C2AADL_Reverse: A model-driven reverse engineering approach to development and verification of safety-critical software

Zhibin Yang, Zhikai Qiu, Yong Zhou, Zhiqiu Huang, Jean-Paul Bodeveix, M Filali

C2AADL_Reverse: A Model-Driven Reverse Engineering Approach for Development and Verification of Safety-critical Software

Zhibin Yang[a,*], Zhikai Qiu[a], Yong Zhou[a], Zhiqiu Huang[a], Jean-Paul Bodeveix[b], Mamoun Filali[b]

[a]*School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China*
[b]*IRIT-Universit de Toulouse, Toulouse, France*

Abstract

The safety-critical system communities have been struggling to manage and maintain their legacy software systems because upgrading such systems has been a complex challenge. To overcome or reduce this problem, reverse engineering has been increasingly used in safety-critical systems. This paper proposes `C2AADL_Reverse`, a model-driven reverse engineering approach for safety-critical software development and verification. `C2AADL_Reverse` takes multi-task C source code as input, and generates AADL (Architecture Analysis and Design Language) model of the legacy software systems. Compared with the existing works, this paper considers more reversed construction including AADL component structure, behavior, and multi-threaded run-time information. Moreover, two types of activities are proposed to ensure the correctness of `C2AADL_Reverse`. First, it is necessary to validate the reverse engineering process. Second, the generated AADL models should conform to desired critical properties. We propose the verification of the reverse-engineered AADL model by using UPPAAL to establish component-level properties and the Assume Guarantee REasoning Environment (AGREE) to perform compositional

*Corresponding author
Email addresses: `yangzhibin168@163.com` (Zhibin Yang), `2427153594@nuaa.edu.cn` (Zhikai Qiu), `zhouyong@nuaa.edu.cn` (Yong Zhou), `zqhuang@nuaa.edu.cn` (Zhiqiu Huang), `bodeveix@irit.fr` (Jean-Paul Bodeveix), `filali@irit.fr` (Mamoun Filali)

Preprint submitted to Journal of Systems Architecture: Embedded Software Design May 25, 2021

verification of the architecture. This combination of verification tools allows us to iteratively explore design and verification of detailed behavioral models, and to scale formal analysis to large models. In addition, the prototype tool and the evaluation of `C2AADL_Reverse` using a real-world aerospace case study are presented.

Keywords: Safety-Critical Systems, Model-Driven Development, Model-Driven Reverse Engineering, AADL, Compositional Verification
2020 MSC: 00-01, 99-00

1. Introduction

Safety-critical systems (SCS) are the systems whose failure could result in loss of life, substantial economic loss, or damage to the environment [1]. There are many well-known examples in different domains such as aircraft flight control, space missions, and nuclear systems. The SCS communities have been struggling to manage and maintain their legacy software systems because upgrading such systems has been a complex challenge. As surveyed by FAA (Federal Aviation Administration), reverse engineering (RE) has been increasingly used in many industries, including aircraft applications [2].

In contrast with forward engineering, reverse engineering can be defined as the process of examining an already implemented software system to create a higher abstraction level representation in a different form. Reverse engineers typically start with a low-level representation of a system (such as source code, or execution traces), and try to build more abstract representations from these (such as architectural models, or use cases, respectively)[3]. The main objective of RE is to provide a better understanding of the software systems current state, which can be used to correct (e.g. fix bugs), update (e.g. alignment with updated user requirements), upgrade (e.g. add new capabilities), or even completely re-engineer the system under study [4].

Generally, reverse engineering a software system is a time-consuming and error-prone process. Its difficult to predict how much time RE will require

and there are no standards to evaluate the quality of the result of RE [4]. To overcome these difficulties, model driven reverse engineering (MDRE) [4, 5, 6] has been proposed to enhance the traditional reverse engineering processes. MDRE is the application of model driven engineering (MDE) principles and techniques to RE in order to generate relevant model-based views on legacy systems, thus facilitating their understanding and manipulation.

There have been several past works on MDRE which can be classified as two categories: *specific* and *general* solutions. This is determined depending on whether they aim to reverse engineer the system from a single technology and/or with a predefined scenario in mind (e.g., a concrete kind of analysis), or to be the basis for any other type of manipulation in later steps of the reverse engineering process [7]. Manev et al. [8] propose a tool, called ITACG (IoT software Analysis and Code-Generation tool), for performing reverse engineering and extraction. This is accomplished by scanning the source code of the target system and extracting architectural information from it, which is stored into a UML model. Umair Sabir et al. [9] present a MDRE framework named Src2MoF to generate UML structural and behavioral diagrams from the Java source code. In order to address several kinds of scenarios relying on different legacy technologies, Hugo Bruneliere et al. [7] give an extensible and generic model driven reverse engineering: MoDisco. MoDisco has three layered architecture i.e. infrastructure, technologies and use case layers. It denes a basic meta-model approach for MDRE based on Knowledge Discovery Meta-model (KDM) specication to provide support for XML, JSP and Java. MoDisco only deals with structural aspects and does not support the MDRE for behavioral aspects from source artifacts.

Most of the existing works of MDRE mainly consider general domains such as desktop or business applications. In this paper, we consider MDRE in the domain of complex embedded systems, especially the safety-critical systems. Complex embedded software systems are typically special-purpose systems developed for control of a physical process with the help of sensors and actuators. They are often the systems requiring a deep combination of software, runtime

operational system and hardware platform. Typical non-functional analysis of the requirements in this domain, such as safety, schedulability, and so on, needs the modeling of architecture, functional behaviors and runtime. These characteristics already make it apparent that complex embedded systems d-iffer from desktop and business applications. Compared with the modeling languages used in the existing works of MDRE such as UML, AADL (Archi-tecture Analysis and Design Language) [10] is a powerful modeling language for complex embedded system, which provides a unified formalism for the modeling of architecture, functional behaviors, and runtime. This paper proposes `C2AADL_Reverse`, a MDRE approach for safety-critical software development and verification. `C2AADL_Reverse` takes multi-task C source code as input, and generates AADL model of the legacy software systems. Moreover, when MDRE exists in the domain of safety-critical systems, validation of the MDRE process and verification of the resulted models are highly desirable because such software systems have to undergo development regulations and certification restrictions. Therefore, the reverse-engineered AADL components become the basis for applying MDD development in the same application domain, and should be analyzed and verified.

1.1. Research Problems

Currently, there are several researches on AADL automatic code generation (i.e. forward engineering). For instance, OCARINA [11] and RAMSES [12] support automatic code generation from AADL to C, Ada and Java. Regarding the reverse generation of AADL models, Wang et al. [13] propose an approach for extracting AADL models from existing embedded software in order to reduce maintenance costs. In an effort to bridge the semantic and syntactic gaps between the two languages, they have defined a set of mapping rules from C to AADL models. For Integrated Modular Avionics (IMA) systems, Lesovoy et al. [14] present an approach to extract the AADL models from source code of ARINC 653-compatible application software. They apply the ideas of counterexample and path feasibility check to the task of extracting the architectural

information from source code. As mentioned before, safety-critical software often run on various embedded platforms, reverse engineering needs to deal with the information such as static structure, dynamic run-time, and functional behavior. However, the existing approaches mainly deal with structural aspects instead of behavioral and run-time aspects of source artifacts. Safety-critical software systems are large and intricate, often constituting hundreds of components. Thus, the challenge is to be able to derive the information about the functional behaviors and the runtime dynamics of a system. In particular, as multi-core processors are widely used in safety-critical software [15], the reverse engineering of multi-task synchronization, mutex, communication, and task scheduling has become an important problem.

Moreover, how to evaluate or measure a MDRE effort? On the one hand, we can use the generated model of MDRE to produce another version of the original software and make the comparison between the two versions to validate the MDRE process. On the other hand, automatic formal verification techniques such as model-checking can be used to analyze the behaviours of the generated model. Since the increasingly size of the source code, formal verification of reverse-engineered AADL models often faces the so-called state-explosion problem. An approach to deal with the state-explosion problem is the use of compositional verification [16, 17, 18] which leverages the structure of the system. The basic idea is to apply divide-and-conquer approaches to infer global properties of complex systems from properties of their components.

To overcome the above-mentioned research problems, we have implemented a complete framework for the proposed approach, that is `C2AADL_Reverse`, as shown in Fig.1. It includes five phases, (1) analysis of the original source code, (2) extraction of an intermediate model, (3) generation of an AADL model, (4) validation of the C2AADL process, and (5) formal verification of the generated AADL model. Compared with the existing AADL RE method, this paper considers more reverse constructions including AADL component structure, behavior, and multi-threaded run-time information. For the validation of the reverse process, we generate a second version of the original software and

compare the two versions of code. Moreover, we propose the verification of the generated AADL model by using UPPAAL to establish component-level properties and the Assume Guarantee REasoning Environment (AGREE) [19, 20] to perform the compositional verification of the architecture.

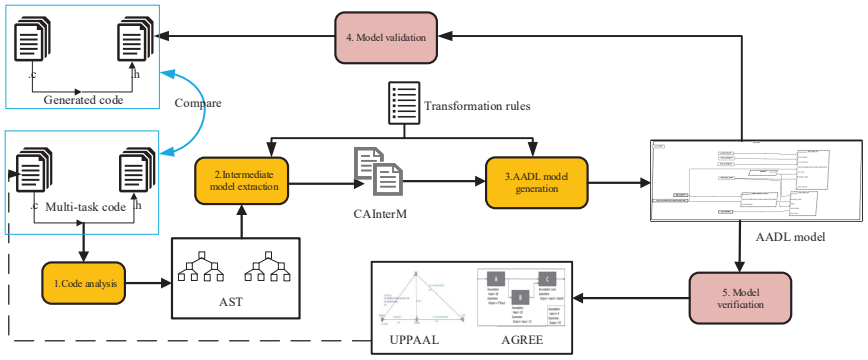


Figure 1: The framework of C2AADL_Reverse

1.2. Main Contributions

The main contributions of the paper can be summarized as follows:

- *A new MDRE approach named C2AADL_Reverse*: The transformation from multi-task C source code to AADL is divided into three parts:
  - Structural aspect: the transformation from global variables, local variables, data types, function definitions and multi-task structures to AADL components;
  - Behavioral aspect: the transformation from function and task execution behavior to AADL behavior annex [21], which involve various types of branch statements, assignment statements, and function call statements;
  - Run-time aspect: the transformation from multi-task communication, multi-task synchronization and mutex, and task scheduling to AADL execution-model properties.

- *Validation and verification approach of C2AADL_Reverse*: Two types of activities are proposed to ensure the correctness of `C2AADL_Reverse`. First, it is necessary to validate the reverse engineering process. Second, the generated AADL models should conform to desired critical properties. We propose the verification of the generated AADL model by using the model checker UPPAAL to establish component-level properties and the AGREE environment to perform the compositional verification of the architecture. This combination of verification tools allows us to iteratively explore design and verification of detailed behavioral models, and to scale formal analysis to large models.

- *The prototype tool*: The `C2AADL_Reverse` prototype tool adopts a modular architecture, which is implemented based on the AADL open source environment OSATE [22], in which an intermediate model is proposed to facilitate the transformation from C source code to AADL.

- *Case study*: A real-world aerospace industrial case, the rocket launch control subsystem, is used to show the feasibility of the method presented in the paper.

1.3. Outline

The rest of this paper is organized as follows. Section 2 introduces the AADL language, the principle of compositional verification, and the AADL compositional verification tool AGREE. Section 3 presents the details of the transformation rules of the `C2AADL_Reverse` approach. Section 4 introduces the formal verification method of the generated AADL models. In section 5, we give the prototype tool. Section 6 illustrates a real-world aerospace industrial case study to show the eectiveness of the `C2AADL_Reverse` approach. Section 7 discusses related work and Section 8 provides concluding remarks and plans for future work.

2. Preliminaries

In this section, we rst provide an overview of the AADL language, and then introduce the principle of compositional verification and the AADL compositional verification tool AGREE.

2.1. AADL

The SAE Architecture Analysis and Design Language (AADL) is a textual and graphical language used to design and analyze the software and hardware architecture of embedded real-time systems. AADL is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform, as shown in Fig.2. *Software components* and their connections are used for software architecture modeling, including Data, Thread, Thread Group, Process, and Subprogram components. *Hardware components* (such as Processor, Virtual Processor, Memory, Device, Bus, and Virtual Bus) and their connections are used to describe the hardware architecture of the system. In addition, AADL describes the run-time state of the system through properties such as Dispatch, Communication, Scheduling, Mode Change, etc., that is, the *Execution model*. Finally, software components, hardware components, and execution models are combined with *System components* to establish a hierarchical system architecture model. Furthermore, it contains an extension mechanism (called an *annex*, e.g. the *Behavior annex* [21][23]) that can be used to extend the language to support additional features. The Behavior annex is defined for the refinement of thread/subprogram behaviors including functional behaviors of thread/subprogram and dispatch behaviors of thread ports. The behaviors of thread/subprogram can also be defined by the traditional programming language such as C, Ada, etc.

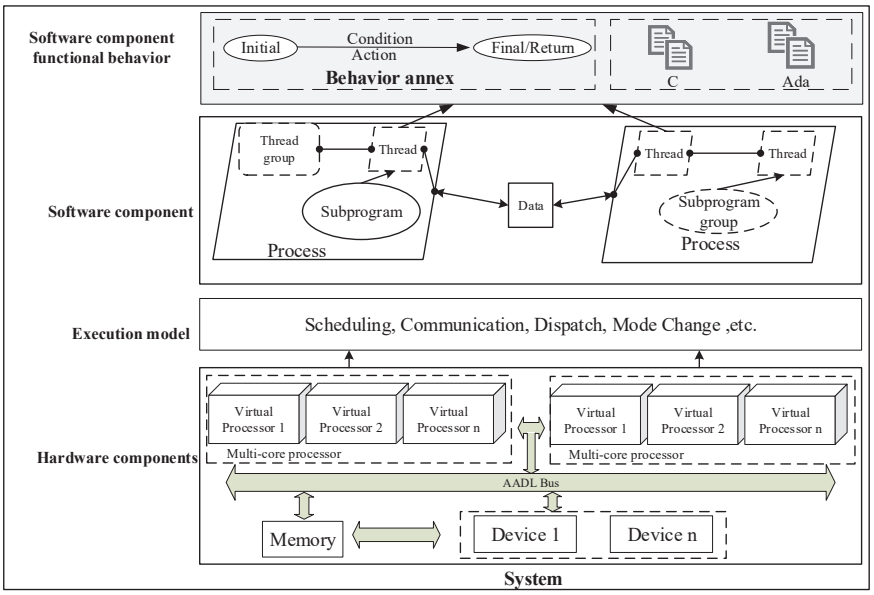


Figure 2: AADL basic modeling concepts

2.2. Compositional Verification and AGREE

Complex embedded systems are always hierarchically organized by using component-based architecture. Automatic formal verification techniques such as model-checking can help to analyze the behaviours of such systems. For instance, thanks to a model-checking tool, one can create a model and analyze all of the behaviors of the components in the architecture model. Actually, most of the time, the architecture model is flattened. Nevertheless, doing so, often faces the so-called state-explosion problem. An approach to deal with the state-explosion problem is the use of compositional verification which leverages the structure of the system. In these techniques, the verification of a composite system is reduced to the verification of its parts.

A well-known compositional approach is based on *assume/guarantee contracts* [16, 17, 18, 24] where each component is annotated with a contract consisting of an assumption specifying how the component expects its environment to behave, and a guarantee specifying the behaviour guaranteed by the com-

ponent if the assumptions hold. The component implementations can be abstracted with contracts that specify the behavioral aspects that are relevant for the system-level properties. Fig.3 shows a toy example of compositional verification, in which component A is decomposed into sub-components B and C, and B is decomposed into D and E. Each component is annotated with a contract $\langle Assume, Guarantee \rangle$. Contracts are refined following the decomposition of components. In general, to verify a system, it is necessary to prove that the implementation of the leaf components satisfy their component-level contracts (either by model checking or through proof methods) and then reason about the system-level contracts starting from the leaf components through all the layers of the architecture.

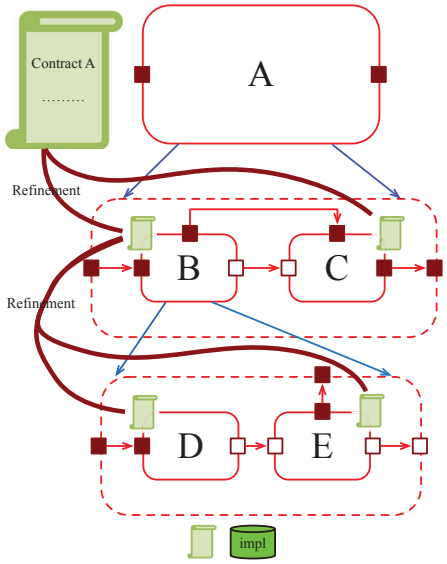


Figure 3: An example of compositional verification

AADL is a good fit for our domain of interest, and provides sufficiently formal notations for modeling and a component-based architecture basis for compositional verification. AGREE [19, 20] is a compositional verification tool for AADL models, which has been integrated into the AADL open-source modeling environment OSATE [22]. AGREE makes use of the AADL annex mechanism,

named AGREE annex [1], to annotate AADL models with contracts. A contract contains a set of assumptions about the component's inputs and a set of guarantees about the component's outputs. The assumptions and guarantees may also contain predicates that reason about how the state of a component evolves over time. When AGREE is proving a system correct, it takes the specied behavior of the system along with its assumptions and the guarantees of any sub-components to verify the guarantees of the system being proven. In addition, AGREE can perform a standard iterative unrolling of the transition relation to find counter-examples, and if one is found, this counter example will be provided to the user. Please note that, AGREE currently mainly handles synchronous architectural models in which execution proceeds in a deterministic discrete sequence of steps [25]. Support for modeling components that execute asynchronously (or quasi-synchronously) will be added to AGREE.

3. C2AADL_Reverse: The MDRE Approach for Generating AADL from C

`C2AADL_Reverse` takes multi-task C code as input, and generates an AADL model. It mainly contains two steps: code analysis and code-to-model transformation.

3.1. Code analysis

AADL is used to describe the model at the architecture level which is higher than C code. In order to fill in the syntactic and semantics gaps between AADL and C, we first build a code structure model from the source code, based on the parsing of the program. The simplified meta-model of the multi-task C code structure is given in Fig.4. The top layer is *Project*. Each project can have several tasks and their communications, i.e., the task layer. The communications among tasks can be achieved by global variables, and synchronization modules (i.e., APIs such as semaphore, mailbox, queue, and so on) provided by OS. In

[1] Available at: http://github.com/smaccm/smaccm

addition, functions which represent sequentially executed source text, can be called by several tasks. Tasks and functions may contain local variables and statements. The *Statement* is an abstract class and is inherited by *Switchstmt*, *Ifstmt*, *ForStmt*, *WhileStmt*, and so on. Here, the *Statement* is duplicated for readability.

Please note that a task is a schedulable unit that can execute concurrently with other tasks and a (shared) function represents sequentially executed instructions that are called by tasks.

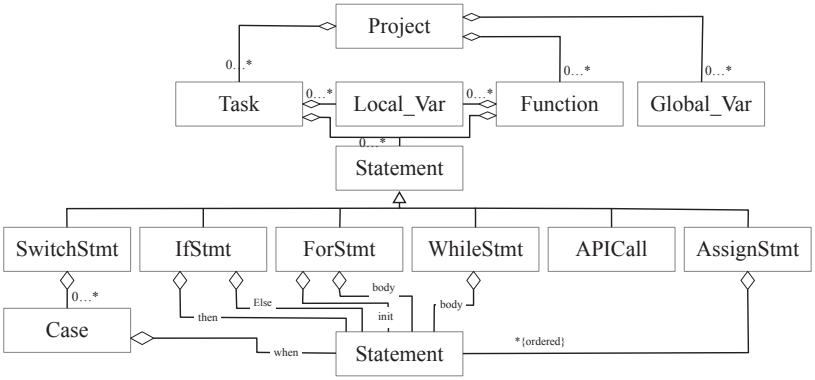


Figure 4: Simplified meta-model of multi-task C code structure

Fig.5 shows the code structure model associated with a part of the source code of the rocket launch control system case study (see Section 6.1). The functions LCU_CT_ContOrd and LCU_CT_MainCont are treated as tasks because the Task_create API adds them to the task module, and the corresponding attributes (such as priority, etc.) are assigned to the tasks through Task_Params. The functions FRAME_LCU_DT_SingleOrder and LCU_CT_ContOrd are treated as shared functions. Each of them represents a sequential execution and can be called by multiple tasks. For instance, the task FRAME_LCU_DT_NetData calls the function FRAME_LCU_DT_SingleOrder to send data to the task LCU_CT_MainCont through mailbox.

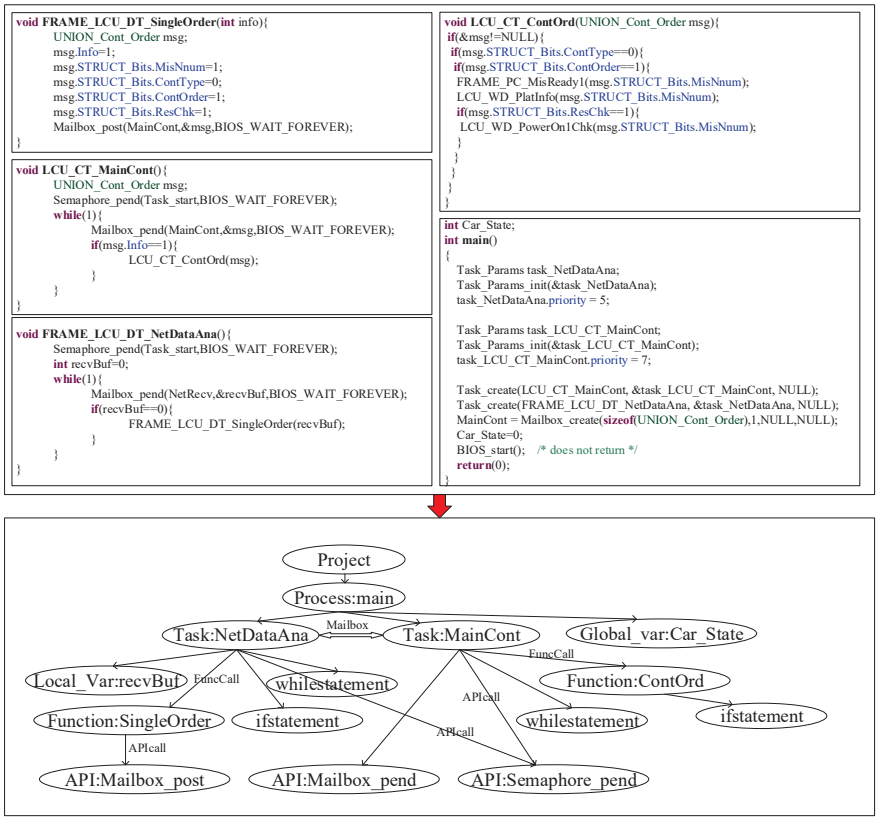


Figure 5: An example of the construction of code structure model

3.2. Code-to-Model transformation

The transformation from C to AADL consists of three parts: structure, behavior, and run-time information.

3.2.1. Structure transformation

As shown in Table 1, the structure transformation rules include the transformations from basic data types and composite data types to AADL data components, from function definitions to AADL subprogram components, and from task structures to AADL thread components. Since variables in C language and data components in AADL describe similar data types, we map variables

to data components. In order to make the structure of AADL clearer, we encapsulate the data components into two types of packages: one package represents the basic data type named Base_Types, and the other package represents the extended data type (composite data type) named User_Define. For basic data types, they can be mapped into AADL data types straightforward, while the pointer is represented as a data access feature of component. AADL can also express signed and unsigned integers. For composite data types with internal implementations, the elements of the composite data are mapped to subcomponents declared in the implementation of data component. In addition, the function definitions and task structures are mapped to AADL subprogram and thread components respectively.

Table 1: The transformation rules of structure information

C language	AADL
int a; int *a	Base_Types::Integer;
	requires data access Base_Types::Integer
char a; char *a	Base_Types::Character
	requires data access Base_Types::Character
bool a; bool *a	Base_Types::Boolean
	requires data access Base_Types::Boolean
float a; float *a;	Base_Types::Float
	requires data access Base_Types::Float
struct_name a;	User_Defined::struct_name.impl;
struct_name *a;	requires data access User_Defined::struct_name.impl;

signed int a; unsigned int b;	Base_Types::Integer_32; Base_Types::Unsigned_32; **data** Integer_32 **extends** Integer **properties** Data_Model::Number_Representation ⇒ Signed; **end** Integer_32; **data** Unsigned_32 **extends** Integer **properties** Data_Model::Number_Representation ⇒ Unsigned; **end** Unsigned_32;
struct dataname{ type_spec var_name; }; enum; union;	**data** dataname **properties** Data_Model::Data_Representation⇒ (Struct/Union/Enum); **end** dataname **data implementation** dataname.impl **subcomponents** var_name: data package_name::type_spec **end** dataname.impl
function definition	subprogram component
task structure	thread component

3.2.2. Behavior transformation

The behavior of a function or a task is defined by the statements inside the body of the function or task. The statements always include *assignment*, *if*, *switch-case*, *for*, *while*, and *function call*.

To be more intuitive, we present the description of the AADL behavior annex as a graphic automaton. The upper label of the state transition line indicates

the guard condition, and the lower one indicates the execution action. The **transition** defines transition from a source state (s_i) to a destination state (s_j), a transition out of a source state is initiated once the **guard** condition is satisfied, and **action** represents the action performed when a transition is taken.

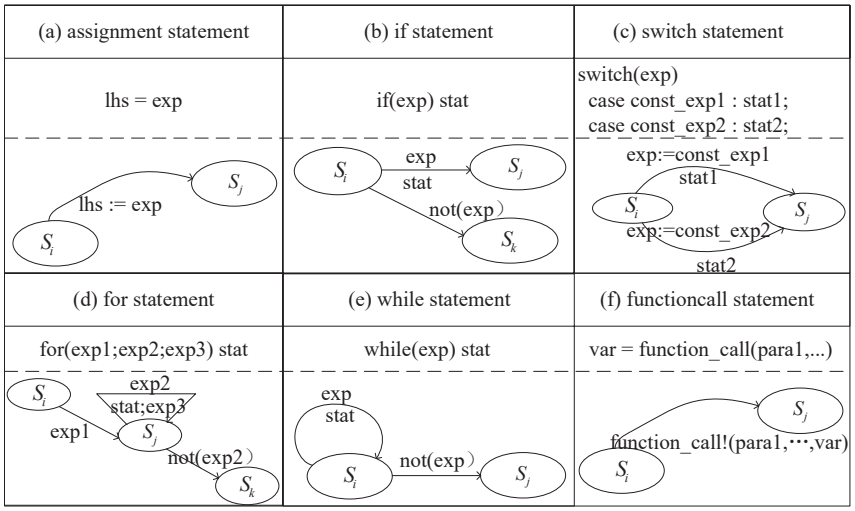


Figure 6: The transformation rules of behavior

The transformation rules are shown in Fig. 6:

(a) shows that the *assignment* statement is transformed to the action of the transition in the AADL behavior annex. Note that the assignment operator (=) of C language is transformed to the assignment operator (:=) of AADL.

(b) and (c) represent the transformation rules of branch statement. The branching conditions (*exp*) of *if* statement and *switch* statement are transformed to the *guard* of the transition. The execution statement (for instance *stat*, *stat1*, *stat2*) is mapped to the *action* of the transition.

(d) and (e) represent the transformation rules of loop statements. A *for* statement always contains four parts: initialization expression (*exp1*), loop condition (*exp2*), iteration expression (*exp3*), and loop content (*stat*).The initialization expression is mapped into an *action* of state transition before the loop;

the loop condition is mapped into the *guard* of the loop state transition; the loop content (*stat*) and the iterative expression (*exp2*) are transformed into the *action* of the loop state transition. The *while* statement is expressed by an automaton with a loop state, on which its *guard* is *exp* and the *action* is *stat*.

(f) represents the transformation rule of *function call* statement. A function call statement always includes three parts: the called function name (*function_call*), the input parameter (for instance *para1*) and the variable that receives the return value (*var*). The called function name is transformed to AADL subprogram name. Note that, in AADL behavior annex, the notation *!* is used to indicate a subprogram call.

The transformations will be more complex in presence of inner statements. We recursively apply transformations for each kind of statements until complex statements have been eliminated.

3.2.3. Run-time information transformation

It needs to consider the use of the platform's API in the transformation of the run-time information. Without loss of generality, in this paper we consider TI SYS/BIOS Real-time Operating System (SYS/BIOS) [26] which is broadly used in the aerospace domain. The transformation rules mainly include task communication, task synchronization, and task scheduling. As shown in Fig.7, to facilitate the presentation of the transformation rules, we use visual graphics to represent AADL models.

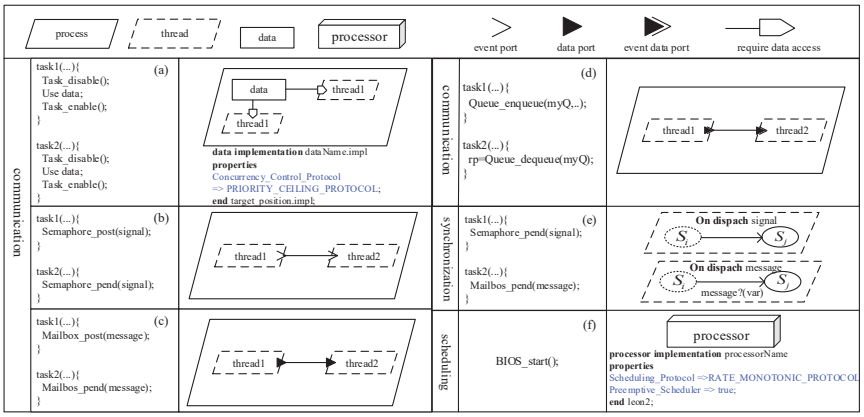


Figure 7: The transformation rules of run-time information

First, four communication mechanisms are considered: shared data, semaphore, mailbox, and queue.

(a) represents the transformation rules for a *shared data*. The shared data is always a global variable, which is accessed by several tasks. The corresponding generated AADL model contains threads and a data component. The threads access the data component through a *Data Access* feature. In addition, a Concurrency_Control_Protocol property associated with the shared data component determines the particular concurrency control mechanism to be used, such as Priority_Inheritance, Priority_Ceiling, and so on.

(b) indicates the transformation rule for a *semaphore*. Semaphores can be declared as either counting or binary semaphores. They can be used for task synchronization and mutual exclusion. Thus, two tasks communicating by sending and receiving semaphores are transformed into two AADL threads connected through *Event ports*.

(c) gives the transformation rule for a *mailbox*. Mailboxes can be used to pass buffers from one task to another. The mailbox communication mechanism is represented in AADL by connections through *Data ports*.

(d) indicates the transformation rule for a *queue*. Queue provides support for creating lists of objects. The queue mechanism is mapped to AADL thread connections through *Event Data ports*. Please note that, in SYS/BIOS, a queue can be implemented as a doubly-linked list. However, in order to be compatible with the AADL semantics, we mainly consider the basic FIFO queue.

Second, in (e), we represent the transformation rule for the synchronization mechanism. SYS/BIOS provides a fundamental set of functions for intertask synchronization (such as semaphore pend(), mailbox pend()). When count of semaphore is less than 0 or the buffer of mailbox is unavailable, tasks will be blocked when acquiring a message. As mentioned in Section 2.1, except for expressing the functional behaviors of a thread/subprogram, the behavior annex can also describe the dispatch behaviors of thread, which is a good way to refine the synchronization between different threads. Thus, the inter-task synchronization is represented by the *On Dispatch* condition on the transition of the AADL behavior annex (the dotted oval state in (e)).

At last, in (f), we represent the transformation rule for the scheduling mechanism. SYS/BIOS dynamically schedules and preempts tasks based on the tasks priority level and the tasks current execution state. In AADL, processor components with properties such as Rate_Monotonic_Protocol and Preemptive_Scheduler⇒true describe the same semantics. So we map this scheduling mechanism to processor components with scheduling properties.

4. Validation and Verification Approach of C2AADL_Reverse

Two types of activities are required to ensure the correctness of the reverse engineering approach proposed in this paper. First, it is necessary to validate the reverse engineering process. Second, the generated AADL models should conform to desired critical properties.

The validation and verification approach of `C2AADL_Reverse` is shown in Fig.8. At first, the validation of the reverse engineering is performed by code

comparison. We use the AADL code generator OCARINA [11] to generate another version of code from the generated AADL model. If the generated version of code is close enough to the original one, the reverse-engineering effort was adequate. Second, we consider a compositional verification method for the generated AADL model, i.e., we assemble verification of system-level properties by using UPPAAL to establish component-level properties and AGREE to perform the compositional verification of the architecture.

Section 6.2 will give the validation of C2AADL_Reverse through the use of the rocket launch control system case study. Here, we mainly give the compositional verification method for the generated AADL model.

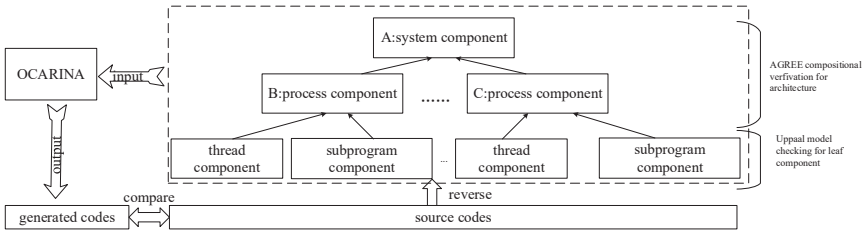


Figure 8: Validation and Verification Approach of C2AADL_Reverse

4.1. Model checking of leaf components

As the first step of compositional verification of a component-based system, it is necessary to prove that the behaviors of the leaf components satisfy their component-level contracts. Thus, in this paper, UPPAAL is used to verify AADL Behavior annex models. UPPAAL is a tool supporting the simulation and the verification of models defined by a set of timed automata communicating through synchronous channels and shared variables. The state transition of the Behavior annex is similar to the automata in the UPPAAL model. Therefore, the transformation from AADL behavior annex into UPPAAL model is straightforward.

First, the formal definitions of behavior annex and time automaton are stated as follows:

Definition 1 Behavior annex $BA = \langle S, S_0, V, G, A, T \rangle$, where:

- S is a set of states, in which a state may be qualified as *initial* state, *final* state, or *complete* state, or combinations thereof. A state without qualification will be referred to as *execution* state.
- S_0 is the initial state, $S_0 \in S$.
- V is a set of local variables.
- G ($Guards$) is a set of state transition conditions.
- A ($Actions$) represents the actions that need to be performed during state transitions.
- T is a set of state transitions, $T \subseteq S \times (G \times A) \times S$.

Definition 2 A timed automaton [27] $TA = \langle L, l_0, V, C, A, I, E \rangle$, where:

- L is a set of locations;
- l_0 is the initial state, $l_0 \in L$;
- V is a set of variables;
- C is a set of clocks;
- A is a set of actions;
- $I = L \to B(C)$ assigns invariants to locations, $B(C)$ is the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$ and $\bowtie \in \{>, \leq, =, \geq, <\}$;
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset.

On the one hand, BA is transformed into TA. Globally, the state set of BA is transformed to the states of TA ($BA.S \to TA.L$); the initial state of BA corresponds to the initial state of TA ($BA.S_0 \to TA.l_0$); the local variable of BA is transformed to the variable in TA ($BA.V \to TA.V$); the state transition

of BA is transformed to the state transition of TA ($BA.T \rightarrow TA.E$); the operation on time of BA such as *computation()* is converted into the invariant(I) of the state in TA, for instance $computation(5ms)$ is represented by $cl <= 5$; there are two clocks in the C of TA: cl and $global_{cl}$, cl records the time of the operation performed by each state transition and $global_{cl}$ records the time when the automaton is executed.

On the other hand, the $Assume$ of the contract ($\langle Assume, Guarantee \rangle$) of the component is transformed into the initialization operation when the time automaton starts to execute, and the *Guarantee* of the contract is transformed into the property written in TCTL language that needs to be verified by UPPAAL. As mentioned in Section 2.2, AGREE makes use of the AGREE annex to annotate AADL models with contracts. The underlying formalism of the AGREE annex language is a subset of Past-Time Linear Temporal Logic (PLTL). It would be highly desirable if the formalism of the contracts expressed with the AGREE annex is consistent with the formalism of the properties used in UPPAAL. Now, we manually generate the required component-level properties for verification in UPPAAL from the contracts. In the future, the translation between the AGREE contracts and the UPPAAL properties will be automated.

Fig.9 shows an example of the transformation from BA to UPPAAL. It verifies that the component meets the contract without deadlock and timeout.

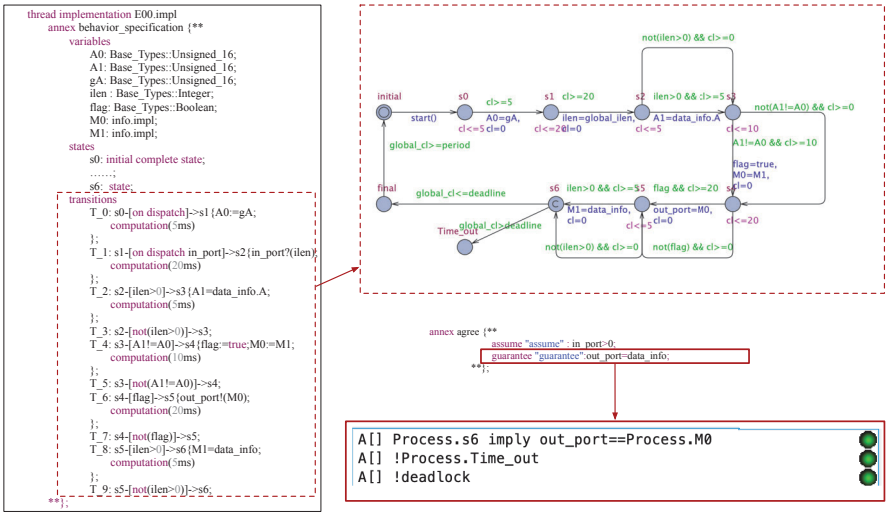


Figure 9: The transformation of BA to UPPAAL

4.2. Compositional verification of AADL architecture model

To formally argue that the system satisfies its requirements, assume-guarantee contracts provide an appropriate mechanism for capturing the information from requirements or source code to reason about system-level properties. A contract specifies precisely the information that is needed to reason about the component's interaction with other parts of the system. Furthermore, the contract mechanism supports a hierarchical decomposition of the verification process that follows the natural hierarchy in the system model.

In the AGREE framework, it uses the AADL AGREE annex to specify the contracts of the component of each layer and the underlying formalism of the AGREE annex is the past-time operator subset of PLTL. We thus establish that the properties of the top-level system are proved given that the properties of the lowest layer i.e. leaf-level components are true (by model checking with UPPAAL shown in Section 4.1). As shown in Fig.10, we take the LCU_CT_ContOrd function of the rocket launch control system case study (Section 6.2) as example. It is reversed into a component containing two sub-components, and the sub-component contract is used to prove whether the upper-level component

meets the requirements.

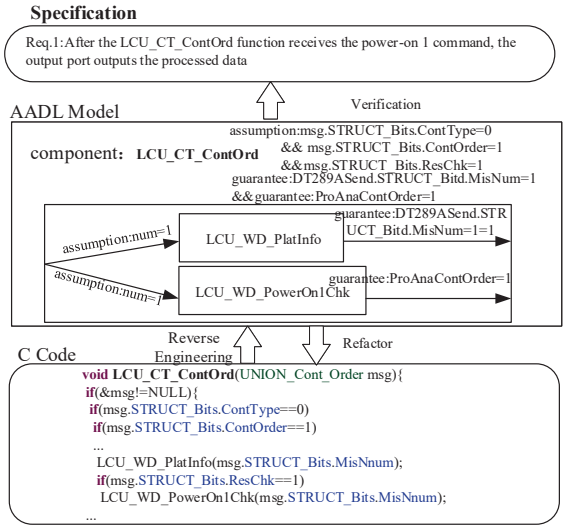


Figure 10: The framework of compositional verification for architecture

5. Prototype Tool Support

As mentioned above, the `C2AADL_Reverse` prototype tool adopts a modular architecture, which is implemented with the Eclipse plug-in technology. Moreover, an intermediate model is proposed to facilitate C code information extraction and AADL model generation.

5.1. Intermediate model

As shown in Section 3.1, the meta-model of the multi-task C code structure represents the abstract concepts of source code, which is helpful for the description of the transformation rules. Here, we propose an intermediate model named CAInterM to facilitate the implementation of the `C2AADL_Reverse` tool. The intermediate model contains the elements of the meta-model of the multi-task C code structure, the file structures, and the connection information among tasks or functions. The intermediate model CAInterM is shown in Fig.11.

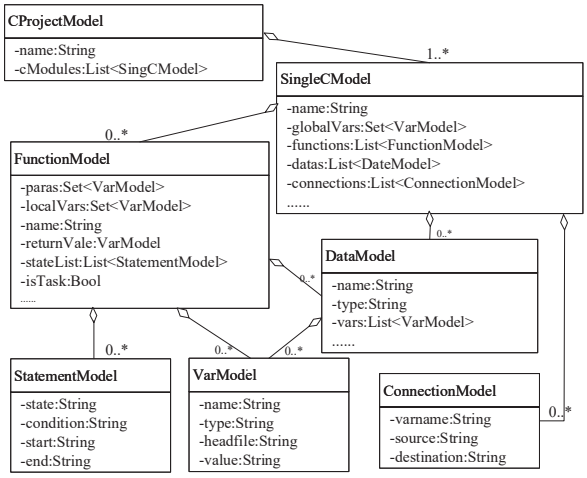


Figure 11: Intermediate model: CAInterM

The CProjectModel records the content of the project in the meta-model of the multi-task C code structure, and is used to generate the system components of the AADL model. SingleCModel represents a C file structure, which is used to generate an AADL file. FunctionModel corresponds to Function or Task in the meta-model. The isTask attribute in FunctionModel is used to mark whether it is a Task. If isTask is true, a thread component is generated. Otherwise a subprogram component is generated. The Datamodel records the contents of the Global_Var in the meta-model, corresponding to the data components of the generated AADL model. StatementModel stores the content of IfStmt, WhileStmt, AssignStmt, ForStmt and SwitchStmt in the meta-model, which is used to generate the states and transitions of the AADL Behavior annex. VarModel represents a Local_Var in the meta-model, which is used to generate variables in BA. ConnectionModel records the source tasks and destination tasks or functions of communication-related APICall in the meta-model, which is used to generate connections between components. Moreover, CAInterM represents the multi-task C code as a hierarchical structure to facilitate the generation of AADL models.

5.2. Implementation of C2AADL_Reverse tool

The `C2AADL_Reverse` tool is built on the open source AADL modeling environment OSATE. The framework of the tool is shown in Fig.12. The tool adopts a modular architecture, including C2AST, AST2CAInterM and CAInterM2AADL. The C2AST and AST2CAInterM modules are mainly responsible for extracting the structure, behavior, and runtime information from the source code to the intermediate model, and the CAInterM2AADL module transforms the information stored in the intermediate model to the AADL model.

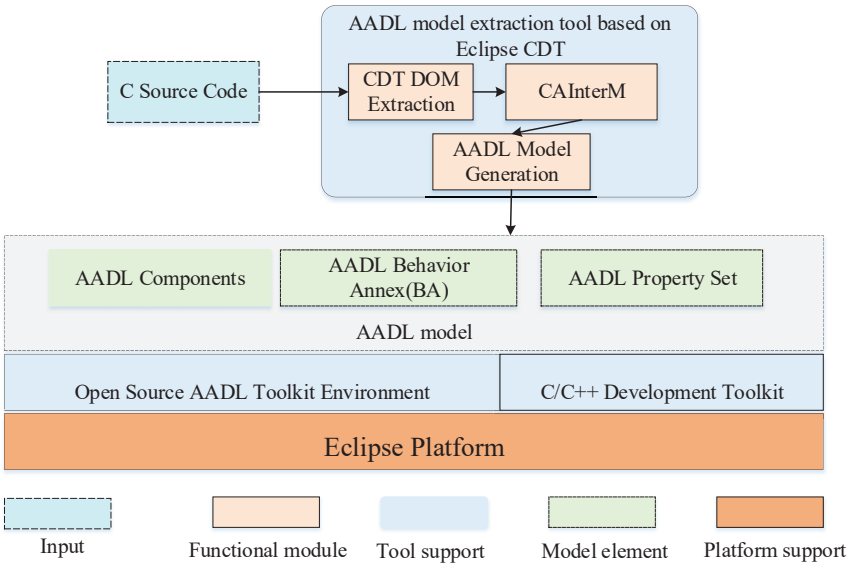


Figure 12: The framework of C2AADL_reverse tool

The tool is implemented as a plug-in of OSATE, which is shown in Fig.13. It mainly provides three functionalities, such as project management, reverse engineering, code/model viewing and modification.

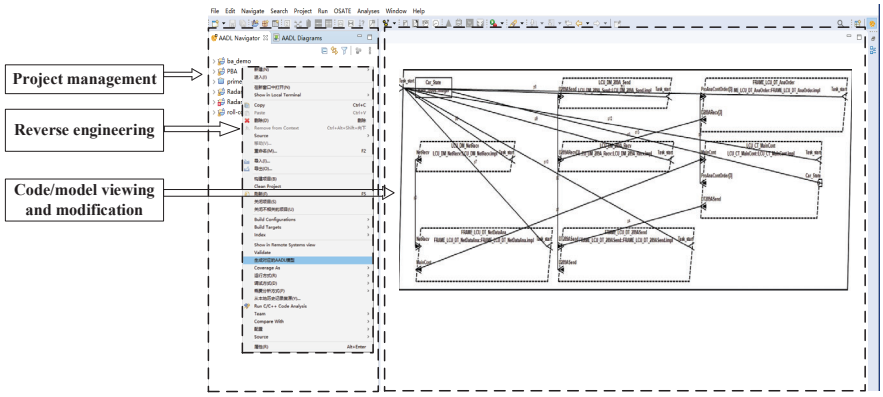


Figure 13: C2AADL_Reverse tool

6. Evaluation

6.1. Industrial Case Studies

The Rocket Launch Control System (RLCS) is a critical subsystem of the rocket launcher system (RLS). The function of the RLCS is to control the rocket to perform various operations and automatically execute rocket launch operation from launch command receipt to the the time when the rocket leaves the launcher. RLCS running on the launch control unit (LCU) computer and the LCU computer interacts with other modules of the RLS through a bus and/or a network. The RLCS can ensure the normal execution of the launch of the rocket through a series of hardwaresoftware interactions. A simplified architecture of RLCS is shown in Fig.14, which mainly includes a management chassis, a launcher module, a power supply chassis, a power-on control module, a launch control unit detection board and a rocket control module. The power-on control module includes three sub-modules: power-on 1, power-on 2 and power-on 3. In addition, the launcher module controls several devices such as exhaust cover and hatch, etc.

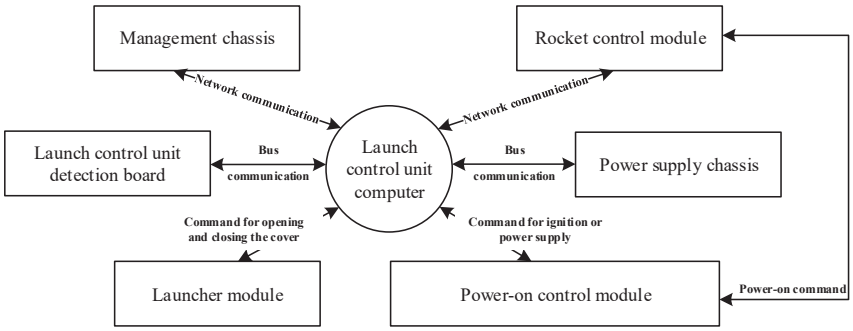


Figure 14: The rocket launch control system

The structure of RLCS can be divided into four layers, namely driver layer, driver management layer, process layer, and main control layer, among which:

- **Driver layer**: Describes the lowest-level module, which is responsible for the interaction of software and hardware, such as network communication, bus communication, and sending and receiving operations. The driver layer processes the received data and sends them to the tasks of the driver management layer.

- **Driver management layer**: The driver management layer analyzes and assembles the data corresponding to the different communication methods of the driver layer, and sends the assembled data to the process layer, such as the receiving and sending of power-on command;

- **Process layer**: The process layer parses the assembled data sent by the driver management layer into various commands. The functions in the main control layer are realized by the command processing in the process layer, such as power control, control of hatch covers, cartridges, thermal batteries, etc;

- **Main control layer**: The main control layer starts the task of the driver layer according to the command to perform corresponding function execution, such as sequence maintenance processing, electric blast tube in-

spection, data receiving processing, sending control parameter processing, command execution processing, and so on.

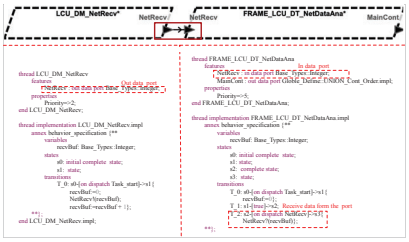
6.2. Results and Analysis

6.2.1. C2AADL transformation

We use the `C2AADL_reverse` tool to reverse a part of source code of the RLCS to AADL model. This section takes power-on control as an example to analyze the generated AADL model from three perspectives: structure, run-time properties, and behaviors.

Fig.15(a) shows the top-level system structure of the generated AADL model. The AADL model is consistent with the RLCS system introduced in Section 6.1, several thread components (such as *LCU_DM_NetRecv*,*FRAME_LCU_DT_NetDataAna*, etc) implement the power-on function of the system through port communication. Fig.15(b) shows the communication between multiple components, thread *LCU_DM_NetRecv* receives data from the network and sends it to the thread *FRAME_LCU_DT_NetDataAna* via the *NetRecv*. Fig.15(c) shows an individual component, including input/output parameters, data access and behavior annex.

(a) The system structure



(b) Multi-component communication

(c) An individual component

Figure 15: Generated AADL models of RLCS

Through the reverse construction of the rocket launch control system, the data statistics of the AADL model of the system are shown in Table 2.

Table 2: The statistics of the AADL model of the launch control system

	AADL model (line)	Threads	Subps	Coverage
Exhaust cover control	1800+	4	14	93%
Rocket hatch control	1600+	3	12	92%
Control of rocket launch preparation/cancellation	1600+	4	11	94%
Rocket power-on control	4000+	18	32	93%
Control of rocket hatch unlock/lock	2000+	6	13	95%

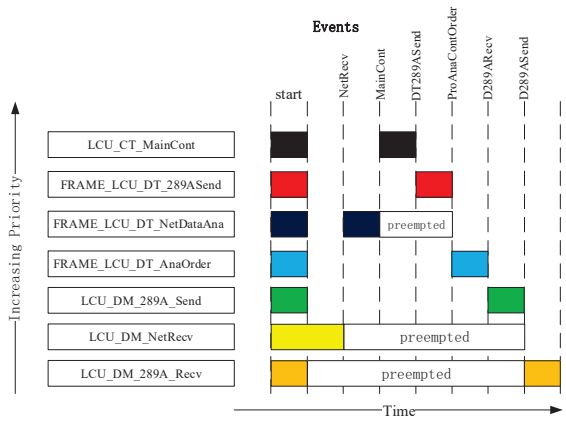
Thermal battery activation control	1800+	6	15	93%
Control of safety mechanism unlock/lock	1900+	6	15	95%
Rocket ignition control	1700+	4	13	93%
Rocket power-off control	1200+	4	9	94%
Rocket launch control system	17600+	55	134	94%

The reason why the coverage rate of the generated model does not reach 100% is that some codes cannot be expressed in the behavior annex, such as bit operation and type mandatory conversion, etc. However, AADL supports to describe the behaviors of thread/subprogram with source code directly, through properties such as Source_Language, Source_Name, Source_Text, etc. Thus, it is used as a complementary of the results of `C2AADL_Reverse`.

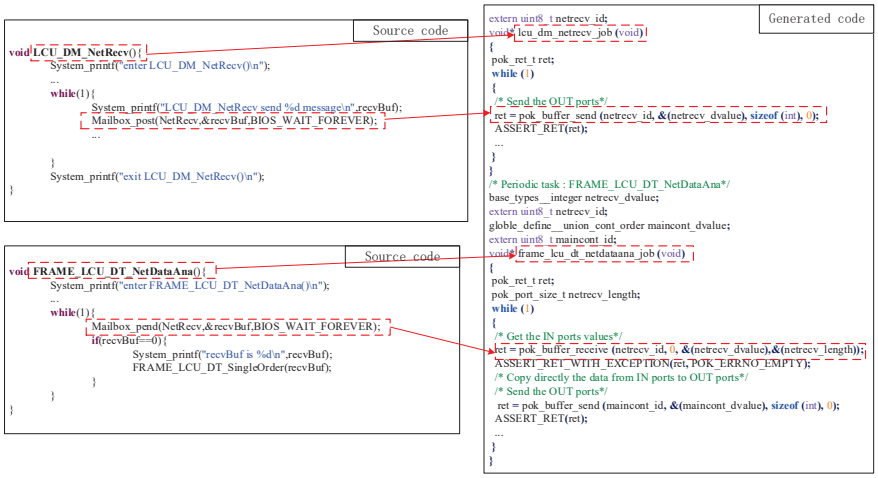
6.2.2. Validation and verification

1) **Validation of C2AADL_reverse**

We refined the generated AADL model by adding processors, virtual processors and scheduling mechanisms, and used the C code generator of OCARINA [28] to produce an executable version from our model of RLCS. We can compare the generated codes with the original ones by using code reviewing and executing. Fig.16(a) shows that the execution sequence of the tasks in the generated code is consistent with the original code. Fig.16(b) shows the comparison between a part of the original codes and the generated codes. OCARINA mainly considers the POK operational system [29]. The pok_buffer_send and pok_buffer_receive used for inter-task communication in the POK OS are consistent with the Mailbox_post and Mailbox_pend in SYS/BIOS. Thus, the `C2AADL_Reverse` method can generate adequate models.



(a) The execution scenario of generated code



(b) The comparison between OCARINA generated code and the original program

Figure 16: Validation of reverse process

2) **Verification of the generated AADL model**

- Requirements Formalization

The requirement document of RLCS is got from our industrial partner. It has more than 300 pages and has ten sections, such as a Launch control system

(LCS), Exhaust cover control (ECC), Rocket hatch control (RHC), and Rocket power-on control (RPC), etc. These requirements were developed hierarchically following the system architecture. A simplified requirements hierarchy is shown in Fig.17.

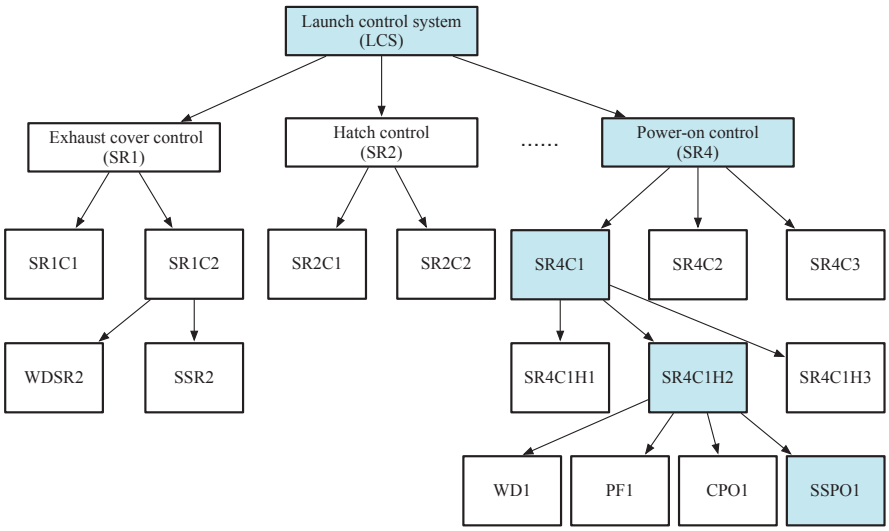


Figure 17: A simplified LCS requirements hierarchy

In order to illustrate the relationship between the requirement decomposition and the system structure decomposition of the RLCS, this paper takes the rocket power-on control (SR4) as an example to illustrate the further detailed decomposition of the requirement. Rocket power-on control consists of three sub-components: rocket power-on 1 control (SR4C1), rocket power-on 2 control (SR4C2) and rocket power-on 3 control (SR4C3), and each power-on control is used to control different battery components. For instance, power-on 1 can be decomposed into rocket self-check result query function (SRC1H1), rocket power-on 1 function (SR4C1H2), and rocket type identification code query func-tion (SRC1H3). Furthermore, SR4C1H2 can be divided into watchdog function 1 (WD1), power-off 1 (PF1), continuous power-on 1 (CPO1), and single-step power-on 1 (SSPO1). AGREE currently mainly handles synchronous architec-tural models in which execution proceeds in a deterministic discrete sequence

of steps, so we consider the model which execute synchronously. Based on the requirement decomposition, the requirements of each component are formalized as contracts. Several contracts are shown in the following paragraphs.

Requirement 1: When the task *FRAME_LCU_DT_NetDataAna* of the driver management layer receives the data sent by the task *LCU_DM_NetRecv* of the driver layer, it sends the encapsulated data to the main control task (contType=0 means single-step power-on, MisNnum=1 means power-on 1). It is formalized in AGREE annex (R1G) shown as follows:

```
thread FRAME_LCU_DT_NetDataAna
    features
        NetRecv : in data port Base_Types::Integer;
        MainCont : out data port Globle_Define::UNION_Cont_Order.impl;
    properties
        Priority=>5;
    annex agree {**
    property judge_MainCont=
    MainCont.Info=1 and MainCont.STRUCT_Bits.MisNnum=1 and MainCont.STRUCT_Bits.ContType=0;

    assume "A:FRAME_LCU_DT_NetDataAna receive data from NetRecv" : NetRecv = 0;
    guarantee "G:FRAME_LCU_DT_NetDataAna send data to MainCont" : judge_MainCont;
**};
end FRAME_LCU_DT_NetDataAna;
```

Requirement 2: The task *LCU_CT_MainCont* of the main control layer, after receiving the data of single-step power-on 1, calls related functions to process the data, sends an order to the task *FRAME_LCU_DT_289ASend* of the driver management layer, and sends a data with the same MisnNum to the task *FRAME_LCU_DT_AnaOrder* of the driver management layer. It is formalized as the contract R2G:

```
thread LCU_CT_MainCont
    features
        MainCont : in data port Globle_Define::UNION_Cont_Order.impl;
        DT289ASend : out data port Globle_Define::UNION_Data_Bind_Info.impl;
        ProAnaContOrder : out data port Base_Types::Integer;
        Car_State:requires data access Globle_Define::Car_State.impl;
    properties
        Priority=>7;
    annex agree {**
        property judge_MainCont=
            MainCont.Info=1 and MainCont.STRUCT_Bits.MisNnum=1 and MainCont.STRUCT_Bits.ContType=0;

        property judge_DT289ASend=
            DT289ASend.STRUCT_Bits.MisNnum=1 ;

assume "A:LCU_CT_MainCont receive cont from NetDataAna" : judge_MainCont;
guarantee "G:LCU_CT_MainCont send order to FRAME_LCU_DT_289ASend" : ProAnaContOrder=1;
guarantee "G:LCU_CT_MainCont send data to FRAME_LCU_DT_AnaOrder" :  judge_DT289ASend;
    **};
end LCU_CT_MainCont;
```

Requirement 3: After the task *FRAME_LCU_DT_AnaOrder* of the driver management layer receives the data from the main control layer, it processes the data and sends the processed data to the corresponding task at driver layer for specific execution operations. It is formalized as the contract R3G shown as follows:

```
thread FRAME_LCU_DT_AnaOrder
    features
        ProAnaContOrder : in data port Base_Types::Integer;
        D289ARecv : out data port Base_Types::Integer;
    properties
        Priority=>4;
    annex agree {**
        assume "A:FRAME_LCU_DT_AnaOrder receive data from LCU_CT_MainCont":ProAnaContOrder=1;
        guarantee "G:FRAME_LCU_DT_AnaOrder send data to LCU_DM_289A_Recv":D289ARecv=1;
    **};
end FRAME_LCU_DT_AnaOrder;
```

- Compositional verification based on AGREE

AGREE is used to perform compositional reasoning on the rocket power-on control subsystem. The results are shown in Fig.18, in which 70 contracts are verified.

Property	Result
Verification for Process_main.impl	70 Valid
Contract Guarantees	8 Valid
LCU_DM_NetRecv assume: A:LCU_DM_NetRecv	Valid (0s)
FRAME_LCU_DT_NetDataAna assume: A:FRAME_LCU_DT_NetDataAna receive data from NetRecv	Valid (0s)
LCU_DM_289A_Send assume: A:LCU_DM_289A_Send	Valid (0s)
LCU_DM_289A_Recv assume: A:LCU_DM_289A_Recv	Valid (0s)
FRAME_LCU_DT_289ASend assume: A:FRAME_LCU_DT_289ASend	Valid (0s)
FRAME_LCU_DT_AnaOrder assume: A:FRAME_LCU_DT_AnaOrder	Valid (0s)
LCU_CT_MainCont assume: A:LCU_CT_MainCont receive cont from NetDataAna	Valid (0s)
Subcomponent Assumptions	Valid (0s)
This component consistent	1 Valid
LCU_DM_NetRecv consistent	1 Valid
FRAME_LCU_DT_NetDataAna consistent	1 Valid
LCU_DM_289A_Send consistent	1 Valid
LCU_DM_289A_Recv consistent	1 Valid
FRAME_LCU_DT_289ASend consistent	1 Valid
FRAME_LCU_DT_AnaOrder consistent	1 Valid
LCU_CT_MainCont consistent	1 Valid
Component composition consistent	1 Valid
Verification for LCU_DM_NetRecv	4 Valid
Verification for FRAME_LCU_DT_NetDataAna	9 Valid
Verification for LCU_DM_289A_Send	3 Valid
Verification for LCU_DM_289A_Recv	3 Valid
Verification for FRAME_LCU_DT_289ASend	4 Valid
Verification for FRAME_LCU_DT_AnaOrder	4 Valid
Verification for LCU_CT_MainCont	26 Valid

Figure 18: The compositional verification results of the rocket power-on control subsystem model

Following the same way, we give the results of the compositional verification

of the exhaust cover control subsystem. The system-level requirement of the exhaust cover control subsystem is SR1, which can be decomposed into exhaust cover switch cover processing (SR1C1) and exhaust cover switch cover control (SR1C2). SR1C2 can be divided into exhaust cover self-inspection watchdog (WDSR2) and exhaust cover self-inspection function (SSR2). The compositional reasoning result is shown in Fig.19. According to the results of compositional verification, it can be known that the compositional verification of the exhaust cover control subsystem involves 13 contracts, among which the system-level contract SR1G is satisfied.

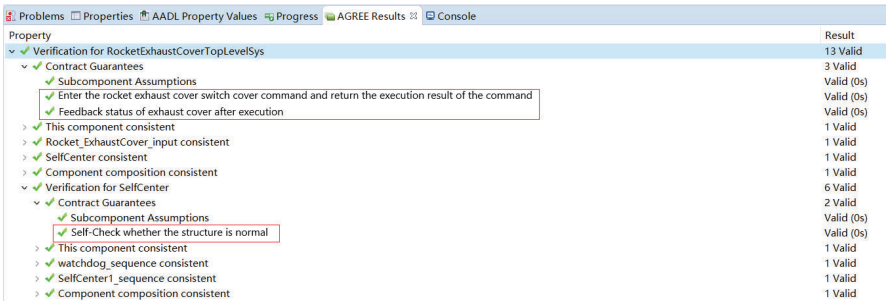


Figure 19: The compositional verification results of the exhaust cover control subsystem model

Under the cooperation with the industrial partners, we verify numbers of desired properties of the rocket launch control system. The verification results of the rocket launch control system are shown in Table 3. All data gathering was performed on a MacBookPro running MacOS Catalina 10.15.7 with an Inter Core i7-8750H CPU running at 2.2GHz and 16 GB of RAM. For each slot in the table, we ran the verification three times and recorded the mean time.

Table 3: The statistics of verification results of launch control system

	Contract	Verification time	Correctness
Exhaust cover control (SR1)	13	5s	13/13

Rocket hatch control (SR2)	10	4s	10/10
Control of rocket launch preparation/cancellation (SR3)	12	4s	12/12
Rocket power-on control (SR4)	70	29s	70/70
Control of rocket hatch unlock-/lock (SR5)	15	6s	15/15
Thermal battery activation control (SR6)	14	5s	14/14
Control of safety mechanism unlock/lock (SR7)	14	5s	14/14
Rocket ignition control (SR8)	10	4s	10/10
Rocket power-off control (SR9)	7	2s	7/7
Launch control system	165	64s	165/165

- Individual component verification with UPPAAL

This section takes the control function (LCU_CT_ContOrd) as an example to illustrate the verification of AADL leaf components based on UPPAAL.

As shown in Fig.20, the left side is the AADL model, and the corresponding UPPAAL model is given on the right side. We create the initial function (start()) of the time automata according to the *Assume* in the AADL contract, and write the verification formula of UPPAAL according to the *Guarantee* in Fig.10. The verification results are shown in Table 4.

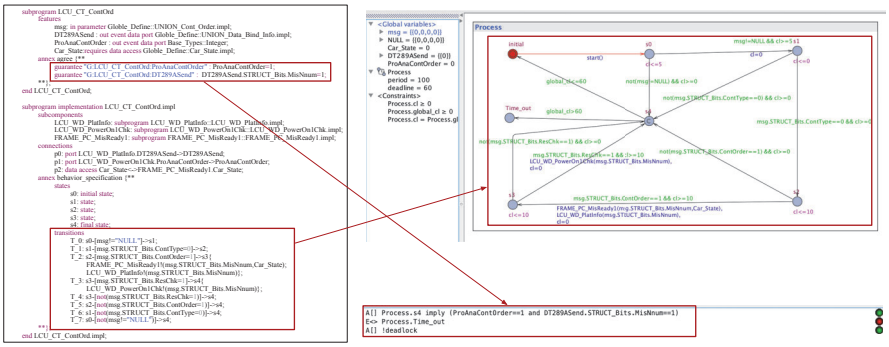


Figure 20: Verification of leaf component based on UPPAAL

Table 4: The verification results of the LCU_CT_ContOrd function

TCTL	Description	T/F
$A[]!deadlock$	No deadlock	T
$E <> Process.Time\_out$	Reachability, automata will eventually go to the Time_out state	F
A[] Process.s4 imply (ProAnaContOrder==1 and DT289ASend.STRUCT_Bits.MisNnum==1)	Functional correctness, after the LCU_CT_ContOrd is executed, two data are sent through the port	T

6.2.3. Analysis

- Research Questions

The major objective of this subsection is to evaluate the eectiveness of our approach. This objective is decomposed into the following research questions:

RQ1: Can we use our approach to obtain AADL models for dierent case studies from the aerospace domain?

Rationale: As a model generation approach, `C2AADL_Reverse` should be able to generate various kinds of legacy systems that implemented in C language. To answer the question, we apply our approach to several case studies from the aerospace domain. Table 5 provides the name, the short description, and the number of AADL models from three industry cases. For the description of Case-A, see Section 6.1. Case-B is is a core system supporting the orbiting operations of spacecraft, which undertakes the tasks of determining and controlling spacecraft attitude and orbit. For Case-C, the radar information processing subsystem refers to a system that uses modulated wave forms and directional antennas to transmit electromagnetic waves to a specic airspace in space, and extract guidance information from the received echoes. Moreover, since C and AADL are broadly used, the approach proposed in this paper can be also applied into other safety-critical domains such as automotive system.

Table 5: Application in different case studies

Case	Description	Number of Requirements
Case-A	Rocket Launch Control Subsystem (RLCS)	17600+
Case-B	Autonomous Guidance, Navigation and Control (AGNC)	13400+
Case-C	Radar Information Processing	11700+

RQ2: Can our approach generates high quality models when compared with other approaches?

This RQ checks how much information our approach can model. To answer the question, we compare the `C2AADL_reverse` tool with several existing tool-s: MoDisco[7], fREX[30], RE-CMS[31], Src2MoF[9], srcYUML [32], and Wang

[13]. In Table 6, we delimit six parameters to compare the most renowned researches. In 1) **Source Artefact** and 2) **Target Model**, as we seen, most of the existing works of MDRE mainly consider general domains, thus several tools support the reverse engineering from object-oriented source code into UML. This paper focuses on the domain of complex embedded systems, especially the safety-critical systems. 3) **Structural** indicates whether the MDRE methods can generate structural models. 4) **Behavioral** indicates whether the MDRE methods can generate behavioral models. fREX and Src2MoF mainly consider UML activity diagram, while we consider AADL Behavior annex. 5) **Runtime** indicates whether the MDRE methods can generate execution models, which are important for the embedded software systems. 6) **V&V** indicates validation and verification.

Table 6: Comparisons with a part of MDRE tools

Parameters / Tools	Source Artefact	Target Model	Structural	Behavioral	Runtime	V&V
MoDisco	Java,JSP	UML	Y	N	N	N
fREX	Java	UML	N	Y	N	N
RE-CMS	PHP	AST of PHP	Y	N	N	N
Src2MoF	Java	UML	Y	Y	N	N
srcYUML	C++	UML	Y	N	N	N
Wang [13]	C	AADL	Y	N	N	N
C2AADL_reverse	C	AADL	Y	Y	Y	Y

7. Related Work

7.1. Model-driven reverse engineering

We distinguish the exiting work on MDRE into two main families: specic vs. general purpose solutions.

7.1.1. General Frameworks of MDRE

Fleurey et al. [33] propose a semi-automatic round-trip model-driven migration and modernization process for the migration of large industrial software. This migration solution includes the automatic analysis of the source code, the generation of abstract models into target platform models, and generation of code for the target system.

The approach proposed by Favre et al. [34] is placed within the context of obtaining models from object-oriented code and formal techniques at the metamodel level to maintain consistency in reverse engineering process, according to the MDA standard. This approach exploits static and dynamic analysis to generate PSMs and PIMs from code and to analyze the consistence of the performed transformations from code to models and between models.

A semi-automatic approach for MDRE to extract business rules from Java applications is presented in [35]. The authors address issues concerning the extraction of knowledge from software artifacts and the representation of the extracted knowledge into the KDM metamodel with the objective to abstract the business logic implemented in a system.

MoDisco [36] is an extensible and generic Eclipse plug-in for model driven reverse engineering proposed and implemented by Bruneliere et al., based on model discovery and model understanding. MoDisco has three layered architecture i.e. infrastructure, technologies and use case layers. It denes a basic metamodel approach for MDRE based on Knowledge Discovery Meta-model (KDM) specication to provide support for XML, JSP and Java.

7.1.2. Specific MDRE Solutions

Several past works have already described how to migrate from a particular technology to another one using dedicated components.

In the automotive industry, General Motors (GM) has been using a custom-built, domain specic modeling language, implemented as an internal proprietary metamodel, to meet the modeling needs in its control software development. Selim et al. [37] explores applying model transformations to address the challenges in migrating GM-specic, legacy models to AUTOSAR (AUTomotive Open System ARchitecture) equivalents. They have built and validated a model transformation using the MDWorkbench tool, the Atlas Transformation Language, and the Metamodel Coverage Checker tool.

In the field of IoT, Manev et al. [8] proposes a tool, called ITACG (IoT software Analysis and Code-Generation tool), for performing reverse engineering and extraction. This is accomplished by scanning the source code of the target system and extracting architectural information from it, which is stored into a UML model.

Regarding the reverse generation of AADL models, Wang et al. [13] proposes an approach for AADL models extraction from existing embedded software and reduce maintenance costs. In an effort to bridge the semantic and syntactic gaps between the two languages, they have defined a set of mapping rules from C to AADL models. However, this method does not consider behavior and runtime information, and does not validate the reverse process. In Integrated Modular Avionics (IMA) systems, Lesovoy et al. [14] proposes an approach to extract the AADL models from source code of ARINC 653-compatible application software. This approach applies the ideas of counterexample and path feasibility check to the task of extracting the architectural information from source code.

However, since safety-critical software often run on various embedded plat-forms, reverse engineering needs to deal with the information such as static structure, dynamic run-time, and functional behavior. In particular, as multi-core processors are widely used in safety-critical software, the reverse engineering

of multi-task synchronization, mutex, communication, and task scheduling has become an important problem. Moreover, when MDRE exists in the domain of safety-critical systems, validation of the MDRE process and verification of the resulted models are highly desirable because such software systems have to undergo development regulations and certification restrictions. To the best of our knowledge, this paper presents a first effort on the validation and verification of the reverse process from C to AADL.

7.2. Verification of AADL models

A number of works have been proposed respect to formal analysis of AADL models. However, they always consider a subset of AADL because that the AADL language is very rich. In [38], a methodology for translating AADL to UPPAAL has been proposed. This work is primarily focused on flow and deadlock analysis rather than behaviorial specification. Chkouri et al., in [39], proposed a method for translating AADL models into BIP, which makes it possible to make use of the BIP toolset for verification. Bodeveix et al. [40] proposed a verification tool chain for AADL models through its transformation to the Fiacre language, and prove the correctness of the transformation from AADL into Fiacre. Yang et al. [41, 42] proposed a methodology for translating AADL to TASM, through which the properties of AADL can be analyzed by the toolset developed for TASM. Yu et al. [43] proposed to apply the POLYCHRONY toolset, based on the synchronous language SIGNAL, for timing modeling, analysis and validation of AADL. Hugues et al. [44, 45] proposed the formal verification of real-time systems modelled with the AADL language and its behaviour annex, and defines a formal semantics of an AADL behavioural subset using the LNT (Lotos NT) language.

Compositional verification has attracted significant research attention because of its viability as a scalable technique for reasoning about complex sys-tems. Backes et al. [25] applied a compositional verification approach to a realistic avionics system and demonstrated the effectiveness of the AGREE tool in performing this analysis. Murugesan et al. [46] assembled proofs of system

level properties by using the Simulink Design Verifier to establish component-level properties and AGREE to perform the compositional verification of the architecture, and verifies a realistic medical cyber-physical system. Cofer et al. [18] described the compositional reasoning framework for proving the correctness of a system design, and provide a proof of the soundness of their compositional reasoning approach. An aircraft flight control system is provided to illustrate the method. Posse [16] proposed an AADL annex sub-language for annotating components with assume/guarantee contracts and a prototype verier that performs compositional analysis. They also provide a prototype plug-in for OSATE supporting an annex language which is called AGCL.

Compared with existing works, this paper focuses on the verification of the resulted models of the reverse process from C to AADL. Because of the broadly use of UPPAAL in the verification of functional behaviors of components, we assemble verification of system-level properties by using UPPAAL to establish component-level properties and the AGREE environment to perform the compositional verification of the architecture. In [47], the authors consider a hierarchical multi-formalism proofs of cyber-physical systems by using AGREE and UPPAAL. However, they focus on multiple abstraction layers. AGREE is used in the high-level analysis, while UPPAAL is applied in the low-level one. Therefore they propose a translation from AGREE contracts to timed automata. In this paper, we combine AGREE with UPPAAL in the same abstraction layer, in which the model checker UPPAAL establishes the behaviors of leaf components and the AGREE environment performs the compositional verification of the architecture.

8. Conclusion and Future Work

This paper has presented a model-driven reverse engineering approach for safety-critical software development and verification, namely `C2AADL_Reverse`. Compared with the existing works, `C2AADL_Reverse` considers more reversed construction including AADL component structure, behavior, and multi-threaded

run-time information. Moreover, when MDRE exists in the domain of safety-critical systems, validation of the MDRE process and verification of the resulted models are highly desirable because such software systems have to undergo development regulations and certification restrictions. We use reverse reverse engineering to validate the reverse engineering process, and verify the generated AADL models by using the model checker UPPAAL to establish component-level properties and the AGREE environment to perform the compositional verication of the architecture. To the best of our knowledge, this paper presents a first effort on the validation and verification of the reverse process from C to AADL. Finally, the effectiveness of `C2AADL_Reverse` is demonstrated using a real-world aerospace case study.

We will further carry out the following future work:

- The source code cannot explicitly express non-functional properties of the software system (such as period, execution time, resource consumption, and so on). At present, we apply third-party dynamic tools (such as WCET analysis tools) to measure timing properties and add them to the corresponding AADL model.

- Inspired by the restricted natural language approach proposed in our previous work [48], the automatic transformation from natural language requirements into AGREE contracts is currently being developed. As well, the translation from AGREE contracts to TCTL properties in UPPAAL will be also automated.

- We are considering the extension of AGREE to support for modeling components that execute asynchronously (or quasi-synchronously), and formalizing the reasoning rules in the theorem prover Coq [49].

Acknowledgements

This work is supported by National Natural Science Foundation of China (62072233) and Aviation Science Fund of China (201919052002).

References

[1] N. G. Leveson, Engineering a safer world: Systems thinking applied to safety, The MIT Press, 2016.

[2] M. D. George Romanski, Reverse engineering for software and digital systems, Tech. rep. (2016).

[3] A. van Deursen, E. Burd, Software reverse engineering, Journal of Systems and Software 77 (3) (2005) 209 – 211, software reverse engineering.

[4] S. Rugaber, K. Stirewalt, Model-driven reverse engineering, IEEE software 21 (4) (2004) 45–53.

[5] C. Raibulet, F. A. Fontana, M. Zanoni, Model-driven reverse engineering approaches: A systematic literature review, IEEE Access 5 (2017) 14516–14542.

[6] H. Bruneliere, Generic model-based approaches for software reverse engineering and comprehension, Ph.D. thesis, Nantes (2018).

[7] H. Bruneliere, J. Cabot, G. Dupé, F. Madiot, Modisco: A model driven reverse engineering framework, Information and Software Technology 56 (8) (2014) 1012–1032.

[8] D. Manev, A. Dimov, Facilitation of IoT software maintenance via code analysis and generation, in: 2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech), IEEE, 2017, pp. 1–6.

[9] U. Sabir, F. Azam, S. U. Haq, M. W. Anwar, W. H. Butt, A. Amjad, A model driven reverse engineering framework for generating high level UML models from java source code, IEEE Access 7 (2019) 158931–158950.

[10] SAE, Architecture Analysis & Design Language (AADL), AS5506C, 2017.

[11] J. Hugues, B. Zalila, L. Pautet, F. Kordon, From the prototype to the final embedded system using the ocarina aadl tool suite, ACM Transactions on Embedded Computing Systems (TECS) 7 (4) (2008) 1–25.

[12] S. Rahmoun, E. Borde, L. Pautet, Multi-objectives refinement of AADL models for the synthesis embedded systems (mu-RAMSES), in: 2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, 2015, pp. 21–30.

[13] G. Wang, X. Zhou, Y. Dong, H. Zhao, Studying on AADL-based architecture abstraction of embedded software, in: 2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing, IEEE, 2009, pp. 14–19.

[14] S. L. Lesovoy, Extracting architectural information from source code of ARINC 653-compatible application software using CEGAR-based approach, Trudy ISP RAN/Proc 30 (3).

[15] S. M. Salman, A. V. Papadopoulos, S. Mubeen, T. Nolte, A systematic methodology to migrate complex real-time software systems to multi-core platforms, Journal of Systems Architecture 117 (2021) 102087.

[16] E. Posse, J. Dingel, Contract-based specification and analysis of aadl models, in: ACVI 2014–Architecture Centric Virtual Integration Workshop Proceedings, Citeseer, 2014, p. 4.

[17] S. Bensalem, M. Bozga, J. Sifakis, T.-H. Nguyen, Compositional verification for component-based systems and application, in: International Symposium on Automated Technology for Verification and Analysis, Springer, 2008, pp. 64–79.

[18] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, L. Sha, Compositional verification of architectural models, in: NASA Formal Methods Symposium, Springer, 2012, pp. 126–140.

[19] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. Heimdahl, L. Wagner, Proof-based coverage metrics for formal verification, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 194–199.

[20] A. Gacek, J. Backes, M. Whalen, L. Wagner, E. Ghassabani, The JKind model checker, in: International Conference on Computer Aided Verification, Springer, 2018, pp. 20–27.

[21] SAE, Architecture Analysis and Design Language (AADL) Annex D: Behavior Model Annex, 2017.

[22] OSATE: Plug-ins for front-end processing of AADL models, Tech. rep., The Software Engineering Institute (2013).

[23] R. B. Franca, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, D. Thomas, The AADL behaviour annex–experiments and roadmap, in: 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), IEEE, 2007, pp. 377–382.

[24] I. ljivo, G. J. Uriagereka, S. Puri, B. Gallina, Guiding assurance of architectural design patterns for critical applications, Journal of Systems Architecture 110 (2020) 101765.

[25] J. Backes, D. Cofer, S. Miller, M. W. Whalen, Requirements analysis of a quad-redundant flight control system, in: NASA Formal Methods Symposium, Springer, 2015, pp. 82–96.

[26] T. Instrument, TI SYS/BIOS v6. 33 real-time operating system users guide, Tech. rep., SPRUEX3K (2011).

[27] G. Behrmann, A. David, K. G. Larsen, A tutorial on uppaal, Formal methods for the design of real-time systems (2004) 200–236.

[28] H. Mkaouar, B. Zalila, J. Hugues, M. Jmaiel, An ocarina extension for AADL formal semantics generation, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, 2018, pp. 1402–1409.

[29] J. Delange, L. Lec, POK, an ARINC653-compliant operating system released under the BSD license, in: 13th Real-Time Linux Workshop, Vol. 10, 2011, pp. 181–192.

[30] A. Bergmayr, H. Bruneliere, J. Cabot, J. García, T. Mayerhofer, M. Wimmer, fREX: fUML-based reverse engineering of executable behavior for software dynamic analysis, in: 2016 IEEE/ACM 8th International Workshop on Modeling in Software Engineering (MiSE), IEEE, 2016, pp. 20–26.

[31] F. Trias, V. de Castro, M. López-Sanz, E. Marcos, RE-CMS: a reverse engineering toolkit for the migration to CMS-based web applications, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 810–812.

[32] M. J. Decker, K. Swartz, M. L. Collard, J. I. Maletic, A tool for efficiently reverse engineering accurate uml class diagrams, in: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2016, pp. 607–609.

[33] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, J.-M. Jézéquel, Model-driven engineering for software migration in a large industrial context, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2007, pp. 482–497.

[34] L. Favre, Formalizing mda-based reverse engineering processes, in: 2008 sixth international conference on software engineering research, management and applications, IEEE, 2008, pp. 153–160.

[35] K. Normantas, O. Vasilecas, Extracting business rules from existing enterprise software system, in: International Conference on Information and Software Technologies, Springer, 2012, pp. 482–496.

[36] H. Bruneliere, J. Cabot, F. Jouault, F. Madiot, Modisco: a generic and extensible framework for model driven reverse engineering, in: Proceedings of the IEEE/ACM international conference on Automated software engineering, 2010, pp. 173–174.

[37] G. M. Selim, S. Wang, J. R. Cordy, J. Dingel, Model transformations for

migrating legacy deployment models in the automotive industry, Software & Systems Modeling 14 (1) (2015) 365–381.

[38] A. Johnsen, K. Lundqvist, P. Pettersson, O. Jaradat, Automated verification of AADL-specifications using UPPAAL, in: 2012 IEEE 14th International Symposium on High-Assurance Systems Engineering, IEEE, 2012, pp. 130–138.

[39] M. Y. Chkouri, A. Robert, M. Bozga, J. Sifakis, Translating AADL into BIP - application to the verification of real-time systems, in: M. R. V. Chaudron (Ed.), Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers, Vol. 5421 of Lecture Notes in Computer Science, Springer, 2008, pp. 5–19.

[40] J.-P. Bodeveix, M. Filali, M. Garnacho, R. Spadotti, Z. Yang, Towards a verified transformation from AADL to the formal component-based language FIACRE, Science of Computer Programming 106 (2015) 30–53.

[41] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, J.-P. Talpin, From AADL to timed abstract state machines: A verified model transformation, Journal of Systems and Software 93 (2014) 42–68.

[42] K. Hu, T. Zhang, Z. Yang, W.-T. Tsai, Exploring AADL verification tool through model transformation, Journal of Systems Architecture 61 (3-4) (2015) 141–156.

[43] H. Yu, Y. Ma, T. Gautier, L. Besnard, P. Le Guernic, J.-P. Talpin, Polychronous modeling, analysis, verification and simulation for timed software architectures, Journal of Systems Architecture 59 (10) (2013) 1157–1170.

[44] H. Mkaouar, B. Zalila, J. Hugues, M. Jmaiel, Towards a formal specification for an AADL behavioural subset using the LNT language, International Journal of Business and Systems Research 14 (2) (2020) 162–190.

[45] H. Mkaouar, B. Zalila, J. Hugues, M. Jmaiel, A formal approach to AADL model-based software engineering, International Journal on Software Tools for Technology Transfer 22 (2) (2020) 219–247.

[46] A. Murugesan, M. W. Whalen, S. Rayadurgam, M. P. Heimdahl, Compositional verification of a medical device system, in: Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, 2013, pp. 51–64.

[47] M. W. Whalen, S. Rayadurgam, E. Ghassabani, A. Murugesan, O. Sokolsky, M. P. Heimdahl, I. Lee, Hierarchical multi-formalism proofs of cyber-physical systems, in: 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), IEEE, 2015, pp. 90–95.

[48] F. Wang, Z. Yang, Z. Huang, C. Liu, Y. Zhou, J. Bodeveix, M. Filali, An approach to generate the traceability between restricted natural language requirements and AADL models, IEEE Trans. Reliab. 69 (1) (2020) 154–173.

[49] A. Chlipala, Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant, The MIT Press, 2013.