

How C differs from Java for Symbolic Program Execution

Christoph D. Gladisch
gladisch@uni-koblenz.de

University of Koblenz-Landau
Computer Science Department
Universitätsstr. 1, 56072 Koblenz, Germany

Abstract. Verificationsystems, test generators, debuggers, and compilers often implement a symbolic execution framework. We have identified a small set of features in ANSI C, MISRA C, and C0 that may require special treatment or special considerations when migrating a symbolic execution framework for Java to a framework for a C dialect. For some of these language features we have developed symbolic execution rules that are implemented in the KeY-System.

1 Introduction

Verificationsystems, test generators, debuggers, and compilers often implement a symbolic execution framework to perform static analysis of a program (target program). Many of these tools already exist for Java or JavaCard as the target language for symbolic execution but haven't been yet extended for symbolic execution of C as the target language. The language C plays an important role because of the existence of a large body of legacy code, many available compilers for different platforms, and the possibility of programming *close to the hardware*.

We have identified a small set of features in dialects of the C language that may require special treatment or special considerations when migrating a symbolic execution framework for Java to a framework for C. For some of these language features we have developed symbolic execution rules that are implemented in the KeY-System.

The comparison of two programming languages is not a trivial task. On the one hand C and Java have the expressive power of a Turing machine. It is possible to replace every construct of one language by a program written in the other language. Therefore in principle any symbolic execution engine is universal if it supports one of these languages. From this *extreme* point of view C and Java can be regarded as equal.

On the other hand C and Java can be regarded as completely different languages with no language constructs in common. For instance, arithmetic expressions in both languages differ, because of differences of the integral types that cause different overflow semantics. Also the semantics of statements like the if-statement or while-statement are different, because in Java they can be abruptly terminated by exceptions in contrast to C. Neither of the two views is helpful when a symbolic execution framework is to be extended for the use with another programming language.

The first approach, where the constructs of one language are simulated by the other language, is in particular not useful when a (semi-) interactive system is extended or reports are shown to the user where she sees the source code. We therefore use a perspective for the comparison that lies between these extreme points of view: we regard differences with respect to the functionality and capabilities of a typical symbolic execution framework like the one implemented in the KeY-System. Two components of the framework and respectively two parts of the language semantics are distinguished: the *static* and the *dynamic* part. We present a small set of constructs in C that are

special with respect to Java concerning the dynamic part. The existence of these features depends on the particular dialect of C and is given by the following list:

- reference and dereference operators (ANSI C, MISRA C, C0)
- assignment of structures (ANSI C, MISRA C, C0)
- evaluation order of expressions and function arguments (ANSI C only)
- explicit object deallocation (ANSI C only)
- unions, pointer arithmetics, and type casts (ANSI C only)
- goto-statement (ANSI C only)

2 About Java and JavaCard

Java [12] is an imperative object oriented programming language with statements, expressions, declarations, and types similar to those of C++. JavaCard [24] can be seen as a subset of Java. The specification of both languages is given in natural language. The syntax and semantic of the languages are unambiguous and fully specified. The languages have strong typing rules in contrast to C and C++. Their object oriented features are interfaces, inheritance, virtual methods, overloading, dynamic object creation and scoping but in contrast to C++ there is multiple inheritance of interface but not of classes. It is also not possible to write purely procedural programs as in C and C++. Common primitive types of Java and JavaCard are: `byte`, `short`, and `boolean`.

Java's primitive data types also include `int`, `long`, `double`, `float`, and `char`. In contrast to JavaCard Java supports multidimensional arrays, and dynamic class loading, threads (concurrency), introspection and Strings as build-in APIs. As a consequence, the class `Object` which is the super class of all other classes has less methods in JavaCard than in Java. For the programming of SmartCards JavaCard supports transactions by which it can be assured that a SmartCard stops execution in a valid state in case of abrupt termination (e.g., due to powerloss).

3 About C Dialects

3.1 ANSI C

The programming language C is a procedural language whose main building blocks are functions. When C was invented and first implemented in the 70'ies and 80'ies, there was no official specification but only the book *The C Programming Language* that was used as "existing practice". The creation of the first standard specification began in 1983 [1], was finalized in 1989, and adopted by ANSI and ISO (International Standard Organization) in 1990 [3]. Since a lot of legacy code existed already at this time that was based on different interpretations of the C "practice" the specification was designed to leave room for these interpretations. Therefore much of the language is intentionally undefined, unspecified or implementation dependend in contrast to the specification of the Java language — clearly, ANSI C has ambiguous semantics. This version of C is commonly called C89. In 1996 and 1999 two more versions C96 [4] and C99 [21] were created.

The specifications are written in natural language and are hard to understand. For instance, "members of the standardization committee and other distinguished researchers participating in the discussions often give contradictory answers when asked about the intended semantics of surprisingly small programs" (*A case study in specifying the denotational semantics of C* by N.Papaspyrou [19]). Also many popular books, e.g. *C: The Complete Reference*, contain "several hundred errors" as reported in [22] and [10]. Similar reviews of many other books are given by ACCU (Association of C & C++ Users) in [11].

3.2 MISRA C

From what has been said above it seems to be hard to write safe and secure software in C. The MISRA (Motor Industry Software Reliability Association) consortium has therefore developed a manual with 141 rules which restrict the use of C96 resulting in the subset of C called MISRA C. “The MISRA consortium is not intending to promote the use of C in the automotive industry. Rather it recognizes the already widespread use of C, and this document seeks only to promote the safest possible use of the language.” [15]. In the following we list some of the important rules of the manual to give an impression of the differences between MISRA C and C96 and at the same time report some of the features of C96 that are unspecified, undefined, or implementation dependent. (Rule ids are given in parentheses.)

- The value of an expression shall be the same under any order of evaluation that the ANSI C Standard permits. (12.1 and 12.2)
- Directly or indirectly recursive functions are not allowed (16.2). Functions have a constant number of arguments (16.1) and may only have a single point of exit via the `return` statement which is at the end of the function (14.7).
- Dynamic heap memory allocation (and deallocation) shall not be used (20.4)
- Pointer arithmetic is restricted to pointers that address elements of the same array. (17.1)
- Use of `goto` (14.4), `continue` (14.5) and the comma-operator (12.10) is prohibited and the use of the `break` statement in loops and in the `switch` statement is restricted.
- Unions shall not be used. (18.4)
- The way expressions may be constructed is restricted regarding the types of the expression arguments by the rules (10.1 and 10.2); and type casts between pointers are restricted to those that are independent from the memory representation (11.1 to 11.5).

3.3 C0

C0 [14, 7] is a C-like language that has a single and formally defined semantics. It was developed in the Verisoft project [25] as the target language for verification using Isabel/HOL [20]. C0 is much simpler than ANSI C. Some of the interesting restrictions of C0 are:

- no side effects in expressions (e.g. operators like `i++`, `i--`)
- no functions as sub expressions
- size of arrays is statically fixed at compile time
- exactly one return statement in each function as last statement
- only one scope for variables inside a function
- evaluation of expressions in the standard order defined by post order traversal of syntax trees
- no pointers to local memories or to functions
- pointers are strictly typed
- two’s complement representation of integral types

4 The Differences

4.1 Distinguishing the static and the dynamic part of the languages

The static component of the execution framework, also called program context, consists of tables typically generated by a compiler. The tables, hold information about type or class declarations,

program variable declarations, and declarations of functions or methods. Besides the tables we also regard the name resolution or binding component as part of the static component. The binding component maps identifiers of program variables and functions or methods depending on the context of usage of the identifier to the correct declarations.

The dynamic component consists of a language in which the operational semantics of the programming language is defined and an execution framework that can execute a program based on these definitions. This component has typically means to express the current state of program variables, a path condition, a stack of executed methods and other program blocks, and a program counter. In endogenous logics like Temporal Logic this counter is a pointer or index to the current program instruction. In exogenous logics like Hoare Logic or Dynamic Logic the program is explicit in the formulas.

The differences of C with respect to Java that we mainly focus on are related to the dynamic parts of the languages. We assume that macros and preprocessor directives are resolved by a preprocessor and are not part of the language.

4.2 How C differs from Java regarding the static part of the languages

We briefly address the differences between some static parts of the languages. Java has primitive types, arrays, classes and interfaces with access modifiers and C has similar – but in detail different – elementary types, arrays, typedefs, structs, unions, bitfields, and pointers. The type system of C can be mapped to the type systems of Java by an encoding such that different types can be distinguished and relations like “pointer of” and “substructure of” can be evaluated based on the encoding. Such an encoding is however very obfuscated, we therefore recommend an implementation of the type system that is independent from Java’s type system.

4.3 How C differs from Java regarding the dynamic part of the languages

We have identified a small set of features of the C dialects that likely require special treatment for the formal definition of the language semantics and for the definition of symbolic execution rules with respect to those for Java. Features that exist only in one of the dialects are marked in parentheses.

Reference and Dereference Operators. The reference operator ‘&’ returns the pointer to a program variable (or C function) and the dereference operator ‘*’ maps a pointer to the value at the location indicated by the pointer. Similar to the dereference operator is Java’s dot operator ‘.’ for accessing members or methods of objects (target objects). Techniques for solving the aliasing problem in Java can be reused here. However, the dereference operator in ANSI C and MISRA C is more powerful than the dot operator in Java, because the dereference operator can be used together with pointer arithmetic. Array indexing is a commonly used application of this technique.

An important difference between the languages results from the combined usage of the reference and dereference operators in C. In this case program variables in C must be treated like objects in Java. Dereferencing a pointer to a program variable in C behaves like accessing the member of an object in Java.

In ANSI C a pointers to a local program variables can be obtained. The pointer to a local program variable becomes invalid when the function in which the program variable is declared terminates. This can be solved by bookkeeping of the local program variables and upon termination marking the program variables as *invalid*.

Assignment of Structures. The explicit or implicit (e.g., by passing arguments to a function) assignment between variables of a structure type has copy-by-value semantics. This causes a component-wise assignment between the corresponding members of the structures. If a constant size array is declared within a structure, then the elements of the array are copied individually to the corresponding array positions of the array that is embedded as a member of the other structure. Thus an assignment in C may represent more than one assignment in contrast to assignments in Java. C0 has copy-by-value semantics also for not embedded arrays.

When using the reference and dereference operators in combination with structures the aliasing problem becomes more complicated. A structure can indirectly not only be manipulated by dereferencing a pointer to the structure and assigning a value to it but also by dereferencing pointers of *members* of the structure and assigning new values to them.

Evaluation Order of Expressions and Function Arguments (ANSI C only). The evaluation order of expressions and of arguments in function calls is unspecified. It is also possible that some subexpressions are not evaluated at all. The evaluation order of expression is not specified in MISRA C but the set of expressions that maybe used in MISRA C is restricted to those expressions that evaluate equally under any order of execution of the subexpressions.

The evaluation of an expression may yield different results upon different orders of evaluation if the subexpressions have side-effects or if one subexpression causes the abrupt termination of the program. An expression has a side-effect if it contains an implicit assignment and thus a change of the execution state during the evaluation of the expression. The evaluation order of expressions with side-effects is not *critical* if no read access to the implicitly modified memory is performed during the expression evaluation.

Unions, Pointer Arithmetic, and Type Casts (ANSI C only). The common concept of unions and arbitrary type casts is that one sequence of bytes in memory can be interpreted in different ways.

The *interpretation* of the memory depends on the type or data structure that is virtually laid over the sequence of bytes in the memory. Translations between arbitrary interpretations of the memory require complicated arithmetic. Aliased program variables of different types are seldom used except when the boundaries of the subsequences of the addressed memory are aligned.

The problem of interpreting non-aligned memory blocks is particularly hard when pointer arithmetic is used. A non-critical usage of pointer arithmetic is array indexing.

Even if memory blocks are not interpreted for different types but a runtime type is changed by a type cast or union this causes a problem for quantification over objects of one type. In verification systems it is important to restrict quantification only to created objects of one type. The usage of unions and type casts can change the type dependent sets of created object. This problem occurs for instance when `malloc` is called — the type of the returned pointer is `*void` and is usually type casted at some time.

Explicit Object Deallocation (ANSI C only). In ANSI C memory can be deallocated using the API function `free` of the standard C library. A deallocated pointer is *invalid*. When the location indicated by the invalid pointer is subsequently modified via the dereferenced pointer this results typically in modification of subsequently allocated heap memory. In general however the behavior is unspecified.

Due to this feature, several kinds of pointers have to be distinguished. Only *dynamic root pointers*, which are the pointers returned from the allocation function, may be used for deallocation. Pointers to global program variables and to local program variables, called *static pointers*, and *dynamic non-root pointers* like pointers to members of dynamically allocated structures and to elements of dynamically allocated arrays must not be deallocated; otherwise the program behavior is undefined (usually the program will terminate abruptly).

Special care has to be taken if a dynamically allocated structure or array is deallocated, because pointers to members of the structure or to array elements become invalid as well.

goto-Statement (ANSI C only). A symbolic execution framework that can handle abrupt termination in Java caused by `break`, `continue`, `return`, or `throw` can also handle the respective jump statements of C. All these statements have in common that the program execution is continued at the beginning or end of the block or an outer block in which the statement occurs. However the goto statement that exists only in ANSI C allows arbitrary program execution jumps within a function.

4.4 Border-Issues

Volatile Program Variables (ANSI C and MISRA C). A program variable may change its value without being explicitly changed by the program that is symbolically executed. For instance, when a reference to a program variable is passed to an *external program* via an API function the external program may change the program variable concurrently, e.g. hardware drivers, system clock, etc. Handling of program variables that may be modified concurrently by an external program requires techniques for handling concurrency. These techniques are also required when full Java is supported, so that we don't classify this as a special feature of C.

The modifier `volatile` in C is used to prevent an optimizing compiler from performing certain optimizations involving volatile program variables. In Java the modifier `volatile` ensures memory synchronization of threads that have shared usage of the volatile program variable.

Function pointers (ANSI C and MISRA C). Invocations of dereferenced function pointers can be handled by techniques for handling polymorphism in Java. A function call through a dereferenced function pointer can be replaced by an if-cascade over the value of the function pointer and with concrete function calls in the then-part. The problem with dereferencing a function pointer obtained from an *external* API function is no harder than with polymorphism when an object is returned from an API method and it belongs to a subclass of the return class type of the method.

5 The KeY-System

The KeY-System [6] is a mixed interactive and automatic verification- and testgeneration system for Java. It is based on an instance of Dynamic Logic [13] called JavaCardDL (JDL) and a sequent calculus that interleaves rules for theorem proving and symbolic execution of Java. The symbolic execution rules handle 100% of JavaCard and some additional features of Java. Dynamic Logic is an extension of First-order Logic where formulas φ can be *prepended* by the modal operators $\langle p \rangle$ and $[p]$ for every program p . The formula $\langle p \rangle \varphi$ means that φ holds after the execution of the program p so that the implication $\phi \rightarrow \langle p \rangle \varphi$ corresponds to the Hoare triple $\{\phi\}p\{\varphi\}$.

Different program states are realized by different first-order interpretations of function symbols (including constants). Therefore program variables are modeled in the logic as constants, an expression like $o.a$, that accesses an object attribute, is modeled as the term $a(o)$, and in case of an array access $o.a[i]$ the corresponding term is $\llbracket(a(o), i)\rrbracket$.

JDL has the additional modal operator ' $\{\}$ ' called update. Updates are assignments between terms (not expressions) and are therefore free of side-effects. This allows a powerful calculus for simplifying and merging sequences of elementary updates like $\{t_1 := t_2\}\{t_3 := t_4\}$ into parallel updates $\{t_1 := t_2 \parallel t_3 := t_4\}$. The latter can be seen as a table from locations to values where the aliasing problem is handled. A conditional update $\{\text{if } c : U\}$ has only the effect of the update U when c is true. An arbitrarily long sequence of elementary updates $\{t_{a_1} := s_{a_1}\} \dots \{t_{a_n} := s_{a_n}\}$ over an ordered universe with $a_1 < \dots < a_n$ can be written as the quantified update $\{\text{for } x : t_x := s_x\}$. Formulas with the modal operators $\{\}$, $\langle \rangle$, and $\llbracket \rrbracket$ can be in any case translated into pure first-order formulas. In symbolic execution rules we abbreviate $\langle \rangle \varphi$ and $\{\} \varphi$ by $\langle \rangle$ and $\{\}$ respectively. Thus the symbolic execution of a side-effect free assignment is handled by the rule:

Replace $\langle a=b \rangle$ by $\{a := b\}$

JDL uses “relativized” constant domain semantics: an infinitely large pool of objects that may be created at runtime exists at any time of program execution but quantification is implicitly restricted to objects that are marked as created by the non-rigid index terms next_T for every type T . The semantic is that objects of type T identified by the object identifier terms $\text{obj}_T(i)$ are created for $i < \text{next}_T$ and not created for $i \geq \text{next}_T$. Quantification over created objects of type T is therefore realized by $\forall i. \forall o. ((i < \text{next}_T \wedge o = \text{obj}_T(i)) \rightarrow \Phi(o))$ and is abbreviated as $\forall o : T. \Phi(o)$. Dynamic object creation is therefore handled by the following rule:

Replace $\langle a = \text{new } T() ; \rangle$ by $\{a := \text{obj}_T(\text{next}_T) \parallel \text{next}_T := \text{next}_T + 1\}$

6 Special Symbolic Execution Calculus Rules for C

We have identified features in the C dialects that require special treatment. Additionally we have developed a symbolic execution calculus for C0 and for some features of ANSI C. Since this paper is about symbolic execution, we do not give advice on how to handle undefined behavior or abrupt program termination because the handling depends on the *purpose* of the symbolic execution framework.

Reference and Dereference Operators. The combined usage of the reference and dereference operators requires that program variables in C must be treated like objects in Java. Dereferencing a pointer to a program variable in C behaves like accessing the member of an object in Java. Therefore the dereference operator is a non-rigid function symbol ' $*$ ' (for every elementary type) that holds the value of the program variable and the program variable itself is a constant in the logic that represents the address of the program variable (etp: expression to pointer). Implicit axioms are required that ensure that all program variables (i.e., their addresses) are distinct. Then after the address of a program variable is determined it is finally dereferenced (ett: expression to term). The translation from expressions consisting of program variables, access to attributes, access to array elements, reference operators, and dereference operators to terms is given by the rewrite rules:

$$\begin{array}{ll}
\text{ett}(X) \rightsquigarrow *(\text{etp}(X)) & \text{etp}(X.m) \rightsquigarrow \cdot^m(\text{etp}(X)) \\
\text{ett}(\&X) \rightsquigarrow \text{etp}(X) & \text{etp}(*X) \rightsquigarrow \text{ett}(X) \\
\text{ett}(X[i]) \rightsquigarrow \square(\text{etp}(X), \text{ett}(i)) & \text{etp}(\mathbf{a}) \rightsquigarrow a
\end{array}$$

where \cdot^m is the corresponding function symbol for each member m , \square is the binary function symbol that represents addresses of array elements, and \mathbf{a} is a program variable identifier. Examples of applying the syntactic transformation `ett` on expressions are:

Expression	\mathbf{a}	$\&\mathbf{a}$	$\mathbf{a}.m$	$*\mathbf{a}$	$(*\mathbf{a}).m$	$*(\mathbf{a}.m)$	$\mathbf{a}[i]$
Term	$*(a)$	a	$\cdot^m(a)$	$*(*(a))$	$\cdot^m(*(a))$	$*(\cdot^m(a))$	$\square(a, *(i))$

Assignment of Structures. An assignment between variables of a structure type can be handled by *unfolding* (`unf`) the assignment into a finite set of assignments (here updates) between the respective members of the structures or array elements in case of an embedded constant size array in the structure. The unfolding of the assignment has to take place over the pointers of the members of the structure (resp. array elements). Therefore the final dereferenciation of the expression must be delayed until assignments between program variables of basic types are obtained.

Replace $\langle X=Y \rangle$ by $\{\text{unf}(X = Y)\}$

$$\text{unf}(X = Y) \rightsquigarrow \begin{cases} \text{unf}(X.m_1 = Y.m_1), \dots, \text{unf}(X.m_n = Y.m_n), & \text{if } X \text{ has a struct type} \\ \text{unf}(X[0] = Y[0]), \dots, \text{unf}(X[n] = Y[n]), & \text{if } X \text{ has an array type}^* \\ \text{ett}(X) := \text{ett}(Y), & \text{if } X \text{ has an elementary type} \end{cases}$$

*) The enumeration of elementary updates between array elements can be replaced by the single combined quantified and conditional update: $\{\text{for } i : \text{if } 0 \leq i \leq n : \text{unf}(X[i] = Y[i])\}$

Evaluation Order of Expressions and Function Arguments (ANSI C only). KeY's symbolic execution calculus for Java eliminates side-effects by the program transformation rule:

Replace $\langle \text{stmt}(\text{expr}_1, \dots, \text{expr}_n); \rangle$ by
 $\langle \text{tmp}_1 = \text{expr}_1; \dots; \text{tmp}_n = \text{expr}_n; \text{stmt}(\text{tmp}_1, \dots, \text{tmp}_n); \rangle$

where `stmt(expr1, ..., exprn)` is a (compound) statement with expressions `expr1, ..., exprn` that are replaced by the temporary program variables `tmp1, ..., tmpn`. In case of *critical* side-effects (see section 4.3) any permutation of the statements `tmp1 = expr1; ...; tmpn = exprn;` may or must be considered depending on the purpose and overall semantics of the symbolic execution framework.

Type casts (ANSI C). A commonly used feature in C is the cast between a pointer of type `*void` and a different pointer. A simple approach is to associate every pointer with a *runtime* type as it is done for Java and treat the type `*void` like the class type `Object` in Java regarding type casts.

A logic and a calculus for handling *type changes of objects* (that would change the relativized constant domain) or for handling mappings or *interpretations* between byte sequences and arbitrary types is very involved and not covered here. An idea for the first problem is however to extend the allocation and deallocation mechanism and add a reallocation mechanism. Changes of the runtime type could then be realized by delayed allocation, deallocation and reallocation.

Explicit Object Allocation and Deallocation (ANSI C only). In order to distinguish different kinds of pointers, the type system is extended by the disjunct types `Static`, `Root`, and `NonRoot` for an additional classification of the terms:

$$\begin{aligned} & \text{obj}_T(n) : \text{Root} \\ & \cdot^m(x), \llbracket(x, i) : \text{NonRoot} \\ a, \cdot^m(a), \llbracket(a, i) : \text{Static} \end{aligned}$$

where $n, i : \text{Nat}$, $x : \text{Root} \cup \text{NonRoot}$, and a is a global or local program variable.

The logic is extended by the non-rigid function $\text{cr} : \text{Root} \rightarrow \{\text{tt}, \text{ff}\}$ that indicates whether the *root* object is created (valid pointer) and the rigid function $\text{root} : \text{Root} \cup \text{NonRoot} \rightarrow \text{Root}$ that returns the root object from a non-root object as defined by the axiom schema for every attribute m :

$$\begin{aligned} \text{root}(\cdot^m(X)) &= \begin{cases} X, & \text{if } X : \text{Root} \\ \text{root}(X), & \text{if } X : \text{NonRoot} \end{cases} \\ \text{root}(\llbracket(X, i)) &= \text{root}(\cdot^m(X)) \end{aligned}$$

For a subset of C that excludes casting of runtime types allocation can be handled by the rule:

$$\begin{aligned} & \mathbf{Replace} \langle \mathbf{a} = (*\mathbf{T})\mathbf{malloc}(\mathbf{sizeof}(\mathbf{T})); \rangle \mathbf{by} \\ & \{*(a) := \text{obj}_T(\text{next}_T) \parallel \text{cr}(*a) := \text{true} \parallel \text{next}_T := \text{next}_T + 1\} \end{aligned}$$

where $(*\mathbf{T})\mathbf{malloc}(\mathbf{sizeof}(\mathbf{T}))$ is regarded as a single expression. The quantified formula $\forall o : T_1. \Phi(o)$ is now an abbreviation for:

$$\forall i. \forall o : T_1. ((\text{root}(o) = \text{obj}_{T_2}(i) \wedge i < \text{next}_{T_2} \wedge \text{cr}(\text{obj}_{T_2}(i)) = \text{tt}) \rightarrow \Phi(o))$$

Deallocation of *Root*-pointers is handled by the rule:

$$\mathbf{Replace} \langle \mathbf{free}(a); \rangle \mathbf{by} \{ \text{cr}(*a) := \text{ff} \}$$

Deallocation is only allowed with dynamic pointers that were obtained by the allocation function and whose referenced objects are created. The handling of the dereference operator must be extended such that the *createdness* of the object that the pointer is dereferenced to is checked. The createdness of a non-root object is the same as of its root object that can be obtained through the function *root* (defined above). How the violation of these rules is handled depends on the purpose and semantics of the symbolic execution framework.

7 Related Work and Conclusion

The *Rationals* [2] and [5] contain much of the text of the ANSI C specifications and discussions about thoughts and intentions of the standardization committee. A formal specification of C89 is given in the PhD thesis [18] by N.Papaspyrou. The specified language “differs slightly from ANSI C”. The specification must leave some space for further instantiation of the language or leave some aspects unspecified because of what has been said above. A dynamic logic for C0 is developed in [23]. For the theorem prover Nqthm the C semantic has been specified and is documented in the technical reports [16, 8]. A derivation of verification rules for C from operational definitions can be found in [17]. Finally, a cross-reference between C and Java is provided by [9].

The situation when working with the semantics of C is different from the situation when working with Java or JavaCard because of the complicated specification and vague semantics. It may be necessary to choose a particular subset of ANSI C or allow the instantiation of the symbolic execution framework for particular semantics.

References

1. American National Standards Institute. *American National Standard for Programming Language: C, ANSI X3.159-1989*, 1989.
2. American National Standards Institute. *Rationale for American National Standard for Information Systems: Programming Language C*, 1989.
3. American National Standards Institute. *ANSI/ISO 9899-1990, American National Standard for Programming Language: C*, 1990. Revision and redesignation of ANSI X3.159-1989.
4. American National Standards Institute. *Technical Corrigendum Number I to ANSI/ISO 9899-1990 American National Standard for Programming Language: C*, 1994.
5. American National Standards Institute. *Rationale for International Standard, Programming Language, C*, April 2003.
6. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.
7. C. Berg, M. Klein, D. Leinenbach, and W. J. Paul. Formal operational semantics of C0. Internal Technical Report 8, Universität des Saarlandes, May 2004. <http://www.verisoft.de> (accessed 23rd August 2005).
8. C. Cook, E. Cohen, and T. Redmond. A formal denotational semantics for C. Internal technical report, Trusted Information Systems, 1994.
9. F. F. Chew. *The Java/C++ Cross-Reference Handbook*. Prentice Hall PTR, Upper Saddle River, NJ, October 1997.
10. C. D. W. Feather. *The Annotated Annotated C Standard*. <http://www.lysator.liu.se/c/schildt.html> (accessed on 22 February 2007). Review of *The Annotated ANSI C Standard* by Herbert Schildt.
11. F. Glassborow. Book review c:the complete reference. <http://www.accu.informika.ru/bookreviews/public/> (accessed on 14th April 2007).
12. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
13. D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.
14. D. Leinenbach. Die sprache C0. Internal Technical Report 2, Universität des Saarlandes, June 2004. <http://www.verisoft.de> (accessed 23rd August 2005).
15. MISRA Consortium and G. McCall. *MISRA-C:2004*. MIRA Limited, Nuneaton Warwickshire UK, 2nd edition, 2004.
16. M. Norrish. An abstract dynamic semantics for C. Internal technical report, Trusted Information Systems, October 1994.
17. M. Norrish. Derivation of verification rules for C from operational definitions. In J. Grundy, J. von Wright, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics: TPHOLs96*, number 1, pages 69–75, 1996.
18. N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Athens, Greece, 1998.
19. N. Papaspyrou. A case study in specifying the denotational semantics of C. Athens, Greece, 1999. <http://citeseer.ist.psu.edu/papaspyrou99case.html> (accessed 15th February 2006).
20. L. C. Paulson. *Isabelle: a generic theorem prover*. LNCS 828. Springer, 1994.
21. H. Schildt. *C: The Complete Reference 4th Edition*. McGraw-Hill Osborne Media, 4th edition, January 2002.
22. P. Seebach. *C: The Complete Nonsense*. http://herd.plethora.net/~seeb/c/c_tcr.html (accessed on 14th April 2007). Review of *C: The Complete Reference 4th Edition* by Herbert Schildt.
23. W. Stephan. Verification of pointer programs in dynamic logic. Internal Technical Report 7, Universität des Saarlandes, December 2003. <http://www.verisoft.de> (accessed 23rd August 2005).
24. Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, Oct. 1997.
25. The Verisoft Project. Website at <http://www.verisoft.de>.