independent) in s) = false` and `(value (a is partially address-independent) in s) = false`.

The predefined symbol `(type of __)` defines the expression type. The expression `(type of a)` returns the type of the expression `a`. The predefined symbol `(cast _ from __ to __)` performs type casting. The expression `(cast a from b to c)` casts the expression `aa` of the type `b` to the type `c`.

The modifiable symbols `(__ is variable)`, and `(__ is constant)` define program variables and constants, respectively.

3. Mixed operational semantics of C-light-exp

The complete set of C-light operational semantics rules, represented by labeled transition systems in Plotkin structural operational semantics style, can be found in [17]. The corresponding rules for C-light-exp, represented in the formalism of OS-STS, have the same structure, and can be obtained by simple rewriting. Therefore we represent only rules for those expressions of C-light-exp, which satisfy the classification of expressions defined in compliance with access to the values of these expressions (address-dependent, address-independent and partially address-independent).

The rules for access to a variable have the form:

```
(if a var a then (assume (* a is address-dependent))
  (assume (* a is variable))
  ((value) ::= (value (value of (address of a))
                address (address of a))))

(if a var a then (assume (* a is partially address-independent))
  (assume (* a is variable))
  ((value) ::= (value (value of a) address (address of a))))

(if a var a then (assume (* a is address-independent))
  (assume (* a is variable))
  ((value) ::= (value (value of a) address undef)))
```

The rules for access to an array element have the form:

```
(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is address-dependent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (address of (value)))
  ((value) ::= (value (value of (shift w2 by w1))
                address (shift w2 by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is partially address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value))
  ((value) ::= (value ((value of w2) [ w1 ])
                address (shift (address of w2) by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is address-independent))
```

```
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value of (value)))
  ((value) ::= (value (w2 [ w1 ]) address undef)))
```

The rules for access to a structure element have the form:

```
(if (a . b) var a b hvar w
 then (assume (* (a . b) is address-dependent))
  a (w ::= (address of (value)))
  ((value) ::= (value (value of (shift w to b))
                address (shift w to b))))

(if (a . b) var a b hvar w
 then (assume (* (a . b) is partially address-independent))
  a (w ::= (value))
  ((value) ::= (value ((value of w) . b)
                address (shift (address of w) to b))))

(if (a . b) var a b hvar w
 then (assume (* (a . b) is address-independent))
  a (w ::= (value of (value)))
  ((value) ::= (value (w . b) address undef)))
```

The common rule for an address-dependent expression assignment has the form:

```
(if (a = b) var a b hvar w1 w2 w3
 then (assume (* a is address-dependent))
  b (w1 ::= (value)) a (w2 ::= (value))
  (w3 ::= (cast (value of w1) from (* type of b) to (* type of a)))
  ((value of (address of w2)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The special rules for a variable assignment have the form:

```
(if (a = b) var a b hvar w
 then (assume (* a is partially address-independent))
  (assume (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value of (address of a)) ::= w)
  ((value) ::= (value w address undef)))

(if (a = b) var a b hvar w
 then (assume (* a is address-independent))
  (assume (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value) ::= (value w address undef)))
```

The special rules for an array element assignment have the form:

```
(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume (* (a [ b ]) is address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value of (value)))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  ((w3 [ w2 ]) ::= w4)
  ((value) ::= (value w4 address undef)))

(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume (* (a [ b ]) is partially address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  (((value of w3) [ w2 ]) ::= w4)
  ((value of (shift (address of w3) by w2)) ::= w4)
  ((value) ::= (value w4 address undef)))
```

The special rules for a structure element assignment have the form:

```
(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume (* (a . b) is address-independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value of (value)))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  ((w2 . b) ::= w3) ((value) ::= (value w3 address undef)))

(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume (* (a . b) is partially address independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  (((value of w2) . b) ::= w3)
  ((value of (shift (address of w2) to b)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

In conclusion, we consider several common rules for expressions which do not satisfy the classification defined in compliance with access to the values of these expressions.

The rule for `*` operator has the form:

```
(if (* a) var a
 then a ((value) ::= (value (value of (value of (value)))
                            address (value of (value)))))
```

The rule for `&` operator has the form:

```
(if (& a) var a then a
  ((value) ::= (value (address of (value)) address undef)))
```

The rule for constant computation has the form:

```
(if a var a then (assume (* a is constant)) ((value) ::= a))
```

4. Safety logic of C-kernel-exp

OS-STS and SL-STS are defined in [4] so that their rules are almost identical for the same programming languages construct in many cases. C-kernel-exp safety logic rules considered here are almost identical to the corresponding C-light-exp mixed operational semantics rules except that `assume*` is used instead of `assume`.

The rules for access to a variable have the form:

```
(if a var a then (assume* (* a is address-dependent))
  (assume* (* a is variable))
  ((value) ::= (value (value of (address of a))
                address (address of a))))

(if a var a then (assume* (* a is partially address-independent))
  (assume* (* a is variable))
  ((value) ::= (value (value of a) address (address of a))))

(if a var a then (assume* (* a is address-independent))
  (assume* (* a is variable))
  ((value) ::= (value (value of a) address undef)))
```

The rules for access to an array element have the form:

```
(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is address-dependent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (address of (value)))
  ((value) ::= (value (value of (shift w2 by w1))
                address (shift w2 by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is partially address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value))
  ((value) ::= (value ((value of w2) [ w1 ])
                address (shift (address of w2) by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value of (value)))
  ((value) ::= (value (w2 [ w1 ]) address undef)))
```

The rules for access to a structure element have the form:

```
(if (a . b) var a b hvar w
 then (assume* (* (a . b) is address-dependent))
  a (w ::= (address of (value)))
  ((value) ::= (value (value of (shift w to b))
                address (shift w to b))))

(if (a . b) var a b hvar w
 then (assume* (* (a . b) is partially address-independent))
  a (w ::= (value))
  ((value) ::= (value ((value of w) . b)
                address (shift (address of w) to b))))

(if (a . b) var a b hvar w
 then (assume* (* (a . b) is address-independent))
  a (w ::= (value of (value)))
  ((value) ::= (value (w . b) address undef)))
```

The common rule for an address-dependent expression assignment has the form:

```
(if (a = b) var a b hvar w1 w2 w3
 then (assume* (* a is address-dependent))
  b (w1 ::= (value)) a (w2 ::= (value))
  (w3 ::= (cast (value of w1) from (* type of b) to (* type of a)))
  ((value of (address of w2)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The special rules for a variable assignment have the form:

```
(if (a = b) var a b hvar w
 then (assume* (* a is partially address-independent))
  (assume* (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value of (address of a)) ::= w)
  ((value) ::= (value w address undef)))

(if (a = b) var a b hvar w
 then (assume* (* a is address-independent))
  (assume* (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value) ::= (value w address undef)))
```

The special rules for an array element assignment have the form:

```
(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume* (* (a [ b ]) is address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value of (value)))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  ((w3 [ w2 ]) ::= w4)
  ((value) ::= (value w4 address undef)))

(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume* (* (a [ b ]) is partially address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  (((value of w3) [ w2 ]) ::= w4)
  ((value of (shift (address of w3) by w2)) ::= w4)
  ((value) ::= (value w4 address undef)))
```

The special rules for a structure element assignment have the form:

```
(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume* (* (a . b) is address-independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value of (value)))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  ((w2 . b) ::= w3) ((value) ::= (value w3 address undef)))

(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume* (* (a . b) is partially address independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  (((value of w2) . b) ::= w3)
  ((value of (shift (address of w2) to b)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The rule for constant computation has the form:

```
(if a var a then (assume* (* a is constant)) ((value) ::= a))
```

Let us consider the rules for loops by the example of a `while` loop:

```
(if (while a invariant i do b) var a i (seq b)
 then (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume a) b (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* i) (assume (not a)))
```

5. Formal definition of partially address-independent and address-independent expressions

In introduction, we gave an informal definition of partially address-independent and address-independent expressions, and in subsequent sections we used predefined logical symbols `(__ is partially address-independent)` and `(__ is address-independent)` for the description of all expression kinds assuming that they define the corresponding kinds of expressions correctly (for a concrete program). In this section, we define them formally on the basis of an OS-STS of a special form which simultaneously executes C-light-exp expressions for two memory models, common and simplified. This OS-STS is defined in such a manner that if a program `p` is safe with respect to a precondition `q`, then the symbols `(__ is partially address-independent)` and `(__ is address-independent)` define the corresponding expression kinds correctly for the program `p`, started in the state when `q` is true.

This OS-STS is obtained from the OS-STS describing the C-light-exp mixed operational semantics in the following way.

Firstly, for all rules of access to the value of a partially address-independent expression, the continuation condition `assert` is inserted which checks the coincidence of these expressions values in two memory models. The collection of such conditions defines the safety property which should be satisfied with partially address-independent expressions.

Secondly, for each rule which uses the address access (for reading or writing), the continuation condition `assert` is inserted which checks whether this address exists. The collection of such conditions defines the safety property which should be satisfied with address-independent expressions, as no address exists for such variables.

The rules for access to a variable have the form:

```
(if a var a then (assume (* a is address-dependent))
  (assume (* a is variable))
  ((value) ::= (value (value of (address of a))
                address (address of a))))

(if a var a then (assume (* a is partially address-independent))
  (assume (* a is variable))
  (assert ((value of a) = (value of (address of a))))
  ((value) ::= (value (value of a) address (address of a))))

(if a var a then (assume (* a is address-independent))
  (assume (* a is variable))
  ((value) ::= (value (value of a) address undef)))
```

The rules for access to an array element have the form:

```
(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is address-dependent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (address of (value)))
  ((value) ::= (value (value of (shift w2 by w1))
                address (shift w2 by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is partially address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value))
  (assert (((value of w2) [ w1 ]) =
           (value of (shift (address of w2) by w1))))
  ((value) ::= (value ((value of w2) [ w1 ])
                address (shift (address of w2) by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume (* (a [ b ]) is address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value of (value)))
  ((value) ::= (value (w2 [ w1 ]) address undef)))
```

The rules for access to a structure element have the form:

```
(if (a . b) var a b hvar w
 then (assume (* (a . b) is address-dependent))
  a (w ::= (address of (value)))
  ((value) ::= (value (value of (shift w to b))
                address (shift w to b))))

(if (a . b) var a b hvar w
 then (assume (* (a . b) is partially address-independent))
  a (w ::= (value))
  (assert (((value of w) . b) =
           (value of (shift (address of w) to b))))
  ((value) ::= (value ((value of w) . b)
                address (shift (address of w) to b))))

(if (a . b) var a b hvar w
 then (assume (* (a . b) is address-independent))
  a (w ::= (value of (value)))
  ((value) ::= (value (w . b) address undef)))
```

The common rule for an address-dependent expression assignment has the form:

```
(if (a = b) var a b hvar w1 w2 w3
```

```
 then (assume (* a is address-dependent))
  b (w1 ::= (value)) a (w2 ::= (value))
  (w3 ::= (cast (value of w1) from (* type of b) to (* type of a)))
  (assert (w2 is address))
  ((value of (address of w2)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The special rules for a variable assignment have the form:

```
(if (a = b) var a b hvar w
 then (assume (* a is partially address-independent))
  (assume (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value of (address of a)) ::= w)
  ((value) ::= (value w address undef)))

(if (a = b) var a b hvar w
 then (assume (* a is address-independent))
  (assume (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value) ::= (value w address undef)))
```

The special rules for an array element assignment have the form:

```
(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume (* (a [ b ]) is address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value of (value)))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  ((w3 [ w2 ]) ::= w4)
  ((value) ::= (value w4 address undef)))

(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume (* (a [ b ]) is partially address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  (((value of w3) [ w2 ]) ::= w4)
  (assert ((shift (address of w3) by w2) is address))
  ((value of (shift (address of w3) by w2)) ::= w4)
  ((value) ::= (value w4 address undef)))
```

The special rules for a structure element assignment have the form:

```
(if ((a . b) = c) var a b c hvar w1 w2 w3
```

```
 then (assume (* (a . b) is address-independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value of (value)))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  ((w2 . b) ::= w3) ((value) ::= (value w3 address undef)))

(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume (* (a . b) is partially address independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  (((value of w2) . b) ::= w3)
  (assert ((shift (address of w2) to b) is address))
  ((value of (shift (address of w2) to b)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The rule for `*` operator has the form:

```
(if (* a) var a then a (assert ((address of (value)) is address))
  ((value) ::= (value (value of (value of (value)))
                      address (value of (value)))))
```

The rule for `&` operator has the form:

```
(if (& a) var a then a (assert ((address of (value)) is address))
  ((value) ::= (value (address of (value)) address undef)))
```

6. The algorithm of search for partially address-independent and address-independent expressions

In [7] the algorithm of static analysis was suggested to find address-independent variables (non-shared variables). In this section we describe the method of proving that the logical symbols `(__ is partially address-independent)` and `(__ is address-independent)` define the corresponding expression kinds correctly (for a concrete program). Then the algorithm of search for partially address-independent and address-independent expressions is reduced to enumeration of different possibilities of these logical symbols interpretations and to finding the correct interpretation.

The proof method is based on safety properties check described in the previous section. This check is built in the rules of SL-STS. This system consists of OS-STS rules represented in the previous section, where `assume` is replaced by `assume*`, and of the modified rules for expressions which require invariants (for loops and `goto` statements), and functions calls.

The rules for access to a variable have the form:

```
(if a var a then (assume* (* a is address-dependent))
  (assume* (* a is variable))
  ((value) ::= (value (value of (address of a))
```

```
                address (address of a))))

(if a var a then (assume* (* a is partially address-independent))
  (assume* (* a is variable))
  (assert ((value of a) = (value of (address of a))))
  ((value) ::= (value (value of a) address (address of a))))

(if a var a then (assume* (* a is address-independent))
  (assume* (* a is variable))
  ((value) ::= (value (value of a) address undef)))
```

The rules for access to an array element have the form:

```
(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is address-dependent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (address of (value)))
  ((value) ::= (value (value of (shift w2 by w1))
                address (shift w2 by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is partially address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value))
  (assert (((value of w2) [ w1 ]) =
           (value of (shift (address of w2) by w1))))
  ((value) ::= (value ((value of w2) [ w1 ])
                address (shift (address of w2) by w1))))

(if (a [ b ]) var a b hvar w1 w2
 then (assume* (* (a [ b ]) is address-independent))
  b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value of (value)))
  ((value) ::= (value (w2 [ w1 ]) address undef)))
```

The rules for access to a structure element have the form:

```
(if (a . b) var a b hvar w
 then (assume* (* (a . b) is address-dependent))
  a (w ::= (address of (value)))
  ((value) ::= (value (value of (shift w to b))
                address (shift w to b))))

(if (a . b) var a b hvar w
 then (assume* (* (a . b) is partially address-independent))
  a (w ::= (value))
  (assert (((value of w) . b) =
```

```
           (value of (shift (address of w) to b))))
  ((value) ::= (value ((value of w) . b)
                address (shift (address of w) to b))))

(if (a . b) var a b hvar w
 then (assume (* (a . b) is address-independent))
  a (w ::= (value of (value)))
  ((value) ::= (value (w . b) address undef)))
```

The common rule for an address-dependent expression assignment has the form:

```
(if (a = b) var a b hvar w1 w2 w3
 then (assume* (* a is address-dependent))
  b (w1 ::= (value)) a (w2 ::= (value))
  (w3 ::= (cast (value of w1) from (* type of b) to (* type of a)))
  (assert (w2 is address))
  ((value of (address of w2)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The special rules for a variable assignment have the form:

```
(if (a = b) var a b hvar w
 then (assume* (* a is partially address-independent))
  (assume* (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value of (address of a)) ::= w)
  ((value) ::= (value w address undef)))

(if (a = b) var a b hvar w
 then (assume* (* a is address-independent))
  (assume* (* a is variable))
  b (w ::= (cast (value of (value)) from (* type of b)
            to (* type of a)))
  ((value of a) ::= w) ((value) ::= (value w address undef)))
```

The special rules for an array element assignment have the form:

```
(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume* (* (a [ b ]) is address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value of (value)))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  ((w3 [ w2 ]) ::= w4)
  ((value) ::= (value w4 address undef)))
```

```
(if ((a [ b ]) = c) var a b c hvar w1 w2 w3 w4
 then (assume* (* (a [ b ]) is partially address-independent))
  c (w1 ::= (value of (value))) b (w2 ::= (value of (value)))
  a (w3 ::= (value))
  (w4 ::= (cast w1 from (* type of c) to (* type of (a [ b ]))))
  (((value of w3) [ w2 ]) ::= w4)
  (assert ((shift (address of w3) by w2) is address))
  ((value of (shift (address of w3) by w2)) ::= w4)
  ((value) ::= (value w4 address undef)))
```

The special rules for a structure element assignment have the form:

```
(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume* (* (a . b) is address-independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value of (value)))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  ((w2 . b) ::= w3) ((value) ::= (value w3 address undef)))

(if ((a . b) = c) var a b c hvar w1 w2 w3
 then (assume* (* (a . b) is partially address independent))
  c (w1 ::= (value of (value))) a (w2 ::= (value))
  (w3 ::= (cast w1 from (* type of c) to (* type of (a . b))))
  (((value of w2) . b) ::= w3)
  (assert ((shift (address of w2) to b) is address))
  ((value of (shift (address of w2) to b)) ::= w3)
  ((value) ::= (value w3 address undef)))
```

The rule for * operator has the form:

```
(if (* a) var a then a (assert ((address of (value)) is address))
  ((value) ::= (value (value of (value of (value)))
                      address (value of (value)))))
```

The rule for & operator has the form:

```
(if (& a) var a then a (assert ((address of (value)) is address))
  ((value) ::= (value (address of (value)) address undef)))
```

The rules for a `while`-loop in comparison with the rules of the mixed safety logic in Section 4 are modified as follows:

```
(if (while a invariant i do b) var a i (seq b) then i (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* true) i (assume a) b (assert i) (stop))

(if (while a invariant i do b) var a i (seq b)
 then (precondition ::=* true) i (assume (not a)))
```

In this case the invariant `i` is not a formula but the algorithm of checking partial address independence and address independence. This invariant has the form `(prove a)`, where `a` is a sequence of expressions of two kinds `(b is partially address-independent)` or `(b is address-independent)`, `b` is a C-kernel-exp expression. The algorithm of checking is defined by the following safety logic rules:

```
(if (prove ((a is partially address-independent) b)) var a (seq b)
 then (assume* (* a is variable))
  (assert ((value of a) = (value of (address of a)))) (prove b))

(if (prove (((a [ b ]) is partially address-independent) c))
 var a b (seq c) hvar w1 w2
 then b (w1 ::= (cast (value of (value)) from (* type of b) to int))
  a (w2 ::= (value))
  (assert (((value of w2) [ w1 ]) =
           (value of (shift (address of w2) by w1))))
  (prove c))

(if (prove (((a . b) is partially address-independent) c))
 var a b (seq c) hvar w
 then a (w2 ::= (value))
  (assert (((value of w) . b) =
           (value of (shift (address of w) to b))))
  (prove c))

(if (prove ((a is address-independent) b)) var a (seq b)
 then a (assert ((address of (value)) = undef)) (prove b))

(if (prove) then emptyseq)
```

The invariant `(prove a)` is constructed for a concrete program point `p` in the following way. In the sequence `a` we add all expressions `b` of the program `p`, which are defined in this point and which are either address-independent expressions (in this case the addition item has the form `(b is address-independent)`) or partially address-independent expressions (in this case the addition item `a` has the form `(b is partially address-independent)`).

The invariants of other loops and `goto` statements are defined similarly. The semantics of function calls is defined so that the function precondition and postcondition coincide and they represent an invariant of the same form as for loops, i. e. function computation has to keep this invariant.

The application domain of the algorithm of search for partially address-independent and address-independent expressions is limited to programs whose invariants do not contain side-effects expressions.

7. Conclusion

In this work the following new results on the two-level mixed verification method of C-light programs were obtained:

1. Three classes of expressions depending on the kind of access to address were extracted: address-independent, partially address-independent and address-dependent.
2. The rules of the C-kernel mixed axiomatic semantics are replaced by the safety logic rules based on safety logic specific transition systems [4], that allowed us to extend the class of proved properties.
3. The rules of C-light mixed operational semantics and, respectively, C-light program states are represented by a new formalism, operational semantics specific transition systems [4].
4. A new kind of a programing language semantics, denotational-operational semantics, was defined.
5. The algorithm of search for partially address-independent and address-independent expressions based on safety logic specific transition systems was suggested.

We plan to integrate the C-kernel safety logic rules into a multi-language system of program analysis and verification SPECTRUM [2, 14] based on the domain-specific language Atoment [3, 5] intended to develop program verification methods and tools.

References

[1] Alkassar E., Hillebrand M.A., Leinenbach D., Schirmer N.W., Starostin A. The Verisoft approach to systems verification // Proc. VSTTE 2008. – Lect. Notes Comput. Sci. – 2008. – Vol. 5295. – P. 209 – 224.

[2] Anureev I.S. Integrated approach to analysis and verification of imperative programs // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2011. – IIS Special Iss. 32. – P. 1–18.

[3] Anureev I.S. Introduction to the Atoment language // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2010. – IIS Special Iss. 31. – P. 1–16.

[4] Anureev I.S. Program specific transition systems // Bull. Novosibirsk Comp. Center. Ser. Computer Science. – Novosibirsk, 2012. – IIS Special Iss. 34. – P. 1–21.

[5] Anureev I.S. Typical examples of using the Atoment language // Automatic Control and Computer Sciences. – 2012. – Vol. 46, No.7. – P. 299–307.

[6] Anureev I.S., Maryasov I.V., Nepomniaschy V.A. C-programs verification based on mixed axiomatic semantics // Automatic control and computer sciences. – 2011. – Vol. 45, No.7. – P. 485–500.

[7] Anureev I.S., Maryasov I.V., Nepomniaschy V.A. Revised mixed axiomatic semantics method of C program verification // Program semantics, specification and verification: Theory and applications (PSSV 2012). Third workshop. – Nizhni Novgorod, 2012. – P. 16–23.

[8] Barnett M., Chang B.-Y.E., Deline R. et al. Boogie: A modular reusable verifier for object-oriented programs // Proc. FMCO 2005. – Lect. Notes Comput. Sci. – 2006. – Vol. 4111. – P. 364–387.

[9] Cohen E., Dahlweid M., Hillebrand M. et al. VCC: A practical system for verifying concurrent C // Proc. TPHOLs 2009. – Lect. Notes Comput. Sci. – 2009. – Vol. 5674. – P. 23–42.

[10] Filliâtre J.-C., Marché C. Multi-prover verification of C programs // Proc. ICFEM 2004. – Lect. Notes Comput. Sci. – 2004. – Vol. 3308. – P. 15–29.

[11] Leino K. R. M. Dafny: An automatic program verifier for functional correctness // Proc. LPAR-16. – Lect. Notes Comput. Sci. – 2010. – Vol. 6355. – P. 348–370.

[12] Leroy X. Formal verification of a realistic compiler // Communications of the ACM. – 2009. – Vol. 52, No.7. – P. 107–115.

[13] Maryasov I. V. The mixed axiomatic semantics method. – Novosibirsk, 2010. – (IIS SB RAS / Tech. rep. No.160). – Available at http://www.iis.nsk.su/files/preprints/160.pdf.

[14] Nepomniaschy V. A., Anureev I. S., Atuchin M. M., Maryasov I. V., Petrov A. A., Promsky A. V. C program verification in SPECTRUM multilanguage system // Automatic control and computer sciences. – 2011. – Vol. 45, No.7. – P. 413–420.

[15] Nepomniaschy V. A., Anureev I. S., Promsky A. V. Towards verification of C programs: Axiomatic semantics of the C-kernel language // Programming and computer software. – 2003. – Vol. 29, No.6. – P. 338–350.

[16] 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.

[17] Nepomniaschy V. A., Anureev I. S., Mikhailov I. N., Promsky A. V. Towards verification of C programs. C-light language and its formal semantics // Programming and computer software. – 2002. – Vol. 28, No.6. – P. 314–323.

[18] Programming languages – C. ISO/IEC 9899:1999.