Studying on AADL-based Architecture Abstraction of Embedded Software

Geng WANG, Xing-she ZHOU, Yun-wei DONG
School of Computer Science,
Northwestern Polytechnical University
Xi'an, Shaanxi, 710072, China
E-mail: gwang_03@mail.nwpu.edu.cn,
{xszhou, yunweidong}@nwpu.edu.cn

Hong-bing ZHAO
Information Center,
Xi'an Power Supply Bureau
Xi'an,710037,China
Doctorf@fmmu.edu.cn

***Abstract*—since embedded software reaches a large scale and complexity, MDA and component-based development have become popular in embedded system development. Architecture-based software evaluation, maintenance and evolution can reduce cost of large-scale software, and raise the efficiency of maintenance and evolution. Especially, for legacy system, source code is regarded as the only dependable module to be reused in new systems. Thereby architecture abstraction from source code can describe the software at the level of architecture to support evaluation, maintenance and evolution. This paper studies on abstracting AADL model from C source code and introduces a set of mapping rules between the two languages. The mapping rules focus on identifying components and their relationships defined in AADL specification. An example is given at the end of the paper to demonstrate the algorithm.**

Keywords: software architecture;component;reengineering; AADL ;VxWorks;

I. INTRODUCTION

Due to the large scale and complex structure of embedded software, development schedule and cost have become decision factors to evaluate a software project development. MDA (Model Driven Architecture [1]) and component-based method can facilitate developing large-scale embedded software. Software architecture [2] is a collection of components with certain forms, which contains data components, executive components and connector components; and it is at a higher level of abstraction than source code. For large-scale software, architecture is useful in software reuse, maintenance and evolution. However, for the majority of legacy systems, source code is the only dependable object, and it is a difficult job to evaluate software quality property for software reuse, maintenance and evolution because of the large scale. In this situation, architecture abstraction method is applied to get software architecture from source code.

AADL (Architecture Analysis and Design Language[3]) is produced by SAE, which can describe component model and analyze interactive behaviors among components and some non-functional properties of software architecture. AADL is a typical architecture model of embedded system which has been applied to design and develop high reliable and real time avionic software. In this paper, we focus on the embedded software based on VxWorks, abstract components and their interactions from C program, and set up the software architecture described by AADL. This paper is organized as follows: Section 2 mentions related work on architecture abstraction. Section 3 provides a framework of architecture abstraction algorithm based on AADL. In section 4, we present a set of detailed mapping rules from C source code to AADL. A case study is given in section 5 to demonstrate the algorithm and section 6 presents our conclusions and future work.

II. RELATED WORK

Nowadays, there are various software architecture description and analysis models. A graphical formal software architecture model called Software Architecture Model (SAM)[4] is based on two complementary formalisms—Petri nets and temporal logic. Petri nets are used to visualize the structure and model the behaviors of software architectures and temporal logic is used to specify the required properties of software architecture. UML is another widely accepted object-oriented system modeling and design language, which has been accepted for software architecture descriptions in recent years. However, they cannot describe non-functional attributes, such as reliability, safety, etc [5].

In the filed of avionics software, AADL has attracted a widely attention. Many organizations such as SEI, Airbus, Honeywell, Rockwell-Collins, and the European Space Agency are all actively involved in related work on AADL. AADL can describe component model and analyze interactive behaviors among components and some non-functional properties of software architecture. AADL can be used for the design and development of high-reliable avionics software. Since architecture-based method has been applied to develop large-scale and complex avionics software, architecture abstraction based on AADL can provide support for the architecture-based reliability evaluation, maintenance and evolution. Therefore research on architecture abstraction of avionics software is of great importance.

Software re-engineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [6]. A re-engineering activity usually consists of two phases: The first is a reverse engineering phase in which the system is analyzed to identify its components and their relations and to represent the system at a higher level of abstraction. The

978-0-7695-3825-9/09 $26.00 © 2009 IEEE
DOI 10.1109/EmbeddedCom-ScalCom.2009.13

second is a forward engineering phase in which the system is reconstructed in a new form. An integrated tool set is provided in [7], which is used to re-engineer legacy software. This tool integrates three independent modules: the Software Reengineering Environment (SRE), the Software Specification Environment (SSE), and the Boeing ARPA/STARS Software Engineering Environment (SEE). Gokul V.Subramaniam and Eric J.Byrne[8] present a nine-step process for deriving an object model from existing unstructured FORTRAN source code. Top-down and bottom-up approaches are used to derive objects, classes, class attributes and methods, and relationships among classes.

At the first phase of re-engineering, architecture abstraction should be carried out. Architecture abstraction means abstracting a higher level of abstraction such as architecture or structure chart from source code. For example, UML model can be recovered from an object-oriented language C++ source code. Architecture abstraction aims to identify the components and their relations which consist of the software architecture. Koschke[9] pointed out that architecture abstraction can improve the understandability and reusability of software and facilitate software evolution. Metrics-based method [10] and formal concept analysis (FCA)[11] are general methods on identifying components.

Architecture abstraction based on AADL has not been widely presented. It is necessary to study architecture abstraction technique based on AADL.

III. THE FRAMEWORK OF ARCHITECTURE ABSTRACTION

AADL can be used to describe the software at the level of architecture which is higher than C source code. Because of the different levels of abstraction, C-language and AADL have the semantic and syntactic gaps. The architecture abstraction must be based on the semantic and syntactic analysis of the two languages, and the mapping relations between C codes and AADL components. The method we present in this paper is shown as figure 1.

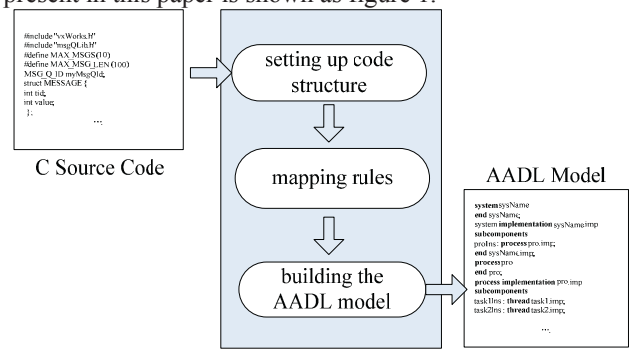

Figure 1. The framework of architecture abstraction.

Our abstraction process has three steps. Firstly, a structure of source code is supposed to be built, because it is difficult to identify components according to C-codes for C-language is an unstructured language. A two layers structure of C source code is built based on the feature analysis of C-program. The top layer is task layer (including tasks and their communications) and the bottom is function layer (including functions and static or global variables called by the tasks), as shown in figure 2:

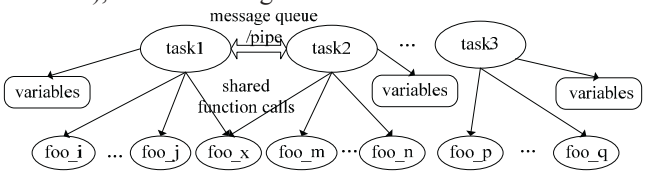

Figure 2. Code structure.

Secondly, a set of mapping rules from C-code to AADL model should be defined in order to bridge the gaps between the two languages. We map tasks to the thread components defined in AADL, variables to the data components and communications among tasks to the connections among thread components. A set of detailed mapping rules will be shown in section 4.

At last, according to the mapping rules, modeling method provided by OSATE is used to set up AADL model. OSATE modeling method is a tool set provided by OSATE [12] modeling environment.

IV. THE MAPPING RULES FROM C TO AADL

In this section, in an effort to bridge the semantic and syntactic gaps between C and AADL, we define a set of mapping rules to support architecture abstraction. The mapping rules defined here are heuristics based on syntactic and semantic features of the two languages. The outline of mappings is shown as figure 3:

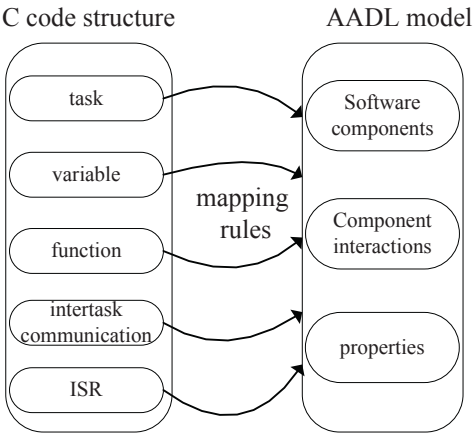

Figure 3. The outline of mapping rules.

A. Identifying Data Components

Data types in the application system can be modeled by data type, data implementation and data instance declarations in AADL. A data type can be used to define the data associated with ports, data implementation to define the internal structure of data type by subcomponents, and data instance to define the reference of other components to data components.

Data types in C language are defined as basic data types (char, int, float, etc.), enumeration, struct, union and pointer, etc. Besides these data types, some identifiers are defined to

declare data types such as static, auto, internal, extern, local and global, etc. A variable in C program consists of type and value.

Since data components in AADL and static variables in C language have the similar descriptions about data type, we map static variables to data components. In order to make the structure of AADL clear, we encapsulate the data components into two types of packages. One package represents the basic data type named basic_data_type and another shows the extended data type named user_data_type. The type of a variable in source code is mapped to a data type in AADL. Generally, for basic data type, it is sufficient to model an application data with a data type, and it is not necessary to declare a data implementation. For extended data type which has the internal implementation, the elements of the extended data are mapped to subcomponents declared in data implementation. The detailed mappings are shown as table1:

TABLE I. THE MAPPING RULES OF DATA

data in C	AADL data components
int a; int *a;	basic_data_type:: int;
char a; char *a;	basic_data_type:: char;
float a; float *a;	basic_data_type::float;
bool a; bool *a;	basic_data_type::boolean;
long int a; long int *a;	basic_data_type::long_int;
double a; double *a;	basic_data_type::double;
typedef struct { int a; }mydata;	user_data_type::mydata; **data implementation** mydata.imp **subcomponents** a: **data** basic_data_type::int **end** mydata.imp;

Because of the limited space of this paper, only a part of data types are listed in table 1. For instance, it doesn't include data types declared by unsigned and signed. The mapping rules should consider all the data types defined both in standard C and VxWorks. As listed in table 1, for basic data types which don't have internal elements, the corresponding data components only have data type; for extended data types which have internal elements, the corresponding data components have data type as well as data implementation declaration. C language defines a special type named "pointer". For pointer data type as shown in table 1, we consider the data type which pointer points to as the data component type. Since data component describes the static data, we only consider the global variables and static variables in C source code (global variable is not a static variable, but it is statically stored).

B. Identifying Thread and Subprogram Components

A thread component in AADL is a concurrent schedulable unit of sequential execution. Multiple threads represent concurrent paths of execution. A variety of execution properties can be assigned to threads, including timing (e.g., worst case execution times), dispatch protocols (e.g., periodic, aperiodic, etc.), memory size, and processor binding. Like a data component, a thread component can be modeled by thread type, thread implementation, thread instance declaration. In VxWorks, multiple tasks represent concurrent paths of execution [13][14]. Thereby, we map tasks to the thread components.

Besides tasks, general functions are defined in VxWorks. Functions represent a sequential execution. Functions can be called by multiple tasks, and they should be reentrant. Subprogram component in AADL represents sequentially executable source text – a callable component with or without parameters that operates on data or provides server functions to components that call it. Based on the similar semantics between function and subprogram component, we map general functions to the subprogram components.

In addition, VxWorks provides two special handler methods— interrupt service routine (ISR) and signal handler routine (SHR). ISR runs on the outside of any task's context and has independent stack. SSR runs in the task's context and uses the task's stack. So, we map the ISRs to thread components which have a high priority and map SHRs to subprogram components. This subprogram is called by the thread component whose corresponding task receives the signal. The mappings from C program to AADL executive components are shown as table 2.

TABLE II. THE MAPPING RULES OF EXECUTIVE COMPONENTS

elements in C	AADL components
task	thread component
ISR	thread component
function	subprogram component
SHR	subprogram component

C. Identifying Connections

VxWorks provides various intertask communications such as shared data structures, semaphores, message queues, pipes and signals. And AADL provides different component interactions, such as port connections, data access connections, subprogram calls. It is necessary to build the mapping rules from intertask communications to component interactions.

1) Port Connection

A port represents a communication interface for the directional exchange of data, events, or both (event data) between components. Ports are classified as data port, event port and event data port. Data port interfaces for typed state data transmission among components without queuing, and event port interfaces for the communication of events and event data port interfaces for message transmission with queuing.

VxWorks provides an intertask communication called message queue. With a message queue, the message sender can send messages to the queue and the message receiver can get messages from the queue. Message queue is mapped to *data port connection* of AADL. The message sender has an *out data port* feature and the receiver has an *in data port* feature. As pipe is similar with the message queue, the same mapping rules are applied to pipe.

Semaphore is another intertask communication mechanism in VxWorks. Binary semaphores can be used for task synchronization. A semaphore can represent a condition or event that a task is waiting for. A task or ISR signals the occurrence of the event by giving the semaphore while

another task waits for the semaphore. This synchronization mechanism is mapped to *event port connection* between thread components in AADL. The giving semaphore's task has an *out event port* feature and the taking semaphore's task has an *in event port* feature.

VxWorks supports a software signal method. Any task or ISR can raise a signal for a particular task. The task being signaled immediately suspends its current execution and executes the task-specified signal handler routine (SHR). It has been introduced in section 4.2 that a task in VxWorks is mapped to a thread component and the SHR is supposed to be a subprogram component. Thereby, the corresponding thread component of raising signal's task has an *out event port* feature, and the receiving signal's task has an *in event port* feature. The event (the signal in C) causes mode transition of receiving event's thread component. Detailed introduction on mode will be shown in section 4.4.

VxWorks allows C functions to be connected to any interrupt. The system suspends its current execution when an interrupt comes and executes the ISR. When completing the ISR, system executes the task being suspended. In section 4.2, we have presented that the ISR is mapped to a thread component. AADL specification defines that thread component must be mapped onto processor component through binding relationship. This ISR thread is bound to a processor component. A processor is an abstraction of hardware and associated software that is responsible for scheduled and executed threads. The interrupt is generated by a device, which is mapped to an AADL execution platform component— device component. The device component that generates interrupt has an *out event port* feature which sends interrupt signal to processor component and processor component receives the interrupt signal from *in event port* feature and then takes the charge of interrupt handling. Since this paper focuses on the software components, we don't have further discussion on interrupt abstraction.

The mapping rules on port connection are shown as table 3.

TABLE III. THE MAPPING RULES OF PORT CONNECTION

elements in C	**AADL port connections**
messageQueue/pipe	data port connection
(messageQueue)msgQsend/ (pipe)write	(thread)out data port
(messageQueue)msgQreceive/ (pipe)read	(thread)in data port
binary semaphore(synchronal task)	event port connection
(task)semtake	(thread)in event port
(task)semgive	(thread)out event port
signal handler	event port connection
generating signal's task	(thread)out event port
receiving signal's task	(thread)in event port
interrupt service	event port connection
device generating interrupt	(device)out event port
OS	(processor)in event data port

2) Data Access Connection

Besides port connection, AADL provides data access connection to describe the connections between data components and other components requiring data. In C source code, data access is defined as an operation to a variable. The operation includes assignment, reference and copy, etc. The operation to a variable in C source code is mapped to a data access connection in AADL where the thread or subprogram component has a *requires data access* feature and the data component provides the data. As presented in section 4.1, only global variables and static variables are considered. The mappings are shown as table 4.

TABLE IV. THE MAPPING RULES OF DATA ACCESS CONNECTION

operation of variable in C	**AADL data access connection**
operation of variable in task	(thread)requires data access
operation of variable in ISR	(thread)requires data access
operation of variable in functions	(subprogram) requires data access
operation of variable in SHR	(subprogram) requires data access

3) Subprogram Calls

Within AADL specification, subprogram calls are declarations within a thread or subprogram implementation demonstrating the call by a thread or subprogram. And the subprogram is shared in process space. In VxWorks, functions are called by tasks, and reentrant functions are shared by multiple tasks. As presented in section 4.2, tasks and ISRs are mapped to the thread components, and functions and SHRs are mapped to the subprogram components. So we map the call relationships between tasks and functions to the subprogram calls from thread to subprogram. The mappings are shown in table 5.

TABLE V. THE MAPPING RULES OF SUBPROGRAM CALLS

function calls in C	**AADL subprogram calls**
function/SHR calls another function	**subprogram implementation** subprogramName **calls**{defining_call_identifier: **subprogram** called_subprogram}; **end** subprogramName;
task/ISR calls a subprogram	**thread implementation** threadName **calls**{defining_call_identifier: **subprogram** called_subprogram}; **end** threadName;

D. Identifying Mode

Modes are represented as states within a state machine abstraction. Modes can establish alternative configurations of active components and connections as well as the transitions among these configurations; they also can establish variable call sequences within a thread. In section 4.2, ISR is mapped to a thread of AADL model and SHR is mapped to a subprogram. Since ISR has a higher priority and is in the different stack with tasks, the corresponding thread components of task and ISR are in the different process modes. For signal handler routine, the corresponding subprogram components of SHR and functions are in the different thread modes. Interrupt causes the mode transition in the process component and the signal causes mode transition in the thread component. The mappings are shown as table 6.

TABLE VI. THE MAPPING RULES OF MODE

elements in C	**AADL mode**

tasks & ISRs	**process implementation** processName **subcomponents** task : **thread** task **in modes**(normal); ISR : **thread** ISR **in modes**(high);
Function & SHR	**thread implementation** threadName **calls** func: **subprogram** func **in modes** (normal); SHR : **subprogram** SHR **in modes**(high);

E. Constructing System

All the mapping rules presented above describe an entire mapping relationship between C source code and AADL model. However, the corresponding AADL model is not a complete system, because within the AADL specification thread components must be a subcomponent of a process component. The process represents a protected address space and an actively executing component, and a process must contain a thread. In VxWorks, all the tasks share a linear address space. So we map this address space to the process component. Both the data and thread or subprogram components are the subcomponents of this process component. In addition, within the AADL specification the process component should be a subcomponent of a system. The system represents a composite of software, execution platform or system components. So we construct a system component which contains the process. By now, a systematic hierarchical AADL components model has been abstracted from C source code over VxWorks.

V. CASE STUDY: FLIGHT CONTROL SYSTEM

In this section, a flight control system of avionic software is given to illustrate the mapping rules. When the aircraft is in normal flight, six control commands support the normal flight. The flight control commands include horizontal flight, straight flight, climbing, coordinated turn, circling and trajectory track. To implement these commands, the aircraft should be controlled vertically and laterally. The vertical controls contain three commands: height control, pitch angle control and coordinated turn control while the lateral control contains four commands: roll angle holding zero, course control, roll angle control and lateral deviation control.

The flight control program is developed based on VxWorks environment. Data structure Sensor_Data represents the data parameters related to various aircraft's sensors such as pitch angle rate, speed and height, etc. The six flight control commands are implemented by six tasks and the seven vertical and lateral controls are implemented by seven functions. Each task has to use the data provided by the data structure Sensor_Data, and calls the different functions to implement flight control. Due to the limited space of this paper, a code structure is shown here instead of the complete source code, as shown in figure 4.

Figure 4 demonstrates a partial code structure of the whole flight control system which is developed based on VxWorks environment. The six flight control tasks are shown on the left and the seven vertical and lateral functions are shown on the right. The calls relationship between tasks and functions are expressed by the directed arrows. Based on the mapping rules presented in section 4, the tasks are mapped to the thread components and the functions are mapped to the subprogram components. And the subprograms are called by the corresponding threads. Figure 5 shows the corresponding AADL model which is mapped to the C source code. Due to the limited space, a graphic representation is given instead of textual representation.

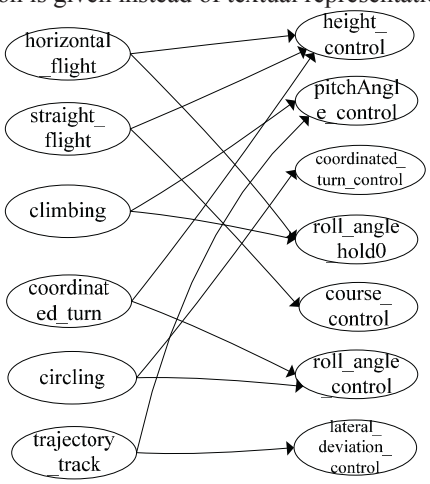

Figure 4. Code structure

This AADL model includes a system component, and a process component which has six thread components and a data component as its subcomponents. The data is user_data_type::Sensor_Data. Each thread maps a flight control command and calls the subprograms of the vertical and lateral control commands. The data component Sensor_Data provides flight parameters to the threads and the threads have "requires data access" feature. The connections between threads and data are "data access connections". Figure 5 is a partial abstraction of the complete flight control system, which only has a part of software abstraction.

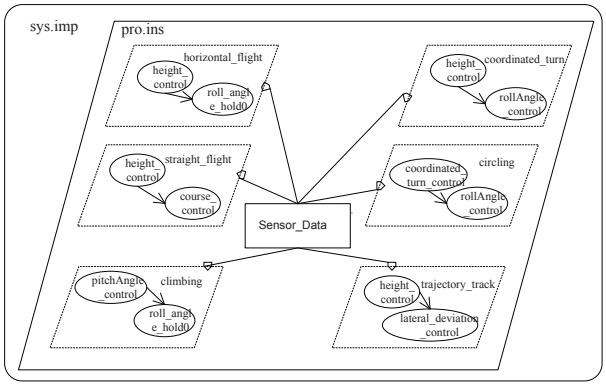

Figure 5. The corresponding AADL model

VI. CONCLUSION AND FUTURE WORK

In this paper, we have introduced architecture abstraction and software re-engineering in software development activity. And then a framework of architecture abstraction from C-code based on VxWorks to AADL model is given. In an effort to bridge the semantic and syntactic gaps between

the two languages, we have defined a set of mapping rules from C to AADL model.

Currently, researches on model abstraction from code are hot issues. Andrew Sutton presents a set of mappings from C++ to UML class models [15][16]. However, because of the character of embedded software (e.g., customization and diversification), the architecture abstraction work for embedded software is more difficult. Since the research on embedded software architecture abstraction has still been in a development stage, some techniques are not very adequate. This paper combines VxWorks operation system which has been widely applied to avionic embedded system with AADL which can describe both functional and nonfunctional attributes of embedded software, and defines a set of mapping rules between C-code and AADL. Using this algorithm, we can carry out the architecture abstraction of embedded software running on VxWorks environment. Since nonfunctional attributes are important character to describe avionic embedded software, we plan to carry on researches on abstraction of nonfunctional attributes. Moreover, in this paper, we focus on the abstraction from static source code, and we plan to add the dynamic analysis of the program execution to provide a more complete and accurate embedded software architecture abstraction algorithm.

REFERENCES

[1] OMG,Model Driven Architecture(MDA).http:// www.omg.org /mda/,2002

[2] Perry,D.E.Software engineering and software architecture .In: Feng, Yu-lin, ed. Proceedings of the International Conference on Software: Theory and Practice. Beijing: Electronic Industry Press,2000.1-4.

[3] Peter H.Feiler,David P.Gluch,John J.Hudak.The Architecture Analysis & Design Language(AADL): An Introduction . Carnegie Mellon University,2006.

[4] Xudong He and Huiqun Yu, Formally analyzing software architectural specifications using SAM, Journal of Systems and Software,2004.

[5] Yujian Fu, Zhijiang Dong, Xudong He, Formalizing and Validating UML Architecture Description of Web Systems, ICWE'06,2006.

[6] E.Chikofsky and J.Cross, Reverse Engineering and Design Recovery – a Taxonomy. IEEE Software, Jan.1990.

[7] Franklin J.Zigman, Integrating Reengineering, Reuse and Specification Tool Environments to Enable Reverse Engineering,1995 IEEE.

[8] Gokul V.Subramaniam and Eric J.Byrne, Deriving an Object Model from Legacy Fortran Code,1063-6773/96,1996 IEEE

[9] R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. PhD thesis, Univ. of Stuttgart, Germany, 2000.

[10] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch Tool. IEEE Trans. Software Eng., 32(3):193–208, 2006.

[11] M. Siff and T. W. Reps. Identifying modules via concept analysis. IEEE Trans. Software Eng., 25(6):749–768,1999

[12] the Software Engineering Institute ,OSATE. http://la.sei.cmu.edu/aadlinfosite/OpenSourceAADLToolEnvironment.html #Topic5

[13] Wind River, VxWorks Programmer's Guide, 5.4

[14] Wind River, VxWorks Reference Mannual, 5.4

[15] Andrew Sutton, Jonathan I. Maletic, Mapping for accurately reverse engineering UML class models from C++, Proceedings of the 12th Working Conference on Reverse Engineering(WCRE'05),2005

[16] Andrew Sutton, Jonathan I. Maletic, Recovering UML class models from C++: a detailed explanation, Information and Software Technology 49(2007) 212-229