Combining ACSL Specifications and Machine Code

P.A. Putro <pavel.putro@ispras.ru>
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000 Russia

Abstract. When developing programs in high-level languages, developers have to make assumptions about the correctness of the compiler. However, this may be unacceptable for critical systems. As long as there are no full-fledged formally verified compilers, the author proposes to solve this problem by proving the correctness of the generated machine code by deductive verification. To achieve this goal, it is required to combine the pre- and postcondition specifications with the machine code behavior model. The paper presents an approach how to combine them for the case of C functions without loops. The essence of the approach is to build models, both machine code and its specifications in a single logical language, and use target processor ABI to bind machine registers with the parameters of the high-level function. For the successful implementation of this approach, you have to take a number of measures to ensure the compatibility of the high-level specification model with the machine code behavior model. Such measures include the use of a register type in the high-level specifications and the translation of the pre- and postconditions into the abstract predicates. Also in the paper the choice of logical language for building models is made and justified, the most suitable tools for implementing the approach of merging specifications are selected and the evaluation of the system of deductive verification of machine code built on the basis of the proposed approach is made using test examples obtained by compiling C programs without loops.

Keywords: deductive verification; formal methods; machine code; ACSL

DOI: 10.15514/ISPRAS-2018-30(4)-6

For citation: Putro P.A. Combining ACSL Specifications and Machine Code. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 4, 2018. pp. 95-106. DOI: 10.15514/ISPRAS-2018-30(4)-6

1. Introduction

The paper presents a step forward towards the creation of a tool capable of proving the correctness of machine code based on the formal specification of a function for a high-level language [1]. Such a tool will allow to avoid the assumption about the correctness of the compiler by verification of the generated code regarding specification of source code functionality. The only way in which the correctness analysis of machine code is not necessary is to create a fully formally verified compiler [2].

However, the existing developments in the field of formally verified compilers [3] now do not allow using all the possibilities of existing unverified analogs, for example, GCC [4]. This work is necessary for the implementation of an alternative approach – deductive verification [5] of compiler products, the correctness of which has not been proven. Using this approach will allow you to safely use the already created software.

Different approaches to formal specification and building a model of machine code behavior were proposed in different machine code verification projects. Here, the formal specification of a function or a sequence of machine code instructions shows the pre- and postconditions for a function and the behavior model describes mathematical and logical state change formulas. The paper discusses an approach to combining ACSL [6] specifications of the C language with the machine code of the PowerPC e500mc processor obtained by compiling these functions. The choice of the target language is caused by the fact that most high-critical system software like operating system kernels is written in C. While the very high-level languages support a variety of protective mechanisms – such as the prohibition of pointers or checks when casting, the C language is designed for maximum performance by allowing the programmer to interact directly with the memory.

Proof of critical code sections by deductive verification methods can improve the reliability of such systems. In the pursuit of performance, compilers try to make the most of the capabilities of the target processor. Machine code produced by compilers can be extremely difficult for manual verification and specification because the compilation disappears all the information about the names of variables and even the order of execution of commands may be different than in the original program. Only the pre- and postconditions for a particular function remain unchanged. Automatic combination of C-level specifications with the logical model of machine code will allow you to check its correctness in a fully automatic mode.

2. Machine code representation

The specification of machine code instructions in logical languages is a complex and lengthy process. Often, the appearance of the function behavior model specification in this language is very different from that provided in the processor specification. In addition, the lack of special tools makes it difficult to debug such models. To solve these problems, the author proposes to use the NML language, together with the MicroTESK tool [7]. The NML language contains special structures and data types to simplify the modeling of the hardware. The MicroTESK toolset includes universal disassembler of the machine code by the NML language and the NML to SMT-LIB [8] translator.

Fig. 1 shows the cmpl operation specification from the official documentation for PowerPC e500 core family [9] processors and fig. 2 shows its NML version. From here, you can see that the NML language allows you to fully describe processor instructions, including their representation in Assembly language and machine code. In addition, the use of the NML language as the basis for the representation of

machine code will allow to reuse all NML models, developed by the MicroTESK development team for the purposes of testing of microprocessors.

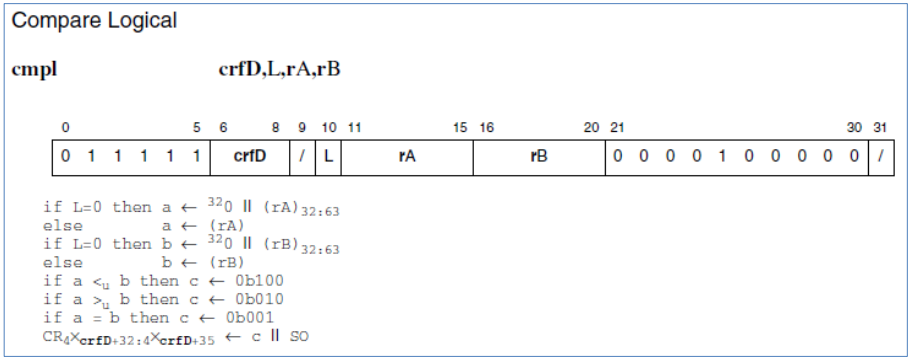

Fig. 1. CMPL official specification

```
op cmpl (crfD: CRFD, L: BIT, ra: R, rb: R)
  init = {
    XO_10 = coerce(card(10), 0b0000100000);
    OPCD = coerce(card(6), 0b011111);
  }
  syntax = format("cmpl %d, %d, %s, %s", crfD, L, ra.syntax, rb.syntax)
  image  = format("%6s%3s%1s%1s%5s%5s%10s%1s", OPCD, crfD, L, "0", ra.image, rb.image, XO_10, "0")
  action = {
    if L == coerce(BIT, 0) then
      if ra < rb then
        temp = 0b001;
      endif;
      if ra > rb then
        temp = 0b010;
      endif;
      if ra == rb then
        temp = 0b100;
      endif;
      CR<(coerce(card(5),crfD)*4+2)..(coerce(card(5),crfD)*4)> = coerce(card(3), temp);
      CR<coerce(card(5),crfD)*4+3> = XER_SO;
    endif;
  }
```

Fig. 2. CMPL NML specification

3. ACSL specifications representation

3.1 ACSL specifications translation

As a logical language, in which ACSL specifications will be translated, the author suggests using the WhyML language [10]. The Why3 tool designed to analyze this language allows you to apply many useful transformations and optimizations. It also allows you to translate WhyML code into logical code for many different provers. In addition, the task of translating ACSL specifications into WhyML code has already been solved by the Jessie plugin [11] for Frama-C [12]. In the course of research [1], it was established that the use of the plugin Jessie directly, not suitable for the tasks of machine code analysis.

Jessie plugin makes a number of simplifying assumptions that do not take into account the peculiarities of machine code. Instead, it was decided to take as a basis

the unfinished code of jessie3 project [13] – part of the Why3 project. The Jessie3 code has been modified and extended to take into account the peculiarities of machine code. In particular, the language WhyML has been described the type of processor registers. In addition, the algorithm of generating targets for the proof was changed for the subsequent fusion – pre- and postconditions were separated from the function behavior model.

3.2 Using register type for compatibility with machine code

Processor registers can be represented by a limited integer type with an extended set of operations. Operations include signed and unsigned arithmetic, bitwise operations, and memory read operations at the address specified in the register and by offset. To describe all such operations high-level languages, use a variety of different types, as well as a cast operation. However, using different data types will complicate the proof of correctness problem for SMT-solvers. This is especially noticeable in the case of bitwise operations, which are available only for bitvectors in SMT-LIB. Bitvectors cast operations to an integer type are not supported by the latest SMT-LIB [14] standard, and various SMT-solvers offer their own version of the implementation of this operation.

The BitVec type from SMT-LIB is well-suited for describing the type of registers because it contains all the necessary arithmetic and logical sign and unsigned operations. However, the theory of bitvectors at the why3 level does not support all the necessary operations and is built as an unsigned type. Based on the standard theory of bitvectors, the author developed a theory to support the type of processor registers. The theory supports both signed and unsigned integer types and there is ongoing work to add support for pointer arithmetic and memory dereferencing. The driver for CVC4 SMT-solver [15] was updated for translation of the register type to the type BitVec with corresponding mapping of operations.

3.3 Splitting specification and behavior model

To merge machine code, you must separate the pre - and post-conditions from the behavior of the high-level function, which will then be replaced by the behavior of the machine code. To implement this approach, the author uses abstract logical predicates of pre- and postconditions checking. These predicates take as input the parameters of the verification function, and the predicate of the postcondition is also taking its result. Further, by means of axioms predicates are defined by a logical expression in accordance with ACSL specifications. In fig. 3 you can see the predicates for pre- and postconditions are generated based on the ACSL specifications of absolute value function (fig. 4), where usabs_pre – the predicate of a precondition, and usabs_post is a predicate of the postcondition.

```
predicate usabs_post r32 r32

predicate usabs_pre r32

axiom usabs_post_axiom :
  forall n:r32, result:r32.
   usabs_post n result <->
    sge result (of_int 0) /\ (eq result n \/ eq result (sub (of_int 0) n))

axiom usabs_pre_axiom :
  forall n:r32. usabs_pre n <-> slt (neg (of_int 2147483648)) n
```

Fig. 3. WhyML abs logic specification

```
/*@ requires -2147483648 < n;
    ensures \result == n || \result == 0-n;
    ensures \result >= 0;
 */
int abs(int n)
```

Fig. 4. ACSL abs specification

3.4 Replacing proof goal

To facilitate the subsequent merging, the proof goal is substituted during translation of WhyML to SMT-LIB. A new goal for the proof can be described as follows: If the precondition of a function with its arguments is satisfied then the postcondition with the arguments of the function and its result is not satisfied. The negation is used because the SMT-solvers operation specifics – searching for example variable values that will satisfy all restrictions described in SMT-LIB model.

```
;;function argument 1
(declare-const _arg (_ BitVec 32))
;;assign _arg here

;;function result
(declare-const _func_res (_ BitVec 32))
;;assign _func_res here

(assert (usabs_pre
_arg ))

(assert (not (usabs_post
_arg  _func_res)))
```

Fig. 5. Proof goal template

If such an example could not be found then the assumption is incorrect and the predicate of the postcondition is always executed. Therefore, the Expected verdict of the SMT-solver – unsat. It is important to note here that arguments and the result of the function execution are not associated with machine code at this stage – the

merge module solves the problem of their binding. Fig. 5 shows SMT-LIB code of goal to prove the correctness of the absolute value function.

3.5 Merging high- and low-level specifications

If you perform all the steps described in the previous sections of this paper, namely, creating an NML model of the machine code and an ACSL to the WhyML translation module, you can perform a merge in two different ways. The first method is the merging at the level of WhyML, and the subsequent translation to SMT-LIB by means of Why3. This approach has a number of advantages, mainly related to Why3 capabilities for WhyML code analysis.

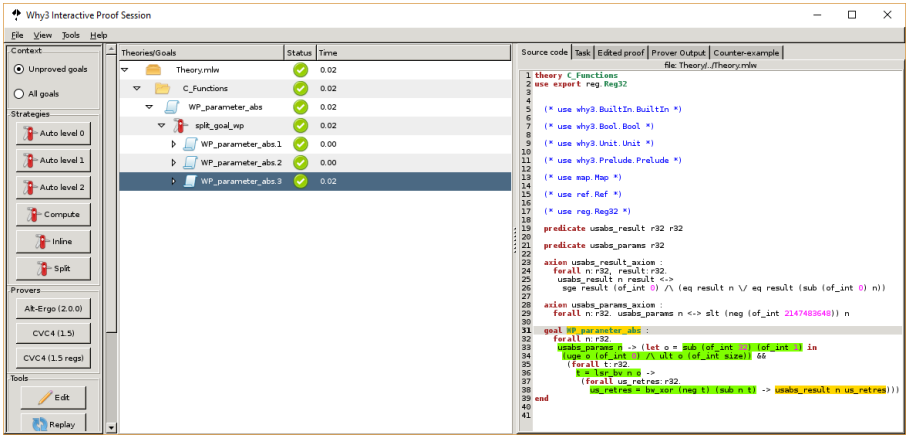

Fig. 6 Why3 IDE

It is worth noting that Why3 IDE (fig. 6), can be used for interactive proof and manual simplifications of verification goals. At the moment the MicroTESK team, with the support of the author, is developing an NML to WhyML translation module. The second approach, as well as the only one implemented at the moment, is merging at the SMT-LIB level. The main advantage of this approach is that the MicroTESK tool has already been implemented NML to SMT-LIB translation module. In addition, the vast majority of operations and data types available in NML have analogs in SMT-LIB.

For example, a set of General-purpose registers is modeled in the NML of the PowerPC processor model as an array of 32-bit registers with a 5-bit index. There is no predefined 5-bit unsigned type in Why3, let alone an array with such an index. However, in SMT-LIB, as in NML, you can manually set the length of BitVec constants. In addition, the translation directly to SMT-LIB allows to avoid unnecessary abstractions that Why3 algorithm for WhyML to SMT-LIB translation can add.

The task of the merge module is to bind together the function arguments and the result of function of high-level language with registers and memory of the model of machine code, and set the environment. Here, the environment refers to machine-

specific things, such as the initial value of the stack register or instruction counter. To do this, it is necessary to take into account the specificity of generation SMT-LIB behaviors of the machine code and the specification for the function and specificity of the ABI of architecture.

Next, in fig. 7 we can see binding of the arguments of instructions with the registers for the PowerPC architecture. Developed by the MicroTESK team, generation SMT-LIB by the NML model produces thousands of lines of code. This code can be divided into two main parts: The declaration of all the logical constants needed to describe the behavior model and the description of the state transformation formulas by means of using one assert per machine code instruction and one for every of machine instruction argument.

```
;;function argument 1
(declare-const _arg_1 (_ BitVec 32))
(assert (= _arg_1 (select GPR!1 (_ bv3 5))))

;;function result
(declare-const _func_res (_ BitVec 32))
(assert (= _func_res (select GPR!47 (_ bv3 5))))
```

Fig. 7 Binding function argument and result

4. Evaluation

The developed approach was successfully used to verify the machine code of the absolute value function on the basis of bitwise operations (“Fig. 8”), for which a verdict was obtained, clearly indicating correctness of the function. Tests were also developed to verify the correctness of the implementation of translation of mathematical and logical operations of the ACSL language. Testing of the NML model was done by means of MicroTESK tool.

```
int abs(int n)
{
    int t = (unsigned int) n >> (32-1);
    return (-t) ^ (n-t);
}
```

Fig. 8 Absolute value function

5. Related works

In the why3-avr [16], [17] project, the deductive verification approach is used to prove the correctness of non-loop programs in the assembly language of the AVR microcontroller. The AVR microcontroller used in this study has a fairly simple instruction set that allows you to manually specify the behavior model for each command in the WhyML language, which does not have special means to describe such structures. Also, the model code is described in such a way that allows the programmer to simply copy the function code in the AVR assembly language and add to it a formal specification to get WhyML code for checking the correctness of the function. This approach is especially useful for direct development in a low-

level language because the Why3 tool has rich capabilities for transformation and analysis of Why3 code. In addition, the use of Why3 allows converting the WhyML code for proving by various SMT solvers.

However, the program in assembly language is different from compiled machine code that in machine code is a sequence of bytes where there is no all information associated with label names and variables, as well as the formal specification. In addition, machine code does not allow you to abstract from your environment as much as assembly language code. For example, in machine code, indicators such as the address of a function in memory and the value of the stack register at the time of entering the function are important. Also, a high-level formal language specification, such as C, uses various abstractions, such as parameter names and variables, that become unavailable after they are translated into assembly language or machine code. The approaches proposed by the author differ from those described in this project in that they allow using the specification of the high-level language function for analyzing machine code, as well as scaling the supported command system with the help of a specialized modeling language hardware NML.

In the Technical report published by the University of Cambridge Computer Laboratory [18], the HOL4 proof assistant [19] is used for Formal verification of machine-code programs. The paper describes a tool able to verify the machine code for subsets of instructions for popular architectures ARMV4, PowerPC, x86. Behavior model for these instructions was developed by independent developers, so models for both ARM and x86 was designed for HOL4 language [20] [21], and the PowerPC model [22] were manually translated from the Coq language [23] to HOL4.

Here it is worth noting the similarity with the project why3-avr because instructions behavior models were specified manually on unspecialized for such a purpose language. The report terminology uses four levels of abstraction to describe the logical implementation and specification of functions. To obtain a low-level function model (level 2) automatic decompiler translates the machine code (level 1) into recursive functions on the HOL4 language, and also generates their specifications. The use of recursion, in this case, avoids the need to define loop invariants. The user can then focus on interactively proving the properties of the generated function using the HOL4 proof assistant.

For verification, the user also needs to describe the high-level model of the function (level 3), as well as the specification of the function for (level 4). Further, by using relations between levels, user proves that the machine code model complies with the functional specification. In contrast to the interactive HOL4 approach, the approach used in the author's study allows the presence of ACSL specifications to carry out all stages in automatic mode. Also in the author's approach to proving the correctness of machine code is not necessary to have a logical model of the behavior of the function in a high-level language. This degree of automation is achieved including the use of automatic SMT-solvers, in contrast to the interactive proof assistant HOL4. Particularly worth noting is the approach to the translation of programs into recursive functions. The use of high-level language loop invariants at

the machine code level is extremely difficult due to the influence of various compiler optimizations. The recursive functions may help to solve these problems.

A number of papers also describe the use of model checking [24] approach for formal verification of machine code. Therefore, in the paper [25] for verification of machine code of the microcontroller Motorola M68hc11 is used Bogor framework [26]. This approach does not imply the presence of function contracts but is based on the use of formally specified behavior models of the system as a whole. As a result, it can be said that the scope of the requirements to be tested varies with the use of deductive verification and model checking.

6. Conclusion

Most of the work that is reviewed specifies the behavior of machine code instructions manually in the logical language. However, in order to simplify and improve the reliability of processor models, the author proposed to describe them in the NML language, designed specifically for such purposes, with the subsequent automatic translation of the model into logical languages. The use of this approach is also facilitated by the presence of a large set of tools in the MicroTESK tool to work with NML, including the NML to SMT-LIB translator. The particularity of ACSL specifications translation to WhyML code, for the case of verification of machine code, such as the need to separate the specification from the behavior model, as well as the importance of the introduction and implementation of the register type.

The observance of such rules and guidelines will allow for automatic merging of function specification and machine code behavior model and thus avoid the need for manual specifying machine code behavior model on the logical language, as required in the project why3-avr. There were proposed two approaches to merge of code specifications and behavior models: at the level of WhyML, and at the level of the SMT-LIB. The first approach allows to use SMT-LIB code generated directly from NML model that help us to avoid extra complexity coming from double translation NML to WhyML and then WhyML to SMT-LIB. The second approach allows to use all the features of the Why3 tool, such as interactive transformations and support of various provers and solvers.

The use of the methods and approaches described in this paper will allow you to fully automate deductive verification of machine code without loops for compliance with the contract specification in ACSL language.

References

[1]. MicroVer – Deductive Verification Tool for Machine Code. Available at: https://forge.ispras.ru/projects/microver, accessed 20.07.2018

[2]. Leroy Xavier. A Formally Verified Compiler Back-end. Journal of Automated Reasoning, vol. 43, issue 4, 2009, pp 363-446

[3]. CompCert – The CompCert C compiler. Available at: compcert.inria.fr, accessed 13-02-2018

[4]. GCC Releases. Available at: http://www.gnu.org/software/gcc/releases.html, accessed 13-02-2018

[5]. Butterfield A., Ngondi G., Kerr A. A Dictionary of Computer Science (ed. 7), Oxford University Press, 2016, 608 p.
[6]. ACSL specification. Available at: http://frama-c.com/acsl.html, accessed 13-02-2018
[7]. Kamkin A., Tatarnikov A. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. In Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012
[8]. C Barrett, R Sebastiani, S Seshia, and C Tinelli. Satisfiability Modulo Theories. In Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, Feb. 2009, pp. 825–885
[9]. EREF: A Programmer's Reference Manual for Freescale Power Architecture Processors, Rev. 1 (EIS 2.1). Available at: http://cache.freescale.com/files/32bit/doc/ref_manual/EREF_RM.pdf, accessed 13-02-2018
[10]. Filliâtre JC., Paskevich A. Why3 – Where Programs Meet Provers. Lecture Notes in Computer Science, vol. 7792, 2013, pp. 125-128
[11]. M. Mandrykin, A. Khoroshilov. A Memory Model for Deductively Verifying Linux Kernel Modules. Lecture Notes in Computer Sciences. vol. 10742, 2018, pp. 256-275
[12]. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowsk. Frama-c: A Software Analysis Perspective. Formal Aspects of Computing, vol. 27, issue 3, 2015, pp 573–609
[13]. Jessie3 at Why3 source repository. Available at: https://gitlab.inria.fr/why3/why3/tree/master/src/jessie, accessed 12.04.2018.
[14]. Barrett C., Fontaine P., Tinelli C. The SMT-LIB Standard Version 2.6. Available at: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf, accessed 12.04.2018
[15]. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds. CVC4. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 171-177
[16]. Schoolderman M. Verifying Branch-Free Assembly Code in Why3. Lecture Notes in Computer Science, vol. 10712, 2017, pp. 66-83
[17]. Why3-avr project repository. Availible at: https://gitlab.science.ru.nl/sovereign/why3-avr, accessed 12.04.2018.
[18]. Myreen M.O.: Formal verification of machine-code programs. Ph.D. thesis, University of Cambridge, 2009
[19]. Konrad Slind and Michael Norrish. A brief overview of HOL4. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 28-32
[20]. Anthony Fox. Formal specification and verification of ARM6. Lecture Notes in Computer Science, vol. 2758, 2003, pp 25-40
[21]. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.
[22]. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 42-54
[23]. Yves Bertot. A short presentation of Coq. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 12-16
[24]. B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen. Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001, 190 p.
[25]. Edelman Joseph R. Machine Code Verification Using the Bogor Framework. Master Thesis, Brigham Young University, 2008
[26]. Bogor framework homepage. Available at: http://bogor.projects.cs.ksu.edu, accessed: 13.02.2018

Совмещение ACSL спецификаций с машинным кодом

П.А. Путро <pavel.putro@ispras.ru>
Национальный исследовательский университет "Высшая школа экономики", 101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. При разработке программ на языках высокого уровня, разработчикам приходится делать предположение о корректности компилятора. Однако это может быть неприемлемо для критически важных систем. Поскольку на данный момент не существует полноценных компиляторов, для которых корректность доказана, автор предлагает решать эту проблему путём доказательства корректности сгенерированного машинного кода методами дедуктивной верификации. Для достижения данной цели необходимо решить ряд задач, одной из которых является слияние модели спецификаций пред- и постусловий с моделью поведения машинного кода. В данной статье представлен подход к проведению слияния спецификаций для случая Си функций без циклов. Суть подхода заключается построении моделей как машинного кода, так и его спецификации на едином логическом языке, и использовании ABI целевого процессора для связывания машинных регистров с параметрами функции высокого уровня. Для успешной реализации такого подхода необходимо предпринять ряд мер по обеспечению совместимости высокоуровневых спецификаций с моделью поведения машинного кода. К таким мерам, в частности, относятся использование типа регистра в высокоуровневых спецификациях, трансляция пред- и постусловий в абстрактные предикаты. Также в статье производится и обосновывается выбор логического языка для построения моделей, выбираются наиболее подходящие инструменты для реализации подхода слияния спецификаций и производится оценка работы системы дедуктивной верификации машинного кода, построенной на основе предложенного подхода, с использованием тестовых примеров полученных путём компиляции Си программ без циклов.

Ключевые слова: дедуктивная верификация; формальные методы; машинный код; ACSL.

DOI: 10.15514/ISPRAS-2018-30(4)-6

Для цитирования: Путро П.А. Совмещение ACSL спецификаций с машинным кодом. Труды ИСП РАН, том 30, вып. 4, 2018 г., стр. 95-106 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(4)-6

Список литературы

[1]. MicroVer – Deductive Verification Tool for Machine Code, Доступно по ссылке: https://forge.ispras.ru/projects/microver, дата обращения 20.07.2018
[2]. Leroy Xavier. A Formally Verified Compiler Back-end. Journal of Automated Reasoning, vol. 43, issue 4, 2009, pp 363-446
[3]. CompCert – The CompCert C compiler. Доступно по ссылке: compcert.inria.fr, дата обращения 13-02-2018
[4]. GCC Releases. Доступно по ссылке: http://www.gnu.org/software/gcc/releases.html, дата обращения 13-02-2018
[5]. Butterfield A., Ngondi G., Kerr A. A Dictionary of Computer Science (ed. 7), Oxford University Press, 2016, 608 p.
[6]. ACSL specification. Доступно по ссылке: http://frama-c.com/acsl.html, дата обращения 13-02-2018

[7]. Kamkin A., Tatarnikov A. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. In Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), 2012
[8]. C Barrett, R Sebastiani, S Seshia, and C Tinelli. Satisfiability Modulo Theories. In Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, Feb. 2009, pp. 825–885
[9]. EREF: A Programmer's Reference Manual for Freescale Power Architecture Processors, Rev. 1 (EIS 2.1). Доступно по ссылке: http://cache.freescale.com/files/32bit/doc/ref_manual/EREF_RM.pdf, дата обращения 13-02-2018
[10]. Filliâtre JC., Paskevich A. Why3 – Where Programs Meet Provers. Lecture Notes in Computer Science, vol. 7792, 2013, pp. 125-128
[11]. M. Mandrykin, A. Khoroshilov. A Memory Model for Deductively Verifying Linux Kernel Modules. Lecture Notes in Computer Sciences. vol. 10742, 2018, pp. 256-275
[12]. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, Boris Yakobowsk. Frama-c: A Software Analysis Perspective. Formal Aspects of Computing, vol. 27, issue 3, 2015, pp 573–609
[13]. Jessie3 at Why3 source repository. Доступно по ссылке: https://gitlab.inria.fr/why3/why3/tree/master/src/jessie, дата обращения 12.04.2018.
[14]. Barrett C., Fontaine P., Tinelli C. The SMT-LIB Standard Version 2.6. Доступно по ссылке: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf, дата обращения 12.04.2018
[15]. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds. CVC4. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 171-177
[16]. Schoolderman M. Verifying Branch-Free Assembly Code in Why3. Lecture Notes in Computer Science, vol. 10712, 2017, pp. 66-83
[17]. Why3-avr project repository. Availible at: https://gitlab.science.ru.nl/sovereign/why3-avr, дата обращения 12.04.2018.
[18]. Myreen M.O.: Formal verification of machine-code programs. Ph.D. thesis, University of Cambridge, 2009
[19]. Konrad Slind and Michael Norrish. A brief overview of HOL4. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 28-32
[20]. Anthony Fox. Formal specification and verification of ARM6. Lecture Notes in Computer Science, vol. 2758, 2003, pp 25-40
[21]. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.
[22]. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 42-54
[23]. Yves Bertot. A short presentation of Coq. Lecture Notes in Computer Science, vol. 5170, 2008, pp. 12-16
[24]. B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen. Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001, 190 p.
[25]. Edelman Joseph R. Machine Code Verification Using the Bogor Framework. Master Thesis, Brigham Young University, 2008
[26]. Bogor framework homepage. Доступно по ссылке: http://bogor.projects.cs.ksu.edu, дата обращения 13.02.2018