International Journal of u- and e- Service, Science and Technology
Vol.8, No.3 (2015), pp.171-178
http://dx.doi.org/10.14257/ijunesst.2015.8.3.16

Design and Implementation of Abstract Syntax of AADL and Code Generation Based on Spoofax/XL

Weining Su, Fan Zhang, Gang Yang and Tianfang Wang

School of Computer Science and Engineering
Northwestern Polytechnical University
Xi'an China
zhangfan@nwpu.edu.cn

Abstract

With the research in the technology of AADL, we design and realize the parser for AADL and code generation tools which transform AADL model into C source code based on Spoofax/XL. This lays a good foundation for further realizing the AADL unified modeling, verification, implementation and code generation integrated environment.

Keywords: *AADL; Transform; Code Generation*

1. Background

Model-driven architecture approach (MDA) [1] proposed by OMG is a very promising method of software development based on abstract models. Architecture Analysis and Design Language (AADL) [2] is an MDA-compliant architecture modeling language. It models the system by scalable symbols, tools, frameworks and precise semantics and provides the validation and analysis of model at the early development of system.

The paper is organized as follows. Section 2 discusses the related work on the research and tools of AADL. Section 3 presents a method of realizing the abstract syntax tree of AADL based on Spoofax/XL. Section 4 describe the testing of the AADL syntax defined by Spoofax/XL. Section 5 realizes the code generation from AADL to program language C, and Section 6 presents some concluding remarks.

2. Related Work

AADL has features to model the real-time aspects of embedded systems and to represent both the software and hardware architectures of the components making up such systems.

OSATE[4] provides a serial of tools such as error analysis, statistical models, scheduling analysis and so on. The company Fremont Associates has developed Furness [5] tool which is integrated in the OSATE development environment in the form of plug-in and only supports periodic thread for scheduling analysis and simulation. France Brest University has researched Cheddar[6] tool which is an open source AADL scheduling analysis tool. Ocarina [7] is an open source tool which use Ada language to develop and AADL models can be generated to Ada or C code by this tool. However, it requires the user to comply with Ocarina AADL format when writing code. Northwestern Polytechnical University, the Embedded Laboratory has developed Embedded Software Model Evaluation and Analysis Tool (ESME) [8] which provides the ability to use AADL for system models design and high-assurance property analysis.

Currently, the research and application of AADL have grown from a simple modeling design phase to a unified development methods for the integrated system about establishment of a comprehensive requirements analysis, model design, testing, validation and code generation. In order to implement the system with the core of syntax definition

ISSN: 2005-4246 IJUNESST

which can convert and validate the code generation, we use the Spoofax/XL[9] to define AADL syntax. Spoofax/XL uses the SDF (syntax definition formalism)[10]. SDF is declarative and highly modular, which merges lexical definitions and context-free grammar definitions into a single formal framework. And its production rules [11] can define both concrete and abstract syntax.

3. Implementation of AADL Syntax based on K

Figure 1 show the lexical definition which is defined in AADL.

```
Lexical Elements

character ::= graphic_character |format_effEctor |other_control_character
graphic_character ::= identifier_letter | digit | space_character | special_character
identifier ::= identifier_letter {[ underline ] letter_or_digit}*
letter_or_digit ::= identifier_letter | digit
numeric_Literal ::= integer_literal | real_literal
Integer_literal ::= decimal_integer_literal | based_integer_literal
real_Literal ::= decimal_real_literal
decimal_integer_literal ::= numeral [ positive_exponent ]
decimal_real_literal ::= numeral . numeral [ exponent ]
eumeral ::= digit {[ underline ] digit}*
exponent ::= E [ + ] numeral | E numeral
positive_exponent ::= E [ + ] numeral
based_integer_Literall ::= base # based_numeral # [ positive_exponent ]
base ::= digit [ digit ]
Based_numeral ::= extended_digit {[underline] extended_digit}*
extended_digit ::= digit | A | B | C | D | E | F | a | b | c | d | e | f
string_literal ::= "{string_element}*"
string_Element ::= "" | non_quotation_mark_graphic_character
comment ::= -{non_end_of_line_character}*
```

Figure 1. Standard AADL Syntax

Figure 2 shows the AADL package and component definitions. These grammar defines the basic composition structure of package and component in AADL. Figure 2 describes that AADL component is composited by component name, component properties, characteristics, flow and modal structure.

```
Packages
package_spec ::=
  package defining_package_name
  ( public package_declarations
    [ private package_declarations ]
    | private package_declarations )
  [ properties ( {property_association}+ | none_statement ) ]
end definIng_package_name ;

Component Types
component_type ::=
   component_category defining_component_type_identifier
  [ prototypes ( {prototype}+ | none_statement ) ]
  [ features ( {feature}+ | none_statement ) ]
  [ flows ( {flow_spec}+ | none_statement ) ]
  [ modes_subclause | requires_ modes_subclause ]
  [ properties ({component_type_property_association | contained_property_association}*
    | none_statement ) ]
  {annex_subclause}*
end defining_component_type_identifier ;
```

Figure 2. Standard Definition of AADL Package and Component Type

3.1. AADL Syntax Definition

As shown in Figure 3, it is AADL lexical definitions by SDF and can be able to identify digital forms which contain positive and negative integer and floating-point data types.


```
module Numerical

imports Character

exports

  lexical syntax
            Digit+                                -> Numeral
            [eE] "+"? Numeral                     -> Exponent
            [eE] "-" Numeral                      -> Exponent
            [eE] "+" Numeral                      -> PositiveExponent
            Numeral PositiveExponent?             -> DecimalIntegerLiteral
            Numeral "." Numeral Exponent?         -> DecimalRealLiteral

            Digit                                 -> ExtentedDigit
            [a-fA-F]                              -> ExtentedDigit
            Digit Digit?                          -> Base
            {ExtentedDigit "_"}+                  -> BasedNumeral
            Base "#" BasedNumeral "#" PositiveExponent?  -> BasedIntegerLiteral

            DecimalIntegerLiteral                 -> IntegerLiteral
            BasedIntegerLiteral                   -> IntegerLiteral
            DecimalRealLiteral                    -> RealLiteral
            IntegerLiteral                        -> NumericLiteral
            RealLiteral                           -> NumericLiteral
```

Figure 3. Lexical Sefinition based on Spoofax/XL

From Figure 3 we can tell that the module in AADL is composed of module name and some definitions such as specific statements, input statement or action.

3.2. AADL Context-free Grammar Definition

AADL components fall into three categories: software components, composition component, application platform components. Figure 4 shows the SDF syntax specification of Figure 2.

```
module ComponentType
Imports Lexical Constant TypeIdentifier Packages ……
exports
            sorts
                        ComponentCategory ……
 context-free start-symbols
 context-free syntax
            AbstractComponentCategory             -> ComponentCategory
            SoftwareCategory                      -> ComponentCategory
            ExecutionPlatformCategory             -> ComponentCategory
      CompositeCategory                           -> ComponentCategory

      Abstract                                    -> AbstractComponentCategory

      Data                                        -> SoftwareCategory
      ……
      System                                      -> CompositeCategory
      Memory                                      -> ExecutionPlatformCategory
      ……
 "thread" DefiningComponentTypeIdentifier
   ComponentPrototypes?
   ComponentFeatures?
   ComponentFlowSpecs?
   Subclause?
   ComponentPropertyAssociations?
   AnnexSubclause*
   "end" DefiningComponentTypeIdentifier ";"      -> ComponentType{cons("thread")}
……
```

Figure 4. SDF Definition of AADL Component Type

AADL can be extended as appendix according to the syntax specification. Various component types and components in the system implementation can be declared as component appendix, which can supplement the component description. At Spoofax/XL, we only defines the syntax for AADL standard appendix [12].

4. Parser Testing based on Spoofax/XL

In Spoofax/XL, the parser which is based on the specific syntax definition automatically transforms the AADL syntax into abstract syntax trees. Figure 5 is a simple module of AADL, and this module defines a package which includes a data input ports and the security level attribute. The AADL abstract syntax tree generated by the parser as shown in Figure 6.

```
module Packages
package Aircraft::Cockpit
  public
    with Avionics :: DataTypes, Safety_Properties;
    AirData renames data Avionics::DataTypes::AirData;
    system MFD
    features
      Airdata: in data port AirData;
      properties
      Safety_Properties::Safety_Criticality => high;
    end MFD;
end Aircraft::Cockpit;
```

Figure 5. AADL Test Example

```
Module(
  "Packages"
, [ Packagespec(
      PackageNames(["Aircraft", "Cockpit"])
    , PackDeclarations(
        [ ImportDeclaration(
            [PackageNames(["Avionics", "DataTypes"]), PackageNames(["Safety_Properties"])]
          )
        , AliasDeclaration(
            Some("AirData")
          , data()
          , ComponentTypeIdentifier(Some(PackageNames(["Avionics", "DataTypes"])), "AirData")
          )
        ]
      , [ system(
            "MFD"
            ......
              properties(
                [PropertyAssociation(
                   UniquePropertyIdentifier(Some("Safety_Properties"), "Safety_Criticality")
                 , PropertyTerm(None(), "high")
                 , None()
                 )]
              )
            )
          , "MFD"
          )
        ]
      )
    , None()
    , None()
    , PackageNames(["Aircraft", "Cockpit"])
    )
  ]
)
```

Figure 6. Abstract Syntax Tree of Example

5. Code Generation based on Spoofax/XL

Spoofax/XL use Stratego[13] to describe the semantics of programming languages. Stratego provides an integrated approach for code analysis, transformation and code generation. Spoofax/XL defines description of editor and implementation of transformation. This design ensures flexibility of the service implementation and allow to integrate with other languages and frameworks in the future.

5.1. Rewrite Rule

Stratego is based on term rewriting that supports programmable rewriting policies introduced by [14].The based transformation are defined by a conditional term rewrite rules:

$$r : t1 \rightarrow t2 \text{ where } s$$

In this term, r is the name of the rule, and t1 and t2 are first-order terms, and s is a strategy expression. When t1 matches the item and the conditions meets s, one rule is applied to an item and will generate an instance matching t2 on the right. Otherwise, the application fails.

According to the program's abstract description, term rewriting rules can be represented by a simple transformation. Using content assist provides an abstract syntax, Spoofax/XL environment supports preparation of these rules and provides a results view of abstract language grammar and transformation.

The context-sensitive transformation can be describe by dynamic rewrite rules, which is instantiated at run time. Its definition is as follows:

$$r : t1 \rightarrow t2 \text{ where rules}(dr : t3 \rightarrow t4)$$

The dynamic rules dr will be defined when r is applied and t1 matching the entries. All the shared variable in t3 and t4 will be inherited in the instantiation of dr.

5.2. Transformation Rule

For transforming AADL component modules to C, the basic idea is to use header file declares the ports and subcomponents, and source file includes component called subroutines, component implementation. The detailed mapping from AADL models to C as described below:

1. The package in AADL models is defined by the #include macro in C file.
2. The software component is defined by the structures in C and variables in the structures indicates the software component properties. And the implementation of the software component is defined as the initialization function which initializes the structures.
3. The header file of component need to declare the related header files which define the subcomponents.
4. The data components in AADL models are correspond to the relevant variables in C and the type of data components are correspond to relevant data types in C.
5. The subprogram components declaration in AADL can be transformed to function call in C, and the implementation of function is consistent with the subprogram components.
6. The process component in AADL models maps to the management structure and modules of processes in C. AADL thread component can be called during initialization of the relevant module.

5.3. Code Generation

Spoofax/XL has realized method of code generation based on the target grammar. Therefore, designers can directly write the familiar language syntax without the need to

master the code format at the time of transformation. Spoofax/XL parser will automatically transform these specific syntax code to generate abstract syntax tree which is significant at the stage of code generation.

Figure 7 shows the control rules in the transformation rules of AADL model into C. The control rules, debug-generate-aterm, define abstract syntax tree, while control rules, generate-c, transform the selected code into C code.

In Figure 8, we design and implement a simple AADL models which contains three data members, four program components and thread components. Figure 9 shows the transformation of the C code for AADL models, which bases on the transformation rules in Section 6.2. In the transformation, data elements are transformed to corresponding structures, and subprogram calls are transformed to a relevant function.

```
rules // Debugging

  // Prints the abstract syntax ATerm of a selection.
  debug-generate-aterm:
    (selected, position, ast, path, project-path) -> (filename, result)
    with
      filename := <guarantee-extension(|"aterm")> path;
      result   := selected

  // Prints the C code of a selection.
  generate-c:
            (selected, position, ast, path, project-path) -> (filename, result)
    with
      filename := <guarantee-extension(|"c")> path;
      result := <aadl-to-c> selected
```

Figure 7. Generation Rules

```
Module DataTest

data matrix
end matrix;

data weather_forecast
end weather_forecast;

data date
end date;

subprogram Matrix_delta
features
  A: in parameter matrix;
  B: in parameter matrix;
  result: out parameter matrix;
end Matrix_delta;

subprogram Interpret_result
features
  A: in parameter matrix;
  result: out parameter weather_forecast;
end Interpret_result;

subprogram getCurrent
features
  result: out parameter Matrix;
end getCurrent;

subprogram getFuture
features
  date: in parameter date;
  result: out parameter Matrix;
end getFuture;

thread Predict_Weather
end Predict_Weather;

thread implementation Predict_Weather.others
subcomponents
contral_input: thread control_in.speed;
contral_output: thread control_out;
contral_threads: thread group control_all;
point_data_set: data setpoint_data;
calls main : {
  current: subprogram weather_DB.getCurrent;
  future: subprogram weather_DB.getFuture;
  diff: subprogram Matrix_delta;
  interpret: subprogram Interpret_result;
};
properties
Period => 12 ms ;
Required_Connection => false;
Compute_Execution_Time => 2.6 ms;
Connection => "reallink" ;
 HiRate => timing :: HiRate;
end Predict_Weather.others;
```

Figure 8. AADL Test Example

```
 #include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Data_matrix{
}Data_matrix;
struct Data_weather_forecast{
}Data_weather_forecast;
struct Data_date{
}Data_date;
void Subprogram_Matrix_delta() {
    A = matrix;
    B = matrix;
    matrix = result;
}
void Subprogram_Interpret_result() {
    A = matrix;
    weather_forecast = result;
}
void Subprogram_getCurrent() {
    Matrix = result;
}
void Subprogram_getFuture() {
    date = date;
    Matrix = result;
}
```

```
struct Thread_Predict_Weather {
 }Thread_Predict_Weather;

 void Thread_Predict_Weather_others(){
     int Period ;
     Period = 12 ;
     bool Required_Connection ;
     Required_Connection = false ;
     float Compute_Execution_Time ;
     Compute_Execution_Time = 2.6 ;
     string Connection ;
     Connection = reallink ;
     string Period ;

     Period = timing.HiRate ;
     void Thread_control_in.speed();
     void Thread_control_out();
     void ThreadGroup_control_all();
     void Data_setpoint_data();

     weather_DB.Subprogram_getCurrent = Subprogram_getCurrent ();
     weather_DB.Subprogram_getFuture = Subprogram_getFuture ();
     Subprogram_Matrix_delta = ();
     Subprogram_Interpret_result =();
 }
```

Figure 10. Generation C Code of Example

6. Conclusion and Future Work

In this paper, we introduces how to use the Spoofax/XL to describe AADL syntax and propose the method for transforming the AADL model into C code and give the relevant implementation rules. At last, we use an example to describe the transformation result of AADL model to C code.

In the next, we will intergrate the model checking and validation tools for optimizing and improving code generation quality, and try to detect and ensure the correctness of code generation.

Acknowledgement

This work is supported by the National Science Foundation of China (No. 61472627). We gratefully acknowledge the financial and technical support from the committee of NSFC. We also wish to thank the anonymous reviewers for their constructive comments.

References

[1] R, J. Soley, “Model driven architecture”, OMG white paper, vol. 308, **(2000)**.
[2] P. H. Feiler, D. P. Gluch and J. J. Hudak, “The architecture analysis & design language (AADL): An introduction”, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, **(2006)**.
[3] “SAE Architecture Analysis and Design Language (AADL)”, AS-5506, **(2004)**.
[4] M. Kerboeuf, A. Plantec, F. Singhoff, A. Schach and P. Dissaux, “Comparison of six ways to extend the scope of Cheddar to AADL v2 with Osate”, Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on. IEEE, **(2010)**.
[5] B. Sun, Y. Dong and X H. Dong, “Enhancing Adaptive Random Testing for AADL Model”, Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), **(2012)**; Fukuoka, Japan.
[6] F. Singhoff, J. Legrand, L. Nana and L. Marcé, “Cheddar: a flexible real time scheduling framework”, ACM SIGAda Ada Letters, vol. 24, **(2004)**, pp. 1-8.
[7] J. Hugues, B. Zalila, L. Pautet and J. Kordon, “From the prototype to the final embedded system using the Ocarina AADL tool suite”, ACM Transactions on Embedded Computing Systems (TECS), vol. 7, **(2008),** pp. 42-65
[8] Y. Dong, Y. Cheng, T. Wu and H. Ye, “Schedulability Analysis for Embedded Systems with AADL Model”, Quality Software (QSIC), **(2013)**; Nanjing, China.
[9] L. C. Kats and E. Visser, “The spoofax language workbench: rules for declarative specification of languages and IDEs”, ACM Sigplan Notices, vol. 45, **(2010)**, pp. 444-463.
[10] J. Heering, P. R. H. Hendriks, P. Klint and J. Rekers, “The syntax definition formalism SDF: Reference

manual. SIGPLAN Not, vol. 24, **(1989)**, pp. 43-75.
[11] M. De Jonge, E. Nilsson-Nyman, L. C. Kats and E. Visser, **"**Natural and flexible error recovery for generated parsers", Software Language Engineering, **(2010)**.
[12] J. Bodeveix, & M. Fllali, "The AADL behavior annex-experiments and roadmap", Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, **(2007)**; Auckland, New Zealand.
[13] M. Bravenboer, K. T. Kalleberg, R. Vermaas and E. Visser, "Stratego/XT 0.17. A language and toolset for program transformation", Science of Computer Programming, **(2008)**, vol. 72, pp. 52-70.
[14] E. Visser,, Benaissa, Z. E. A., & Tolmach, A. Building program optimizers with rewriting strategies. In ACM Sigplan Notice, vol. 34, **(1998),** pp. 13-26

Authors

Weining Su was born in 1989. He is a master in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.

Fan Zhang was born in 1979. He is a associate professor service in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.

Gang Yang was born in 1974. He is a associate professor service in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include distributed embedded computing system modeling technology and the CPS system collaborative design technology.

Tianfang Wang was born in 1992. He is a master in School of Computer Science and Engineering at Northwestern Polytechnical University, His main research interests include modeling, verification and analysis for Large-Scale Complex Embedded system.

