Modeling of ALFA Programs Using PVS Theorem Prover

Shimmi Asokan, G. Santhosh Kumar
Department of Computer Science
Cochin University of Science and Technology
Kochi-22, Kerala, India
shimmideepak@gmail.com, san@cusat.ac.in

N. Jaya Lal
Human Spaceflight Programme
Vikram Sarabhai Space Center
Trivandrum, Kerala, India
n_jayalal@vssc.gov.in

Abstract— **In Safety critical software failure can have a high price. Such software should be free of errors before it is put into operation. Application of formal methods in the Software Development Life Cycle helps to ensure that the software for safety critical missions are ultra reliable. PVS theorem prover, a formal method tool, can be used for the formal verification of software in ADA Language for Flight Software Application (ALFA.). This paper describes the modeling of ALFA programs for PVS theorem prover. An ALFA2PVS translator is developed which automatically converts the software in ALFA to PVS specification. By this approach the software can be verified formally with respect to underflow/overflow errors and divide by zero conditions without the actual execution of the code.**

Keywords- ALFA; Formal Modeling; Formal Verification; Prototype Verification System; Type Correctness Conditions

I. INTRODUCTION

Ada Language for Flight Software Application (ALFA) [1] is a safe subset of SPARK Ada designed for the development of mission critical software. Such software belongs to the class of human-rated software. Safety critical software such as the onboard software used in Human Spaceflight Programme (HSP) should be highly reliable. The errors in such software should be uncovered in the early stages of development life cycle itself. Use of formal methods causes more defects to be detected than would otherwise be the case and in some circumstances it also guarantees the absence of certain defects [2]. Prototype Verification System (PVS) [3] is an efficient theorem prover used for this purpose. To conduct formal verification of software developed in ALFA using the PVS theorem prover, it should be modeled and converted to the formal specification supported by PVS. This paper describes the modeling of the onboard software implemented in ALFA for the PVS theorem prover. An ALFA2PVS translator tool is also developed to automatically convert the software in ALFA to PVS specification according to the modeling rules. The PVS specification generated by this tool can be typechecked in the PVS theorem prover for formal verification. Typechecking the specification generates Type Correctness Conditions (TCCs) for all the type inconsistencies and these TCCs have to be proved to ensure that the software is typesafe [5]. Using the modeling described here, overflows/underflows errors and divide by zero errors can be detected as part of proving the correctness of the software.

The rest of this paper is organized as follows. Section II gives the background. Formal modeling of ALFA programs is described in section III. Section IV describes ALFA2PVS translator for converting the software coded in ALFA to PVS specification. Section V describes the process of formal verification of ALFA programs and a brief conclusion is given in section VI.

II. BACKGROUND

Literature provides works on modeling a programming language for a specification language. A work on detecting runtime errors in MISRA C programs using PVS [5] explains the modeling of MISRA C programs in PVS. A C2PVS translator is developed for automatic conversion of MISRA C programs to PVS specification.

An Assertion Checking Environment (ACE) for unit level formal verification of sequential C programs using static assertion checking technique [6] describes the conversion of MISRA C programs to semantically equivalent program in Simple Programming Language (SPL) for the STeP theorem prover [9]. A c2spl translator automates this conversion.

An adatospl translator was developed as part of the work, system for object code validation (OCV). It describes a methodology and a system for the validation of translation of a safe subset of Ada to assembly language programs [7]. STeP is used for performing proof of refinements. The modules coded in a safe subset of SPARK Ada were translated to SPL using the translator. This translator is also used for formal specification and verification of a launch vehicle onboard software component namely Fault Detection and Isolation Logic (FDIL) [8].

III. FORMAL MODELING OF ALFA PROGRAMS

A procedure in ALFA is encoded as a theory in PVS. A theory consists of a list of states. A state is represented as a tuple of n+1components where n is the number of variables in the program. The first component in the tuple is a flag variable. A state s_1 is represented as s_1: state = ($s_1\text{`}1$, $s_1\text{`}2$,……, $s_1\text{`}i$,……., $s_1\text{`}n+1$) where $s_1\text{`}1$ is the component that corresponds to the flag variable and $s_1\text{`}2$,……, $s_1\text{`}i$,……., $s_1\text{`}n+1$ represents the values of the variables in the program. An instruction in ALFA represents a state transition in PVS which modifies the program variables participating in that instruction.

978-0-7695-3845-7/09 $26.00 © 2009 IEEE
DOI 10.1109/ARTCom.2009.134

Conversion of the ALFA code to PVS specification involves modeling the type definitions, the statements and the control constructs of ALFA in PVS.

A. Modeling the Datatypes

In ALFA the range of values for the datatypes, *short_integer, unsigned types and float types* are confined between a maximum and minimum value. But in PVS the values of the variables of type integer and real ranges from $-\infty$ to $+\infty$ and that of natural ranges from 0 to $+\infty$ [3]. The data types in ALFA are defined as subtypes of the *integer, natural* and *real* types in PVS.

The *short_integer* of ALFA is represented as a subtype of the *integer* type in PVS, confined between the maximum and minimum permitted short_integer values, -32768 to +32767 in ALFA.

conf_short_int: TYPE = {a: int| a<=sint_max AND a>=sint_min AND sint_max>=sint_min}

sint_max and sint_min are +32767 and -32768 respectively.

The arithmetic and relational operators on short_integer are represented as functions in PVS which takes arguments of type *conf_short_int* and returns value of PVS *int* type. The division operation on short_integer is represented as

div_conf_short_int(a: conf_short_int, b: conf_nzshort_int): int= round_to_int(a/b)

An operation a^{-b} should be converted in PVS as

div_conf_short_int (1, exp_conf_short_int (a, b))

The unsigned types in ALFA are modeled as subtypes of the *natural* datatype of PVS.

conf_unsigned16: TYPE = {a: nat| a<=usint_max AND a>=usint_min AND usint_max>= usint_min}

The bit_AND operation on unsigned_16 in ALFA is modeled as follows.

bitAND_conf_unsigned16 (a: conf_unsigned16, b: conf_unsigned16): bvec= AND(a,b)

The float datatypes are modeled as subtypes of the *real* type in PVS.

*conf_float32: TYPE = {a: real| ((a <= float32_MAX AND a >= float32_MIN) OR (a <= -1*float32_MIN AND a >= -1*float32_MAX) OR (a = 0)) AND (float32_MAX >= float32_MIN)}*

The datatypes of ALFA and the operations on them are encoded as theories in PVS. Whenever a datatype is referred the corresponding theory is imported to the specification file and the operations on them can be replaced with the corresponding functions.

B. Modeling the Assignment Statements

Assignment statement in ALFA is modeled in PVS as a new state, with the value of the component that corresponds to the variable being replaced with the value of the expression. An assignment statement of the form,

st_1: v_i: = expression1; is modeled as

s_2: state = (s_1`1, $s_1$`2,......, expression1, s_1`i+1,......., $s_1$`n)

C. Modeling the Selection Constructs

A *nested if* construct is modeled in PVS as follows.

s_2: state = IF boolexpn1(s_1) = TRUE THEN s_1 ELSE unreachable ENDIF

....

s_4: state = IF boolexpn2(s_3) = TRUE THEN s_3 ELSE unreachable ENDIF

s_5: state transition for the last statement in the inner if part

s_6: if reachable(s_5) then s_5 else s_3 endif

s_7: state transition for the last statement in outer *if* part

....

s_{13}: state transition for the last statement in *elsif* part

s_{14}: state = if reachable(s_{13}) then s_{13} elsif reachable(s_7) then s_7 else s_1 endif

The state unreachable is defined as a state where the flag variable is false and reachable is defined as a function which checks the flag variable and returns true if the flag variable is true.

reachable: [state -> bool] = LAMBDA (s: state): (s`1 = TRUE)

D. Modeling the Iterative Constructs

Statements inside the iterative constructs in ALFA are encoded as a separate theory in PVS. This theory is typechecked separately to ensure that there are no type inconsistencies in the statements inside the loops. The state after the iterative construct is modeled as a state where the loop condition is not satisfied and the loop invariant conditions are satisfied [5].

IV. ALFA2PVS TRANSLATOR

An *ALFA2PVS* translator tool automates the generation of PVS specification from the ALFA program. Fig 1 shows the steps involved in the development of *ALFA2PVS* translator. The annotated ALFA procedure is the input to the translator. The source program is parsed based on the grammar of ALFA. The comments in the source file are extracted before parsing.

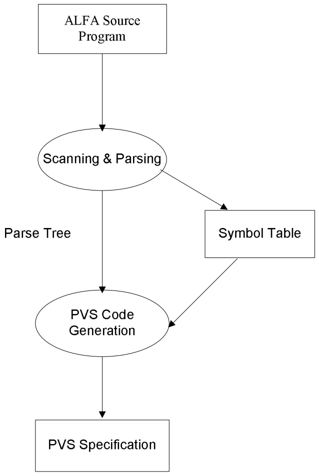

Figure 1. ALFA2PVS Translator

These comments are then redistributed in the parse tree after parsing the file. PVS code generation is done by performing parse tree walking and translating the ALFA statements to their corresponding form in PVS, according to the modeling rules described in the previous section. The annotations in the source file are converted to axioms of the form given below by the translator.

axiom_n: AXIOM s1`3>=1 AND s1`3<=100

For iterative constructs, the translator generates a separate PVS file containing the translations for the statements inside the loop.

V. FORMAL VERIFICATION OF ALFA PROGRAMS

The process involved in the formal verification of ALFA programs using the PVS theorem prover is shown in fig 2. The first step in formal verification is to annotate the ALFA source code. Annotations are derived manually from the data range of the input variables. These annotations are then inserted as comments in the source code by the user. The preconditions should begin with the word "precond", post conditions with the word "postcond" and the loop invariants should begin with the word "loopinvariant".

-- precond DAPCPrev >= -10 AND DAPCPrev <= 10

The PVS theorem prover accepts input which is encoded in its own specification language. So the program in ALFA should be first converted to this form. The ALFA programs are modeled in PVS and *ALFA2PVS* translator tool automates the generation of PVS specification from the ALFA code. The PVS specification which is the output of the translator is loaded to the PVS theorem prover and typechecked. Since PVS specification language is strongly typed, the type inconsistencies in the specification generate TCCs. These TCCs are to be proved to prove that the specification is typesafe. The prover commands [4], the axioms in the specification and the theories in the prelude library can be used to discharge the TCCs. Any unproved TCC can be traced as a possible runtime error in the ALFA program. The subtraction operation on unsigned types, z = x - y, generates subtype TCC in PVS to ensure that the operation does not underflow.

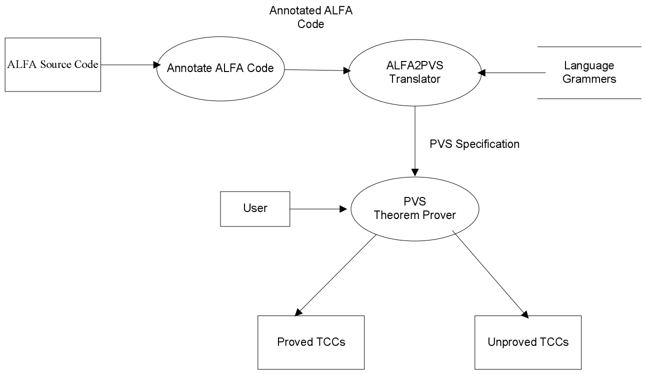

Figure 2. Formal Verification of ALFA Program

All the variables are declared as type conf_unsigned16. The operation sub_conf_unsigned16() returns a result of type nat. Assigning this value to the variable z of type conf_ unsigned16 results in type inconsistency. Typechecking this in PVS generates the following TCC to ensure that the result does not underflow.

sub_conf_unsigned16(s1`2, s1`3) <= usint_max AND sub_conf_unsigned16(s1`2, s1`3) >= usint_min AND usint_max >= usint_min

Also this operation on the unsigned_16 variables x and y, generates the TCC to ensure that x-y >=0.

sub_conf_unsigned16_TCC1: OBLIGATION FORALL (x, y: conf_unsigned16): x - y >= 0;

VI. CONCLUSION

For safety critical systems, verification is the most important phase. For such systems formal methods improves defect detection early in the life cycle and thus guarantee that the software is reliable. This paper details the modeling of ALFA programs in PVS and the development of an *ALFA2PVS* translator that automates the conversion of ALFA procedures to PVS specification according to the modeling rules. Typechecking the PVS specification generated by the translator generates TCCs which are to be discharged using prover commands. Unproved TCCs indicates possible errors in the program. Proving the TCCs is now done interactively. This technique enables formal verification of ALFA programs and helps to detect overflow/underflow errors and divide by zero errors without the actual execution of the code.

REFERENCES

[1] ADA Language for Flight Software Application, Reference Manual, VSSC, May 2006.

[2] NASA: Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I-Planning and Technology Insertion [NASA/TP-98-208193], December 1998.

[3] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, PVS Language Reference, Computer Science Laboratory, SRI International, Version 2.4, November 2001.

[4] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, PVS Prover Guide, Computer Science Laboratory, SRI International, Version 2.4, November 2001.

[5] Ajith K.J., Babita Sharma, A.K. Bhattacharjee, S.D. Dhodapkar, S. Ramesh: Detection of Runtime Errors in MISRA C Programs: A Deductive Approach, LNCS 4680, Springer Berlin / Heidelberg, pages 491-504, September 2007.

[6] Babita Sharma, S.D. Dhodapkar, S. Ramesh: Assertion checking environment (ACE) for formal verification of C programs, Reliability Engineering and System Safety 81 (2003).

[7] A. K. Bhattacharjee, Gopa Sen, S. D. Dhodapkar, K. Karunakar, Basant Rajan and R. K. Shyamsundar: A System for Object Code Validation, LNCS 1926, Springer/Verlag, pages 152–169, 2000.

[8] Krishna Sankara Narayanan, Sampada Sonalkar, Formal Specification and Verification of Fault Detection and Isolation Logic, CFDVS, IIT Bombay, January 2004.

[9] Bjorner N, Browne A, Colon M, Finkbeiner B, Manna Z, Pichora M, Sipma H, Uribe T, The Stanford Temporal Prover user's manual, Stanford University, July 1998.

[10] Paul S. Miner: Defining the IEEE-854 Floating Point Standard in PVS, Technical Report: NASA-95-tm110167, June 1995.