Extracting architectural information from source code of ARINC 653-compatible application software using CEGAR-based approach

S.L. Lesovoy <lesovoy@ispras.ru>
Ivannikov Institute for System Programming of RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. It may be useful to analyze and reuse some components of legacy systems during development of new systems. By using a model-based approach it is possible to build an architecture model from the existing source code of the legacy system. The purpose of using architecture models is to analyze the system's static and dynamic features during the development process. These features may include real-time performance, resources consumption, reliability etc. The architecture models can be used as for system analysis as well as for reusing some components of the legacy system in the new design. In many cases it will allow to avoid creation of a new system from scratch. For creation of the architectural models various modeling languages can be used. In the present work Architecture Analysis & Design Language (AADL) is used. The paper describes an algorithm of extracting architectural information from source code of ARINC 653-compatible application software. ARINC 653 specification defines the requirements for software components of Integrated Modular Avionics (IMA) systems. To access the various services of ARINC 653 based OS an application software uses function calls defined in the APplication/Executive (APEX) interface. Architectural information in source code of application software compliant with ARINC 653 specification includes different objects and their attributes such as processes in each partition, objects for interpartition and intrapartition communications, as well as global variables. To collect the architectural information, it is necessary to extract all APEX calls from source code of application software. The extracted architectural information can be further used for creation the architecture models of the system. For source code analysis an approach based on Counterexample-guided abstraction refinement (CEGAR) algorithm is used. CEGAR algorithm explores possible execution paths of the program using its representation in the form of Abstract Reachability Graph (ARG). In a classical CEGAR algorithm a path in a program to be explored is called a counterexample and it means a path to the error state. In CPAchecker tool the basic predicate-based CEGAR algorithm has been extended for explicit-value analysis. In this paper the extended for explicit-value analysis CEGAR algorithm is applied for the task of extracting architecture information from source code. The main contribution of this paper is the application the ideas of counterexample and path feasibility check for the task of extracting the architectural information from source code.

Keywords: architectural information, architecture models, ARINC 653, IMA, CEGAR

DOI: 10.15514/ISPRAS-2018-30(3)-3

For citation: Lesovoy S.L. Extracting architectural information from source code of ARINC 653-compatible application software using CEGAR-based approach. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 31-46. DOI: 10.15514/ISPRAS-2018-30(3)-3

1. Introduction

The purpose of using architecture models is to analyze the system's static and dynamic features during the development process. These features may include real-time performance, resources consumption, reliability etc. This aspect is extremely important while developing complex systems that include both software and hardware components produced by the different suppliers. Using model-based approach at the early stages of the development of the project will help to avoid a waste of time and money for correction of system defects when the system is created. For creation of the architectural models various modeling languages can be used. The most popular ones used for architecture modelling are SysML[1] and AADL[2,3].

The model-based development process includes two major project steps. At the first step, the system model is being created. There are different levels for representation of the system model. The primary focus of this paper is the architectural models. On the second step of the project the system model will be used as input for detailed design and system implementation. This step may also include the model transformations to some intermediate formats used in system design and implementation. In the ideal case, the system model can be transformed to the source code of the system.

It may be useful to analyze and reuse some components of legacy systems during development of new systems. By using a model-based approach it is possible to build an architecture model from the existing source code of the legacy system. This model can be used as for system analysis as well as for reusing some components of the legacy system in the new design. In many cases, it will allow to avoid creation of a new system from scratch.

A process of model creation for existing system is called a model-driven reverse engineering (MDRE). If the source code of a legacy system is available then it is possible to build a system model from its source code. This process contains two steps. The first step is source code analysis. The second step is model transformations to the target output format. This paper describes the first step – source code analysis for application software that is based on Integrated Modular Avionics (IMA) architecture and ARINC 653 specification. The goal of source code analysis is to extract architecture information that is necessary for creation of the architecture model of the system.

The rest of the paper is organized as follows. Section 2 provides an overview of IMA architecture and ARINC 653 specification. It also contains a simple example of source code to be used for further analysis. Section 3 describes the concept of

architectural information in source code, a general approach and a particular algorithm used for extracting architectural information from source code. Section 4 describes the results and outlines the future research and development tasks.

2 IMA

IMA architecture is widely used in avionics industry for implementation the safety critical applications. In IMA systems multiple avionics applications can share resources of a single hardware platform (core module) without any mutual influence. ARINC 653 [4] is a set of documents that define the requirements for software components of IMA systems. The key concept of ARINC 653 is a partition. ARINC 653 compatible Operating System (OS) provides a dedicated portion of memory and predefined time slot within a fixed schedule for each partition. It prevents any affect from software executing in one partition to software in other partitions.

ARINC 653 specification defines that IMA system may include the following software components: core software, application partitions and system partitions. Core software consists of OS and APplication/EXecutive (APEX) interface. The APEX interface defines a set of services provided by the OS for application software. In each application, partition can be allocated only one application. System partitions contain system software that can directly interact with the OS without using APEX interface.

Communications between applications allocated in different partitions is called the interpartition communication. The interpartition communication is only available via communication channels. To access a communication channel the application can use the ports created inside a partition. ARINC 653 supports two port types: sampling ports and queuing ports.

An application software compliant with ARINC 653 specification has a typical structure. Such a software can be located in a single partition or in multiple partitions. To access the various services of ARINC 653 based OS an application software uses function calls defined in the APEX interface. For each partition several processes can be created. One process is responsible for partition initialization. This process creates other processes and various objects. Finally, when the initialization of partition has been finished this process sets the partition to NORMAL state using SET_PARTITION_MODE function call. Since this moment a scheduling for all processes created inside a partition is started. It is important to note that after the initialization of partition has been finished there is no way to create any new processes and objects.

An ARINC 653 process is quite similar to a POSIX thread. To create a process, it is necessary to create a structure that contains the process's attributes and pass it to CREATE_PROCESS function. ENTRY_POINT is an attribute of the process that contains the address of the function that will be called when the process is started. This function implements the application logic of the process and its communication procedures with other processes.

Communications between processes within a single partition is called intrapartition communication. Buffers and blackboards are used for communication between processes inside a partition. Semaphores, events and mutexes are used for process synchronization. Any objects for communication and synchronization can be created using function calls defined in the APEX interface. The processes located inside the same partition can also communicate via global variables.

At the end of this section, a simple example of application software will be demonstrated and explained. The source code fragment of the application software compliant with ARINC 653 specification is shown in Fig.1. Source code fragment in Fig.1 includes three functions: Run_10_Hz, Run_Monitor and main. In function main two processes, one event and three sampling ports are created.

```
static void Run_10_Hz(void) {
    …
    while (1) {
        SET_EVENT ( wakeup, ret );
        READ_SAMPLING_MESSAGE(port_raw_data,
             (MESSAGE_ADDR_TYPE)&sensor_data,
             &len, &validity, &ret);
        // Some operations with data …
        WRITE_SAMPLING_MESSAGE(port_data_out,
             (MESSAGE_ADDR_TYPE)&output_data,
             &len2, &ret);
        PERIODIC_WAIT(&ret_pause); }
}
static void Run_Monitor(void) {
    while (1) {
        WAIT_EVENT ( wakeup, TimeOut, ret );
        RESET_EVENT ( wakeup, ret );
        // Some operations with data …
        WRITE_SAMPLING_MESSAGE( port_status,
             (MESSAGE_ADDR_TYPE)&status_data,
            &len, &ret ); }
}
void main(void) {
    PROCESS_ATTRIBUTE_TYPE Proc_10_Hz_Attributes;
    Proc_10_Hz_Attributes.ENTRY_POINT = Run_10_Hz;
    Proc_10_Hz_Attributes.PERIOD = 100000000LL;
    strncpy(Proc_10_Hz_Attributes.NAME, "Proc_10_Hz",
        sizeof(PROCESS_NAME_TYPE));
    CREATE_PROCESS( &Proc_10_Hz_Attributes, &pid_p0,
        &ret );
    START( pid_p0, &ret );
```

```
    PROCESS_ATTRIBUTE_TYPE Proc_Monitor_Attributes;
    Proc_Monitor_Attributes.ENTRY_POINT = Run_Monitor;
    Proc_Monitor_Attributes.PERIOD =
        INFINITE_TIME_VALUE;
    strncpy(Proc_Monitor_Attributes.NAME, "Proc_Monitor",
        sizeof(PROCESS_NAME_TYPE));
    CREATE_PROCESS( &Proc_Monitor_Attributes, &pid_p1,
        &ret );
    START( pid_p1, &ret );

    EVENT_NAME_TYPE EventName;
    strncpy( EventName, "Wakeup", …);
    CREATE_EVENT ( EventName, wakeup, ret );
    …
    CREATE_SAMPLING_PORT( "RAW_DATA",
        port_size, DESTINATION, period, &port_raw_data, …);
    CREATE_SAMPLING_PORT( "DATA_OUT",
        port_size, SOURCE, period, &port_data_out, …);
    CREATE_SAMPLING_PORT( "STATUS", port_size,
        SOURCE, period, &port_status …);
    SET_PARTITION_MODE ( NORMAL, &ReturnCode );
    return 0;
}
```

Fig. 1. Source code fragment with APEX calls.

For process creation, APEX call CREATE_PROCESS is used. The first argument of CREATE_PROCESS has a type PROCESS_ATTRIBUTE_TYPE. It is a structure that contains attributes for the created process. The ENTRY_POINT attribute is equal to Run_10_Hz for the first process and is equal to Run_Monitor for the second one. Run_10_Hz and Run_Monitor are the function's names that are called when the processes are started.

Below in the main function, APEX call CREATE_EVENT is used to create an event object. An event object has a name Wakeup. Then APEX calls CREATE_SAMPLING_PORT are used to create three sampling ports. These ports have the following names: RAW_DATA, DATA_OUT and STATUS. In the end of the main function APEX call SET_PARTITION_MODE is used to set the partition to the NORMAL state. After that, OS will invoke functions Run_10_Hz and Run_Monitor.

A function Run_10_Hz is called periodically with period 10 milliseconds. This value for period was set in PERIOD attribute during the creation of the first process. Each time when the function Run_10_Hz is called, it activates the event Wakeup, reads a

message from sampling port RAW_DATA, performs some operations with data and writes a message to sampling port DATA_OUT.

Function Run_Monitor belongs to the second process that is an aperiodic. This function waits for event Wakeup, resets it, performs some operations with data and writes a message to sampling port STATUS.

3 Source code analysis

3.1 Architectural information in source code

The main goal of source code analysis in the paper is to extract the architectural information from it. Architectural information in source code of application software compliant with ARINC 653 specification includes the processes in each partition and their attributes, all objects created for interpartition and intrapartition communications and their attributes. It also includes the ways of communications and synchronizations between processes located inside the same partition or in different partitions. If the global variables are used for communication between processes inside partition then these variables also should be considered as architectural information.

The source code fragment in Fig.1 contains the following architectural information: two processes, one event and three sampling ports. Attributes of each process and each object (event, port) are also important architectural information. For synchronization between two processes the event object is used. In the first process APEX call SET_EVENT is used to activate an event. The second process uses APEX call WAIT_EVENT for receiving this event. Sampling ports are used in both processes to communicate with external environment, i.e. with processes allocated in other partitions or with external devices.

The source code of real avionic application can contain hundreds of processes communicating with each other and with external environment via large number of the objects. Extracting such architectural information from source code can be time consuming task. This paper proposes a way to do it automatically. The next sections describe a general approach and a particular algorithm used for source code analysis.

3.2 General approach for source code analysis

For source code analysis, an approach based on Counterexample-guided abstraction refinement (CEGAR) algorithm is used. In CPAchecker tool [5] the basic predicate-based CEGAR algorithm has been extended for explicit-value analysis [7]. CPAchecker is a tool for configurable program analysis (CPA) [5,6] that combines the traditional program analyses and software model checking. In this paper the extended for explicit-value analysis CEGAR algorithm is applied for the task of extracting architecture information from source code. The algorithm is implemented in CPAchecker tool.

The algorithm presented in this paper uses a Control-Flow Automata (CFA) as intermediate representations of the program to be analyzed. CFA is a directed graph

containing nodes and edges. A node corresponds to a program location. An edge corresponds to a certain operation of the program, for example, an assignment statement, a conditional branch or a function call. During the analysis, the algorithm constructs an Abstract Reachability Graph (ARG) using a program CFA. ARG is also a directed graph but its nodes correspond to abstract states of the program. Each abstract state contains a program location, a data state and a call stack. A data state is a mapping between program variables and their values. In data state some program variables may not have the values.

ARG represents possible execution paths of the program. It means that ARG can contain both feasible (real) program paths as well as the infeasible (spurious) paths. The program path is feasible if it can be executed at runtime otherwise it is infeasible. A path in ARG is a sequence of abstract states connected by edges. An abstract state is reachable if there is a feasible program path that contains this state.

3.3 Extracting APEX calls from source code

Before starting the algorithm description, it is necessary to explain some important concepts used by the algorithm. The algorithm constructs the ARG by sequentially adding the new abstract states to it. For the current state the algorithm gets the list of all its successors and adds each of them to ARG. There is an edge between the current state and each its successor.

Target states.

Some edges may correspond to a function call in source code. If this function is defined in APEX interface, the algorithm will need to collect additional information about this function call.

An abstract state in ARG which immediately follows such a function call is called the target state. Any target state has an incoming edge with APEX call. For each target state there is a path in ARG from the initial state to it. The algorithm performs a feasibility check for these paths.

Precision.

Explicit-value analysis tracks values for the program variables. In many cases it is enough to track only a part of program variables that are important for a particular analysis. A set of program variables that are being tracked for the current abstract state is called a precision. Different abstract states may have different precisions. The empty precision means that no variables are being tracked. The full precision means that all variables are being tracked. As described in [6] the value analysis algorithm implemented in CPAchecker can change a precision during the analysis depending on some conditions. It is called a precision adjustment.

An edge in ARG can correspond to a program operation that changes a value of a program variable. For example, an assignment operation changes the value of the left-hand operand, for a function call the values of arguments are assigned to function's parameters, etc. When the algorithm handles an edge between the current state (predecessor) and next state (successor) it uses a precision of the predecessor. If

precision of the predecessor contains the current variable, then the algorithm evaluates and stores its new value in abstract state. The algorithm of analysis can use the values of variables stored in abstract states for different purposes.

Fig. 2 presents a pseudocode of the main algorithm for extracting the architectural information from source code. This algorithm implements a classical CEGAR cycle extended for explicit value analysis [7] and is applied for the task of extracting architectural information from source code.

CFA of the program is used as an input data for the algorithm. The algorithm uses two variables to store the abstract states: “reached” and “waitlist”. A variable “reached” contains the set of abstract states that have been explored already. A variable “waitlist” contains the set of abstract states that have to be explored on the next steps of the algorithm.

At the beginning, the algorithm takes the initial state from CFA and put it to “waitlist”. After that, the external loop of the algorithm begins. The algorithm takes and removes the current state from waitlist. Further the algorithm gets all reachable successors for the current state using function “getAbstractSuccessors”. A pseudocode for the function “getAbstractSuccessors” is shown on Fig.3. The first operation of the function gets all successors (CFA nodes) of the current state. Then the function consecutively handles the edges (function “handleEdge”) between the current state and each its successor. The function “handleEdge” takes two parameters. The first parameter is an edge to be explored. The second parameter is a precision. The precision is taken from the edge predecessor. Depending on the operation in source code that the edge corresponds to, the function “handleEdge” performs the following actions:

- For an assignment operation, the algorithm evaluates a new value for this variable. The new value for a variable will be stored in abstract state if this variable is contained in the precision.
- For a function call, the function’s arguments are assigned to function’s parameters.
- For a conditional branch, a logical value for a condition is evaluated. If the logical value of a conditional branch is equal to FALSE then the function “handleEdge” return FALSE. It means that this successor is not reachable. In all other cases the function returns TRUE and the current successor is added to the list of reachable successors. So, function “getAbstractSuccessors” returns for the current state a list of all its reachable successors.

```
FUNCTION main
INPUT
    CFA of the program;
OUTPUT
```

```
    Architectural information
VARIABLES
    reached – a set of states that have been reached;
    waitlist – a set of states to be explored;
BEGIN
    initState = getInitialState(CFA);
    addStateToWaitlist(initState);
    // Traverse through all CFA nodes.
    LOOP WHILE waitlist ≠ 0 // External loop.
        curState = getAndRemoveStateFromWaitlist();
        // Get all reachable successors of the current state.
        successors = getAbstractSuccessors(curState);
        // Traverse through all reachable successors.
        FOR EACH nextState IN successors // Internal loop.
            IF isTargetState(nextState)
                path = getPathToState(nextState);
                IF isPathFeasible(path) = FALSE
                    // Refine the path.
                    performRefinementForPath(path,
                        reached, waitlist);
                    BREAK // Go to external loop.
                END IF
            END IF
            merge(nextState, reached);
            update(reached);
            addStateToWaitlist(nextState);
        END FOR EACH
    END LOOP
END
```

Fig. 2. The main algorithm for extracting the architectural information from source code

Further in internal loop the main algorithm traverses through all reachable successors for the current state. At this part of algorithm, a successor is called as a “nextState”. The algorithm checks whether a nextState is a target state. If it is a target state, the algorithm calculates a path in ARG from the initial state to the current target state and checks its feasibility using function “isPathFeasible”. In a classical CEGAR algorithm a path in a program to be explored is called a counterexample and it means a path to the error state. In the current algorithm it is just a path to the target state we need to explore.

The algorithm of function “isPathFeasible” is shown in Fig. 4. To check the path

feasibility the algorithm consecutively passes through all edges of the path, starting from the initial state. The algorithm analyses the operations for each edge. To track all program variables, for each state on the path the full precision is set, i.e. the algorithm performs the feasibility check for a path with the full precision.

```
FUNCTION getAbstractSuccessors
INPUT
    curState // Current state.
RETURN
    reachableSuccessors // All reachable successors.
BEGIN
    // Get all successors of the current state.
    allCFASuccessors = getAllSuccessors(curState);
    FOR EACH successor IN allCFASuccessors
        edge = getEdge(curState, successor);
        precision = getPrecisionForState(curState);
        IF handleEdge(edge, precision) = TRUE
            // Add  successor to reachableSuccessors.
            addToSet(reachableSuccessors, successor);
        END IF
    END FOR EACH
    RETURN reachableSuccessors;
END
```

Fig. 3. The algorithm of function getAbstractSuccessors

Each edge on the path is handled with the function “handleEdge” that was already described above. For a conditional branch the function “handleEdge” may return FALSE if logical condition is not satisfied. The path is not feasible if for any edge on the path the logical condition is not satisfied. In this case the function “isPathFeasible” returns FALSE. In all other cases the path is feasible. If the path is feasible then at the last state of the path the values for all program variables assigned on this path are known. The last edge and the last state of the path is passed to a function “handleApexCall”.

```
FUNCTION isPathFeasible
INPUT
    path
RETURN
    TRUE – path is feasible;
```

```
    FALSE – path is not feasible;
BEGIN
    // Traverse through all edges.
    FOR EACH edge IN path
        precision = FULL;
        IF handleEdge(edge, precision) = FALSE
            RETURN FALSE
        END IF
        IF isLastEdge(edge, path) = TRUE
            lastState = getSuccessor(edge);
            handleApexCall(edge, lastState);
        END IF
    END FOR EACH
    RETURN TRUE
END
```

Fig. 4. The algorithm of function isPathFeasible

The last edge contains the information about the APEX call. The last state contains values for all program variables on the path. The function "handleApexCall" extracts all architectural information including the function name for the last APEX call, values for its argument and call stack. It is important to note that the algorithm extracts architecture information only from the APEX calls that belong to a feasible paths. The algorithm collects the architectural information for each APEX call and uses it as output data. The format of the output data will be described in the next section.

If the algorithm has detected that a path is infeasible, then it will refine this path. During refinement procedure the precision for some abstract states of the path are changed by adding variables for tracking. The refinement procedure is described in detail in [7]. Finally, the algorithm will update the ARG in such a way that will eliminate the infeasible path or its part for the further analysis.

At the end of the internal loop the algorithm tries to merge the nextState with already reached states, updates reached states and adds the last explored state (nextState) to "waitlist". These steps are described in details in [7] (see section "Reachability Algorithm for CPA").

The described above steps of internal loop are being repeated for each reachable successor of the current state.

Then the algorithm leaves the internal loop and continues its execution by taking the first step on the main loop. It takes the next state from "waitlist" variable and repeats all steps already described above. The algorithm terminates when all ARG abstract states have been processed.

3.4 Output format

The algorithm keeps the collected architectural information in the internal format. For further processing the architectural information has to be transformed to the external representation. The export format depends on the tool that is used for creation the architecture models. The architectural information can also be exported to human-readable format. In Fig. 5 the architectural information extracted by the algorithm from the source code fragment in Fig. 1 is presented in a human-readable format. The presented architectural information is divided onto sections. The first section contains information about ARINC653 processes. There are two processes with names Proc_10_Hz and Proc_Monitor. Below the process name there are the list of its attributes. On the Fig3 there are only three attributes are presented: PROCESS_ID, ENTRY_POINT and PERIOD. PROCESS_ID is a serial number of the process inside a partition. ENTRY_POINT is a name of the function that is being called when the process is started. PERIOD shows the period's duration in milliseconds. INFINITE_TIME_VALUE in source code corresponds to aperiodic process. The next sections contain the information about other ARINC653 objects created in the source code.

ARINC653_SAMPLING_PORTS section shows three sampling ports and its attributes.

ARINC653_SAMPLING_MESSAGES section shows what processes are using sampling ports for sending (WRITE subsection) and for receiving (READ subsection) messages. For example the port DATA_OUT is used by the first process (function Run_10_Hz) for sending messages.

ARINC653_EVENTS contains information about the events that have been created and used in the source code.

ARINC653_EVENTS section has three subsections: SET_EVENT, WAIT_EVENT and RESET_EVENTS. The name of the subsection corresponds to the APEX call. For example, a subsection SET_EVENT corresponds to APEX call SET_EVENT that activate an event.

```
==ARINC653_PROCESSES==
Proc_10_Hz
  PROCESS_ID: 0
  ENTRY_POINT: Run_10_Hz(0)
  PERIOD = 100 ms
  …
Proc_Monitor
  PROCESS_ID: 1
  ENTRY_POINT: Run_Monitor(1)
  APERIODIC
  …
==ARINC653_SAMPLING_PORTS==
```

```
1) RAW_DATA
  MAX_MESSAGE_SIZE = 128
  PORT_DIRECTION = DESTINATION
  REFRESH_PERIOD = 1000
…
2) DATA_OUT
…
3) STATUS
…
==ARINC653_SAMPLING_MESSAGES==
=WRITE=
1) PORT_NAME=DATA_OUT;
    ENTRY_POINT=Run_10_Hz(0);
2) PORT_NAME=STATUS;
    ENTRY_POINT=Run_Monitor(1);
=READ=
1) PORT_NAME=RAW_DATA;
    ENTRY_POINT=Run_10_Hz(0);

==ARINC653_EVENTS==
=SET_EVENT=
1) EVENT_NAME=Wakeup;
    ENTRY_POINT=Run_10_Hz(0)
=WAIT_EVENT=
1) EVENT_NAME=Wakeup;
    ENTRY_POINT=Run_Monitor(1)
=RESET_EVENTS=
...
```

Fig. 5. The architectural information in human-readable format.

In the analyzed source call there is only one such a call for event with a name Wakeup. The ENTRY_POINT string contains a name of the ENTRY_POINT function where this call was made. In the real code the ENTRY_POINT function is determined using a call stack information. The serial number of the process is shown in parentheses. In the Fig.3 we can see that event Wakeup was set in function Run_10_Hz that belongs to the process with PROCESS_ID equal to 0 (Proc_10_Hz). From the section WAIT_EVENT, we can understand that the function Run_Monitor waits for the event Wakeup using APEX call WAIT_EVENT. The function Run_Monitor belongs to process Proc_Monitor. So, we can see that the event Wakeup is used by two processes for synchronization.

The representation of architectural information in the human-readable format is presented only for explaining the content of such information and is useful mainly for debug purposes. As it was mentioned above for further processing the architectural

information should be transformed to the format that is supported by the external tools.

4 Results and conclusions

The algorithm presented in the paper allows extracting architectural information from source code of ARINC 653-compatible application software. The main contribution of this paper is the application the ideas of counterexample and path feasibility check for the task of extracting the architectural information from source code. In the presented algorithm the task of extracting architectural information from source code has been solved by transforming it into the task of path feasibility check.

The work of the algorithm is demonstrated on the simple example. By this moment the algorithm has been tested on the several software applications that are compatible with ARINC 653 specification. These applications contained up to 50 ARINC 653 process and up to 30 objects for communications.

The next task to be done is to extend the algorithm for extracting from source code the global variables that are used for communication between processes inside partition. It is also necessary to implement the algorithm of transformation of the architecture information to the architecture model.

References

[1]. OMG Systems Modeling Language (OMG SysML™) Version 1.5, 2017.
[2]. [Online]. Available: http://www.omg.org/spec/SysML/1.5/
[3]. SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2017. [Online]. Available: http://standards.sae.org/as5506c/
[4]. Feiler P., Gluch D. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, 2012.
[5]. ARINC Specification 653P1-4. Avionics Application Software Standard Interface Part 1 – Required Services. Published by SAE-ITC, Maryland, USA. August 21, 2015.
[6]. [Online]. Available: https://cpachecker.sosy-lab.org/
[7]. D. Beyer, T. A. Henzinger and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. In Proc, of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 29-38.
[8]. Beyer D., Löwe S. Explicit-State Software Model Checking Based on CEGAR and Interpolation. Lecture Notes in Computer Science, vol. 7793, pp 146-162.

Извлечение архитектурной информации из исходного кода ARINC 653 совместимых приложений с использованием алгоритма CEGAR

С.Л. Лесовой <lesovoy@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Модельно-ориентированный подход к разработке позволяет построить архитектурную модель существующей системы по ее исходному коду. Построенная архитектурная модель существующей системы позволяет проанализировать ее различные статические и динамические характеристики, включая производительность, требуемые аппаратные ресурсы, надежность и другие. Архитектурные модели могут использоваться как для анализа, так и для повторного использования некоторых компонентов существующей системы в новом проекте. Во многих случаях такой подход позволяет избежать построения новой системы с нуля. Для создания архитектурных моделей могут использоваться различные языки моделирования. В данной работе используется язык анализа и проектирования архитектуры (AADL). Данная статья описывает алгоритм извлечения архитектурной информации из исходного кода ARINC 653 совместимых программных приложений. Спецификация ARINC 653 определяет требования к программным компонентам для систем интегрированной модульной авионики (ИМА). Для доступа к различным сервисам операционной системы программные приложения используют прикладной исполняемый интерфейс. Архитектурная информация в исходном коде программных приложений совместимых с требованиями спецификации ARINC 653 включает процессы в каждом разделе, объекты для взаимодействия между процессами внутри и за пределами раздела, а также глобальные переменные. Для анализа исходного кода и получения архитектурной информации необходимо проанализировать все программные вызовы прикладного исполняемого интерфейса. Извлеченная архитектурная информация далее используется для построения архитектурных моделей системы. Для анализа исходного кода используется подход на основе алгоритма CEGAR (уточнение абстракции с помощью контрпримера), широко используемого при верификации программного обеспечения. Алгоритм CEGAR анализирует возможные пути исполнения программы, используя представление программы в виде абстрактного графа достижимости. В классическом алгоритме CEGAR исследуемый путь программы называется контрпримером и означает путь от начала программы до некоторого ошибочного состояния. Для подтверждения наличия ошибки в коде программы алгоритм CEGAR выполняет проверку достижимости для исследуемого пути. В программном инструменте CPAchecker базовый основанный на предикатах алгоритм CEGAR расширен для анализа явных значений переменных. В данной статье расширенный для анализа явных значений переменных алгоритм CEGAR используется для задачи извлечения архитектурной информации из исходного кода приложений. Основной вклад данной статьи заключается в применении идей контрпримера и проверки достижимости пути к задаче извлечения архитектурной информации из исходного кода приложений.

Ключевые слова: архитектурная информация; архитектурные модели; ARINC 653; интегрированная модульная авионика (ИМА); алгоритм CEGAR

DOI: 10.15514/ISPRAS-2018-30(3)-3

Для цитирования: Лесовой С.Л. Извлечение архитектурной информации из исходного кода ARINC 653 совместимых приложений с использованием алгоритма CEGAR. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 31-46 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-3

Список литературы

[1]. OMG Systems Modeling Language (OMG SysML™) Version 1.5, 2017.
[2]. [Online]. Режим доступа: http://www.omg.org/spec/SysML/1.5/
[3]. SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2017. [Online]. Режим доступа: http://standards.sae.org/as5506c/
[4]. Feiler P., Gluch D. Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, 2012.
[5]. ARINC Specification 653P1-4. Avionics Application Software Standard Interface Part 1 – Required Services. Published by SAE-ITC, Maryland, USA. August 21, 2015.
[6]. [Online]. Режим доступа: https://cpachecker.sosy-lab.org/
[7]. D. Beyer, T. A. Henzinger and G. Theoduloz. Program Analysis with Dynamic Precision Adjustment. In Proc, of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, 2008, pp. 29-38.
[8]. Beyer D., Löwe S. Explicit-State Software Model Checking Based on CEGAR and Interpolation. Lecture Notes in Computer Science, vol. 7793, pp 146-162.