An Approach to Generate the Traceability Between Restricted Natural Language Requirements and AADL Models

Fei Wang, Zhi-Bin Yang, Zhi-Qiu Huang, Cheng-Wei Liu, Yong Zhou, Jean-Paul Bodeveix, and Mamoun Filali

***Abstract*—Requirements traceability is broadly recognized as a critical element of any rigorous software development process, especially for building safety-critical software (SCS) systems. Model-driven development (MDD) is increasingly used to develop SCS in many domains, such as automotive and aerospace. MDD provides new opportunities for establishing traceability links through modeling and model transformations. Architecture Analysis and Design Language (AADL) is a standardized architecture description language for embedded systems, which is widely used in avionics and aerospace industries to model safety-critical applications. However, there is a big challenge to automatically establish the traceability links between requirements and AADL models in the context of MDD, because requirements are mostly written as free natural language texts, which are often ambiguous and difficult to be processed automatically. To bridge the gap between natural language requirements (NLRs) and AADL models, we propose an approach to generate the traceability links between NLRs and AADL models. First, we propose a requirement modeling method based on the restricted natural language, which is named as RM-RNL. The RM-RNL can eliminate the ambiguity of NLRs and barely change engineers' habits of requirement specification. Second, we present a method to automatically generate the initial AADL models from the RM-RNLs and to automatically establish traceability links between the elements of the RM-RNL and the generated AADL models. Third, we refine the initial AADL models through patterns to achieve the change of requirements and traceability links. Finally, we demonstrate the effectiveness of our approach with industrial case studies and evaluation experiments.**

Manuscript received June 1, 2018; revised October 20, 2018, May 6, 2019, and July 19, 2019; accepted August 10, 2019. This work was supported in part by the National Natural Science Foundation of China under Grant 61502231 and Grant 61772270, in part by the National Defense Basic Scientific Research Project under Grant JCKY2016203B011, in part by the National Key Research and Development Program under Grant 2018YFB1003902 and Grant 2016YFB1000802, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20150753, in part by the Avionics Science Foundation of China under Grant 2015ZC52027, and in part by the Fundamental Research Funds for the Central Universities under Grant NP2017205. Associate Editor: W. Eric Wong. *(Corresponding authors: Zhi-Bin Yang; Zhi-Qiu Huang.)*

F. Wang, Z.-B. Yang, Z.-Q. Huang, C.-W. Liu, and Y. Zhou are with the Key Laboratory of Safety-Critical Software (Ministry of Industry and Information Technology), College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: wangfei2014@nuaa.edu.cn; yangzhibin168@163.com; zqhuang@nuaa.edu.cn; lcwj3nuaa@nuaa.edu.cn; zhouyong @nuaa.edu.cn).

J.-P. Bodeveix and M. Filali are with the Institut de Recherche en Informatique de Toulouse, Centre National de la Recherche Scientifique, l'Université Paul Sabatier, Université de Toulouse, Toulouse F-31062, France (e-mail: bodeveix@irit.fr; filali@irit.fr).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2936072

***Index Terms*—Architecture Analysis and Design Language (AADL), model-driven development (MDD), refinement, requirement traceability, restricted natural language requirements (NLRs), safety-critical software (SCS).**

I. Introduction

SOFTWARE has been widely used in safety-critical systems, and it is increasing in size and complexity. Software safety has become critical to the development of such systems, since the consequence of a failure in such software may be serious [1]. Therefore, developing safety-critical software (SCS) imposes special demands for ensuring the quality of the developed software.

Requirement traceability is defined as "the ability to follow the life of a requirement in both a backward and forward direction" [2], and it is a critical element of any rigorous software development process [3], especially for building SCS systems. For example, the Federal Aviation Administration has established DO-178C [4] as the accepted means for certifying all new aviation software, and this standard specifies that at each stage of development software, developers must be able to demonstrate traceability of designs against requirements. Similarly, the U.S. Food and Drug Administration states that traceability analysis must be used to verify that a software design implements all of its specified software requirements, that all aspects of the design are traceable to software requirements, and that all code is linked to established specifications and established test procedures [5].

Model-driven development (MDD) is increasingly used to develop SCS in many domains, such as automotive and aerospace [6]. To improve the quality and control development costs of systems, MDD advocates the use of formally defined models as first-class citizens during software development instead of using models just as informal mediums for describing software systems or to facilitate interteam communication [7]. Existing MDD languages and approaches have covered various modeling demands, such as Unified Modeling Language (UML) [8] for general modeling, SysML [9] for system modeling, Simulink [10] for control system modeling, and Architecture Analysis and Design Language (AADL) [11] for the architectural modeling of embedded systems.

AADL is an architecture description language standardized by the Society of Automotive Engineers (SAE) in 2004 and published as SAE 5506 standard. It is a de facto standard in the

domain of avionics and automotive software systems. AADL is specifically designed for the specification, analysis, code generation, and automated integration of real-time critical systems (real time, embedded, fault tolerant, secure, safety critical, software intensive, and so on) [12]. Moreover, it provides a vehicle to allow analysis of system (and system of systems) designs prior to development and supports a model-based development approach throughout the system life cycle [13].

MDD also provides new opportunities for establishing traceability links through modeling and model transformations [14]. With the support of MDD strategies, transformation-based techniques can generate traceability links along with the generation of artifacts (e.g., design models), which may be represented in different languages and at different levels of abstractions [15].

However, it is difficult to establish the traceability links between requirements and design models in the context of MDD. There is a substantial inherent gap between requirement descriptions and designs, because the transformation from requirements to design models is not included in the model-driven architecture life cycle, which starts from an analysis model (or design model) and ends with deployed code [15]. The reason of this exclusion is perhaps that requirements are always written with free natural language texts, which are not a model formal enough to be understood by computers. As a result, free natural language requirements (NLRs) are not suitable for automated transformations.

Therefore, we leverage MDD techniques into the automatic generation of traceability links between NLRs and AADL models.

A. Research Questions

Traceability links provide a critical support for numerous software engineering activities, including safety analysis, compliance verification, and impact prediction [16]. Therefore, traceability is a critical element of almost all SCS development processes. Although the traceability concept seems very promising to be a significant value gain in a project, it is still not very widely spread in the development practice except for projects under certain circumstances. Despite the needed efforts, the perceived benefits for developers are often low because the quality of captured traceability information is often low or its validity and correctness is hard to ensure [17].

In this article, we apply MDD techniques into automatic generation of traceability links between NLRs and AADL models. In order to achieve this goal, we derive the following research questions.

1) *RQ-1:* How to precisely describe the requirements of SCS and barely change engineers' habits of requirement specification?
2) *RQ-2:* How to automatically establish traceability links between NLRs and AADL models?
3) *RQ-3:* How to maintain the traceability links when the requirements change or AADL models are refined?

B. Contribution

Our overall objective is to devise a methodology and its tool to automatically establish traceability links between NLRs and design models. The contributions of this article are given as follows.

1) We propose a requirements modeling method based on restricted natural language (RM-RNL), which was extended from [18]. The RM-RNL can eliminate the ambiguity of NLRs and barely change engineers' habits of requirement specification.
2) In the context of MDD, we propose a method to automatically generate the initial AADL models from the RM-RNLs and to automatically establish traceability links between the elements of the RM-RNL and the generated AADL models.
3) We refine the initial AADL models through patterns to achieve the change of requirements and traceability links.
4) We demonstrate the effectiveness of our approach with industrial case studies and evaluation experiments.

C. Organization

The rest of this article is organized as follows. Section II presents the background of this article. In Section III, we describe the traceability scenarios and present the formal description of traceability links in each traceability scenario. In Section IV, we introduce the RM-RNL briefly and then describe the automatic generation method of requirement traceability links based on model transformations. The maintenance of requirement traceability during the AADL refinement process is described in Section V. In Section VI, we illustrate the validation of the approach with two industry case studies. In Section VII, we evaluate our approach and discuss potential threats to the validity of the discovered results. In Section VIII, we discuss the work related to our study. Section IX concludes this article.

II. Background

A. MACAerospace Project

MACAerospace is a research project about modeling, analysis, and code generation for aerospace software in China. The aim of the MACAerospace project is to reduce the barriers of entering model-based systems engineering (MBSE) for engineers and implement an MBSE tool that fits the actual demand of Chinese industries.

AADL is a de facto standard in the domain of avionics and automotive software systems. It is a unifying framework for model-based software systems engineering to capture the static modular software architecture, the runtime architecture in terms of communicating tasks, the computer platform architecture on which the software is deployed, and any physical system or environment with which the system interacts [11]. Thus, AADL provides a rigorous and extensible foundation for the application of MDD for embedded systems to allow analysis of system (and system of systems) designs prior to development and supports an MDD approach throughout the life cycle of embedded systems [19].

Moreover, AADL allows introducing both property sets and annex sublanguages as extensions. The first is to extend property sets for information, which cannot be described by predefined

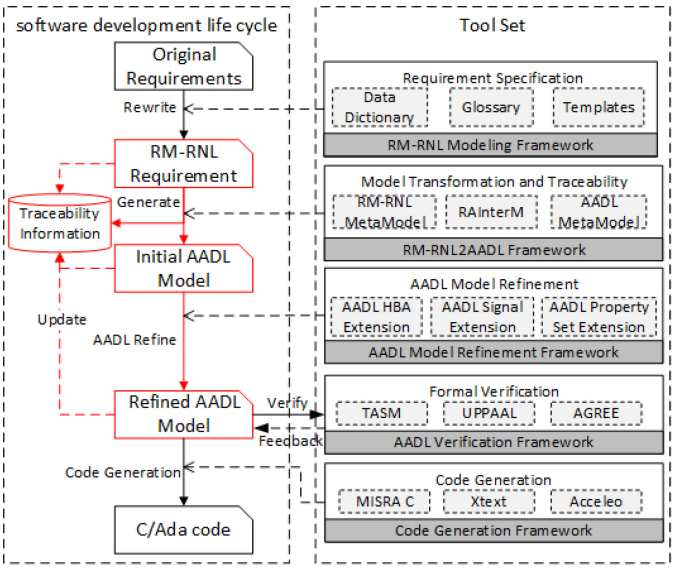

Fig. 1. Global view of the MACAerospace project.

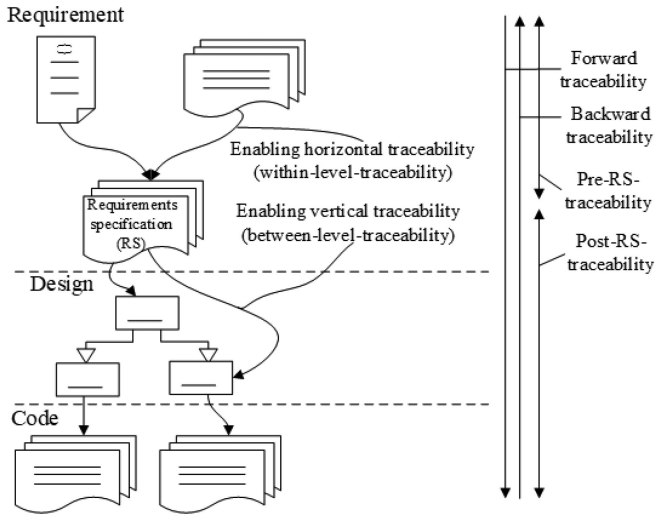

Fig. 2. Dimensions and directions of traceability links.

property sets. The second is to extend AADL annex, which is a sublanguage concerning on specific aspects, and their own semantics should be consistent to AADL core semantics. Existing AADL annexes include Data-Model annex [20], which describes the modeling of specific data constraint with AADL, Error Model Annex V2 [21], which specifies fault and propagation concerns, ARINC653 Annex [22], which defines modeling patterns for modeling avionics system, and Behavior Annex (BA) [23], which describes component behaviors with state machines, and so on.

The global view of the MACAerospace project is given in Fig. 1. MACAerospace is an integrated toolset for modeling, analysis, verification, and code generation based on AADL. Its functions are presented as follows:

1) a requirement modeling method based on restricted Chinese natural language for Aerospace Software. This requirement modeling method can be extended to other safety-critical domains, such as railway and automotive;
2) automatically generate the AADL models from the RM-RNL through model transformations;
3) refinement of the initial AADL models with graphical BA/hierarchical BA [24] and graphical synchronous language SIGNAL [25];
4) provide an integrated verification environment for AADL models using existing verification and analysis tools, such as Timed Abstract State Machine (TASM) [26], UPPAAL [27], and so on. MACAerospace can perform compositional verification based on AGREE [28] at system level. The verification results are feedback to the AADL models;
5) automatically generate C/Ada code from the AADL models;
6) verify the semantic consistency between AADL models and C/Ada code;
7) reverse engineering from C/Ada code to AADL models;
8) traceability and documents generation.

This article mainly focuses on the traceability between NLRs and AADL models. Concretely speaking, as shown in the red part of Fig. 1, the main research contents of this article include automatically generating the requirement traceability links based on model transformations and maintaining traceability links when the requirements change or AADL models are refined.

B. Requirement Traceability

In requirement engineering domain, the term traceability is usually used for the ability to follow the traces to and from requirements. One common definition of requirement traceability is given by Pinheiro [29] as "the ability to define, capture, and follow the traces left by requirements on other elements of the software development environment and the traces left by those elements on requirements." Another definition of requirement traceability is offered by Gotel and Finkelstein [2] as "the ability to describe and follow the life of a requirement, in both forward and backward directions (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)."

Usually, these definitions are also implicitly (e.g., in [30]) or explicitly [31] extended to general traceability of all artifacts as the ability to define, describe, capture, and follow traces from and to artifacts throughout the whole software development process, which seems sensible because all artifacts of a software development are (or at least should be) driven by requirements.

Tracing can be considered for various purposes and so is performed based on different foundations, such as based on logical interrelations among artifacts or based on temporal dependence between artifacts. The most common types of traceability, in the requirement traceability literature, are *forward* and *backward* traceability, *horizontal* and *vertical* traceability, and *prerequirement specification (pre-RS)* and *postrequirement specification (post-RS)* traceability [32], which are illustrated in Fig. 2.

1) Forward Traceability and Backward Traceability [33]: Forward traceability refers to following the traceability links to the artifacts that have been derived from the artifact under consideration. Backward traceability refers to the ability to follow the traceability links from a specific artifact back to its sources from which it has been derived. The forward and backward directions pertain to the logical flow of the software and system development process. These are the fundamental and primitive types of tracing.

2) Horizontal Traceability and Vertical Traceability [34]: These terms differentiate between traceability links of artifacts belonging to the same project phase or level of abstraction, and links between artifacts belonging to different ones. Horizontal traceability is about tracing artifacts at the same level of abstraction. Vertical traceability is used to trace artefacts at different levels of abstraction to accommodate lifecycle-wide or end-to-end traceability. These two types can employ both forward and backward traceability.

3) Pre-RS Traceability and Post-RS Traceability [2]: This two types of traceability are more conceptual in nature, and these can employ each of the above tracing types in some combinations. Pre-RS traceability comprises all those traces that show the derivation of the requirements from their sources and, hence, explicates the requirements production process. Post-RS traceability comprises those traces derived from or grounded in the requirements and, hence, explicates the requirements deployment process. This two types of traceability may employ forward traceability, backward traceability, horizontal traceability, and vertical traceability.

III. Traceability Scenarios

In this section, we derive a description model from the existing body of knowledge on requirement traceability that characterizes major traceability scenarios in this article. Different intended traceability usage scenarios may require different artifacts to be related [35]. For example, the demonstration of regulatory compliance requires traceability between regulatory codes and requirements, while, in contrast, the demonstration of implementation completeness requires traceability between requirements and source code. In this article, we focus on three traceability scenarios in the development process of software:

1) *TS-1:* implementing a new requirement.
2) *TS-2:* implementing a requirement change.
3) *TS-3:* implementing a refined artifact of design model.

The first traceability scenario (TS-1) is based on the RM-RNL, which automatically generates the AADL models and requirement traceability links through model transformations. TS-2 describes the requirement changes, that is, change the elements of the RM-RNL; we should regenerate the AADL models and the traceability links. Therefore, the implementation of TS-1 and TS-2 is the same in our approach. TS-3 describes the refinement of the AADL models; we should maintain the change of the requirements and the traceability links at the same time. Therefore, we propose the concept of "refinement patterns" to refine the initial AADL models and achieve the change of the requirements and the traceability links.

Generally, the software development life cycle mainly includes three kinds of artifacts: requirement specification, design model, and source code. We denote R as a set of requirement specifications that explicitly describe the function and nonfunction constraints that should be implemented in the software system, D as a set of the artifacts in the design model that contain explicit instructions on how to build a software system in order to satisfy R, and S as a set of source codes that implement D in order to build the software system. In addition, the software development life cycle spans two different stages: the *initial development stage* and the *evolution and refinement* stage [36]. We denote CR as a set of requirement change specifications that describe how a software system is supposed to be changed to meet newly emerged, shifted, or misunderstood customers' expectations. Similarly, we denote CD as a set of design *evolution and refinement* and CS as a set of changed source code.

The usage of traceability refers to the activity of following traceability links from a source artifact to a target artifact [37] to achieve an explicit goal in a development project. The usage of traceability has two main benefits: it is essential to change impact analysis, and it helps to determine completion. Thereby, impact analysis refers to identifying the consequences of implementing a new or a changed requirement [38]. Completion analysis refers to resolving either the traceability from requirements to their implementation or vice versa and allows determining whether or not all the specifications and the implementation are complete [39]. While project managers and requirement engineers concern with the change impact on and the completeness of requirements artifacts, system engineers and developers concern with the change impact on and the completeness of source code artifacts.

In summary, analyzing change impact and determining completeness are the most common traceability use cases in practice. Therefore, in order to standardize the process of establishing traceability links (path), we give the standard form of requirement traceability links (path) in three traceability scenarios based on the requirement traceability description model shown in Fig. 3.

In the following subsections, we discuss these three traceability scenarios and refer especially to three characteristics: the source artifacts on which the analysis is applied, the traceability link paths that are followed to conduct the analysis, and the target artifacts to which traceability link paths are resolved. Additionally, we provide illustrating examples for each discussed scenario.

A. Implementing a New Requirement or Requirement Change

First, we consider the horizontal traceability in the requirements phase. In this phase, we concern with the analysis of effects that a new or a changed requirement has on its dependent requirements artifacts. Horizontal traceability relations from a new or changed requirement to dependent requirements or requirement changes are to be resolved by the stakeholders in order to identify what other requirements are potentially impacted. Thus, the source artifact is an element of R or CR; the target artifacts are dependent requirement artifacts, which are

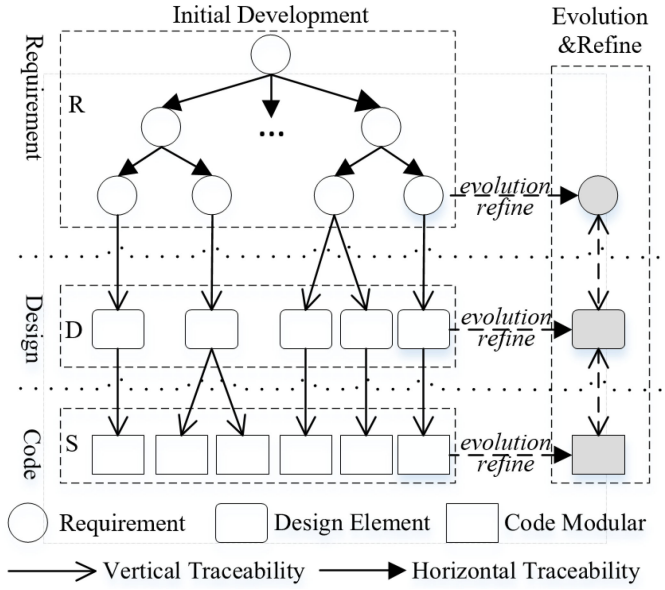

Fig. 3. Requirement traceability description model.

potentially impacted by the new or changed requirement. The traceability link paths between source and target artifacts consist of horizontal traceability links across requirement artifacts only. Thus, traceability paths can consist of any of the following traceability links in any sequence: $r \rightarrow r, r \rightarrow cr, cr \rightarrow cr, cr \rightarrow r$, where $r \in R$ and $cr \in CR$.

Next, we consider the vertical traceability in the process of requirements implementing. In this phase, we must create a set of new design and source code artifacts to implement the new requirements, and also, we can establish a traceability link path between requirements and source code. We also concern with completeness determination for new or changed requirements. The vertical traceability is resolved by the engineers to follow the implementation process subsequently from its originating requirement to its final result (source code) in order to identify whether or not all stated requirements are satisfied by source code. Thus, the source artifact is an element of *R* or *CR*, and the target artifacts are elements of *S*. The traceability link paths between source and target artifacts consist of vertical traceability links only. Each traceability link path connects a requirement with a source code artifact through zero to many intermediate design artifacts: $r[\rightarrow d]^* \rightarrow s$ or $cr[\rightarrow d]^* \rightarrow s$, where $r \in R$, $d \in D$, $s \in S$ and $cr \in CR$.

Examples: The example of implementing a new requirement is shown in Fig. 4(a). In this example, the set of source artifacts consists of the newly created requirement r_2 and the set of target artifacts consists of r_1, which is the only dependent requirement of r_2. The artifacts r_2 and r_1 are connected through the traceability link path: $r_2 \rightarrow r_1$. In addition, we create design artifact d_2 and code artifact s_2 and s_3 to fulfill the new requirement r_2. While r_2 and s_3 are connected through the traceability link path: $r_2 \rightarrow d_2 \rightarrow s_2$, the artifacts r_2 and s_3 are connected through the traceability link path: $r_2 \rightarrow d_2 \rightarrow s_3$. Similarly, the example of implementing a requirement change is shown in Fig. 4(b); we

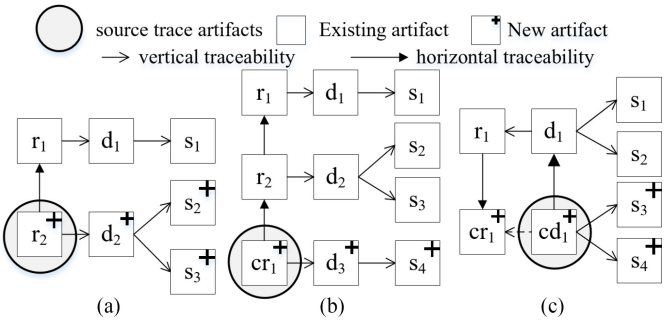

Fig. 4. Overview of the traceability links for each implementation scenario. (a) Implementation of new requirements. (b) Implementation of changed requirements. (c) Implementation of design refinement.

can create horizontal traceability link path: $cr_1 \rightarrow r_2 \rightarrow r_1$, and vertical traceability link path: $cr_1 \rightarrow d_3 \rightarrow s_4$.

B. Implementing a Refined Artifact in Design Models

First, we consider the horizontal traceability of refined artifacts in the design model. In this phase, we concern with the analysis of effects that a refined design artifact has on its dependent design artifacts. Thus, the source artifact is an element of *CD*; the target artifacts are dependent design model artifacts, which are potentially impacted by the refined design artifacts. The traceability link paths between source and target artifacts consist of horizontal traceability links across design artifacts only. Thus, traceability link paths can consist of any of the following traceability links in any sequence: $cd \rightarrow d, cd \rightarrow cd$, where $d \in D$ and $cd \in CD$.

Next, we consider the vertical traceability in the process of implementation. In this phase, we must change or create a set of requirement and source code artifacts to satisfy the refined artifacts in design model, and also, we must maintain the traceability link path from design to requirements and design to source code. In addition, we also concern with completeness determination for the refined design models. Therefore, the vertical traceability can be trace to requirement through backward traceability, and trace to source code through forward traceability. Thus, the source artifact is an element of *CD*, and the target artifacts are elements of *CR* or *CS*. Each traceability link path connects a design artifact with a requirement or source code artifact: $cd[\rightarrow d]^*[\rightarrow r]^* \rightarrow cr$, where $r \in R$, $d \in D$, $cr \in CR$ and $cd \in CD$.

Examples: The example of implementing a refined artifact in design models is shown in Fig. 4(c); the set of source artifacts consists of the refined design model artifacts cd_1 and the set of target artifacts consists of the changed requirement cr_1. While cd_1 and cr_1 are connected through the traceability link path: $cd_1 \rightarrow d_1 \rightarrow r_1 \rightarrow cr_1$.

IV. Automatically Generate the Requirement Traceability Links

In this section, we first introduce the RM-RNL; then, we describe the method of transformation from the RM-RNL to

the AADL models. Finally, we present the traceability between the RM-RNL and the AADL models.

A. RM-RNL

To promote the application of MDD in safety-critical domains and bridge the gap between NLRs and AADL models, we proposed a requirements description method based on the restricted NLRs template in [18]. However, this requirement template mainly describes the hierarchical structure of the requirements and the interface and interaction between the modules, and it lacks of the description ability for functional behaviors.

Specifically, the RM-RNL provides a method to structure requirements and restricts the way how users specify requirements. RM-RNL is a 4-tuples, i.e., RM-RNL ::=$< Glossary, Data\_Dictionary, Requirement\_Template\_Set, Restricted\_Rules >$.

1) Glossary: A Glossary describes the domain and system specified terminologies in requirement specifications, including the names of systems, modes, hardware, states, etc. Each terminology in Glossary can be defined as a 3-tuple, i.e., $Terminology ::=< Id, Name, Type >$.

2) Data_Dictionary: A Data_Dictionary describes all the data and event elements in requirement specifications, including static data (constants or parameters), dynamic data (interactive data), etc. Data has simple or complex types (like *struct* in C). Each data in Data_Dictionary can be defined as a 7-tuple, i.e., $Data ::=< Id, DataName, DataType, DataUnit, Data\ Range, Data - Accuracy, DataDescription >$.

3) Requirement_Template_Set: It is a common practice in industry to decompose requirements with hierarchical structure, which is same as AADL models. Thus, requirement specifications in RM-RNL are organized as four levels: System (SRT), sub-System (SSRT), Function (FRT), and sub-Function (SFRT), for each level we define a template. A higher level requirement template can take lower level ones as its children, and these hierarchical relations build up the structure of requirements. We also define a shared function block requirement template (SFBRT), which describes common functions used in different function blocks.

Specifically, a requirement template usually consists of an identifer (ID), name, input, output, template composition, and requirement constraint. ID represents the unique identifier of the template. Name represents the name of the template that is defined in Glossary. Input and output describe the interaction between the current module and others, and generally include two types of event and data. Template composition describes that a complex system can be decomposed into several subsystems and functions, and a subsystem can be further decomposed into several subsystems and functions, and a function can be decomposed into several subfunctions. Requirement constraint includes mode transition, functional requirement, hardware constraint, interface requirement, and performance requirement.

1) Mode transition describes modes and mode transitions of the module (Not in SFRT and SFBRT).
2) Functional requirements represent the dynamic behavior of the module. For a critical functional requirement, we can mark it as safety-critical function. According to the ICE61508 standard, the safety requirements can be divided into safety functional requirements and safety-related constraints. Therefore, a safety requirement can be expressed by one or more safety-critical functions and safety-related constraints.
3) Hardware constraints represent the hardware requirements of the software systems, such as CPU, memory, etc.
4) Interface requirements represent the requirements of the interface between each module and the environment, such as interface data transfer protocol.
5) Performance requirements represent the quantitative requirements for the ability of software to complete tasks, such as real time, power consumption, maximum processing capacity, etc.

TABLE I
GENERAL RESTRICTION RULES OF SENTENCE

No.	Description
$R1$	The subject of a sentence must be the name of modules or "this module".
$R2$	Sentence must be a declarative sentence.
$R3$	Sentence must be in the present tense.
$R4$	Do not use modal verbs, pronouns, adverbs and adjectives that express negative meanings.
$R5$	Use active voice rather than passive voice.
$R6$	Only complete sentences are allowed.
$R7$	Use simple sentences only. A simple sentence must contain only one subject and one predicate.
$R8$	Clearly describe the interaction between the modules without omitting its subject and object.
$R9$	Same words should be used to represent the same object.

4) Restricted_Rules: For each element in the template of the RM-RNL, we design a set of general restriction rules shown in Table I.

In addition, we define a set of sentence patterns. Each sentence pattern represents a complex sentence structure, which are composed of sequences of single sentences conjunct by the keywords (e.g., *IF THEN ELSE, AND, LOOP FOR, NOT*) for describing different requirements. We have designed several recommended simple sentence patterns for each kind of requirement in the template of the RM-RNL. Here, we give five predefined sentence patterns for functional requirements and example, as shown in Table II.

There are three types of simple sentence in Table II, where *Behavior* denotes a simple sentence describing a single behavior action, such as "this module sends Handshake Information to Guidance, Navigation, and Control Computer (GNCC)." *Condition* denotes the conditions (single conditions conjuncted by AND) of actions, including dispatch conditions and variable conditions such as "IF receiving Handshake Success Information." *TimeRestrain* denotes time constraint such as "function A finishes in 5 s."

Here, we give a running example of RM-RNL, i.e., the "GNCC Controlling Data Retransmission" module in the GNCC system, which is shown in Table III. GNCC Controlling Data Retransmission in Intelligent Terminal Unit is a small unit in

TABLE II
SENTENCE PATTERNS FOR FUNCTIONAL REQUIREMENTS

No.	Sentence Pattern	Explanation	Example of Sentence Pattern
$SP1$	Behavior	A single sentence presenting one action.	This module sends Handshake Information to GNCC.
$SP2$	Condition + Behavior	Execute a behavior action if the condition satisfied.	IF receiving Handshake Success Information, THEN this module sends GNCC Controlling Data to GNCC.
$SP3$	TimeRestrain + Behavior	Execute a behavior action in a time constraint.	This module sends Handshake Information to GNCC in 5ms.
$SP4$	Condition + Behavior+ Else+ Behavior	Execute a behavior action if the condition satisfied, else the other one.	Execute one behavior action if that condition satisfied, else the other one.
$SP5$	TimeRestrain + Condition + Behavior	Execute a behavior action if the condition satisfied in a period of time, that condition can only be a variable condition.	IF Attitude Deviation Angle is not bigger than 26 degree in 30ms, THEN this module sends Earth Capturing Success Information to GNCC.

"+" means sequences of elements explained below in sentences, e.g., "Condition + Behavior" presenting a sentence pattern as "IF Condition, Then Behavior."

TABLE III
MODULE OF GNCC CONTROLLING DATA RETRANSMISSION SPECIFIED BY THE RM-RNL

ID		SSRT1
Name		GNCC Controlling Data Retransmission
Input		1.Controlling Instruction of GNCC Data Retransmission
		2.GNCC Controlling Data
		3.Handshake Success Information
Output		1.GNCC Controlling Data
		2.MF Flag
		3.Handshake Information
		4.Finish Information
		5.Handshake Failure Information
Requirement Constraint	Functional Requirement	1.This module sends Handshake Information to GNCC in 10ms; (SP3)
		2.IF receiving Handshake Success Information, THEN this module sends GNCC Controlling Data to GNCC, ELSE LOOP Function 1 FOR 3 times; (SP4 with keyword LOOP FOR)
		3.IF NOT receiving Handshake Success Information, this module sends "1" to MF Flag AND this module sends Handshake Failure Information to Log; (SP2 with keyword AND)
		4.This module sends Finish Information to GNCC.(SP1)
	Performance Requirement	1.This module sends Finish Information to GNCC.(SP1)

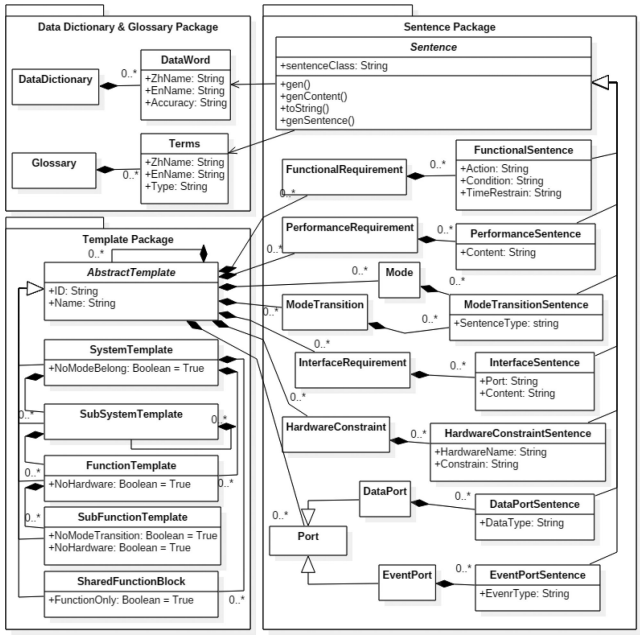

Fig. 5. Metamodel of the RM-RNL.

the data management system in GNCC, and it is in charge of data retransmission for GNCC. Input and output such as "GNCC Controlling Data" in Table III have been defined in the data dictionary. "GNCC" and "GNCC Controlling Data Retransmission" have been defined in the glossary. Sentences 1–4 in Functional Requirement satisfy sentence patterns SP3, SP4 with keyword "*LOOP FOR*," SP2 with keyword "*AND*," and SP1, respectively.

B. Automatically Generate the Requirement Traceability Links Through Model Transformations

Restricted_Rules are mainly used to restrict the expressiveness of natural language to reduce ambiguity and vagueness. The Glossary is transformed to the AADL models straightly. Therefore, we just consider the transformation of Data_Dictionary and Requirement_Template_Set to AADL models.

1) *Transformation of Data_Dictionary:* Data in data dictionary are directly transformed to AADL data components. The items of a data are transformed into *subcomponents* and *properties* of an AADL data component.
2) *Transformation of Requirements Templates:* To simplify the transformation and facilitate the possibilities for further extensions, we propose an intermediate model, which is named as RAInterM (RM-RNL2AADL Intermediate Model).

In the following sections, we present the transformations in details. We first introduce the metamodel of RM-RNL and RAInterM, and then describe the transformations from RM-RNL to RAInterM and the transformations from RAInterM to AADL respectively. Finally, we discuss the traceability between the RM-RNL and the AADL models.

1) Metamodel of the RM-RNL: The requirement specifications with the RM-RNL essentially contains three parts: glossary and data dictionary, requirement templates, and sentence patterns. Thus, we define the metamodel of the RM-RNL, which is shown in Fig. 5.

1) *Data Dictionary and Glossary:* Each data item refers to an instance of *DataWord* and each terminology refers to an instance of *Term* in the metamodel.
2) *Templates:* Are organized as requirement templates for systems, subsystems, functions, subfunctions, and shared

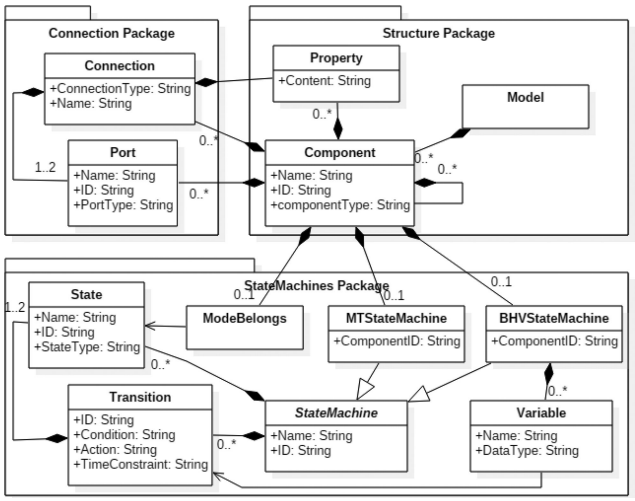

Fig. 6. Metamodel of RAInterM.

function blocks. Each template inherits from a super class *AbstractTemplate*, and a higher level template can be composed by several lower level ones.

3) *Sentence patterns:* Each type of requirements in templates is organized as a list of sentences, which satisfy predefined sentence patterns. Each sentence pattern inherits from a super class *Sentence*, and their transformation rules are defined in the *gen*() operation of *Sentence*.

2) Intermediate model (RAInterM): RAInterM is an intermediate model, which is used to bridge the gap between the RM-RNL and AADL models. As a result, the transformation is divided into two steps: the transformation from the RM-RNL to the intermediate model RAInterM and the transformation from the intermediate model to the AADL models. The reasons of introduction of the intermediate model RAInterM are given as follows.

1) *Simplifying the transformation:* The elements of the RM-RNL and the AADL models satisfy many-to-many mappings. Some mappings are complex, for instance, mappings of sentence patterns, which can be related to multiple templates in the RM-RNL. An intermediate model can separate many-to-many mappings into many-to-one and one-to-many mappings, which can decrease the complexity of the transformation.
2) *Guaranteeing the compatibility of extension:* We can suppress irrelevant features of AADL (e.g., declarations and implementations) in the transformation from the RM-RNL to RAInterM. Thus, when extending sentence patterns in the RM-RNL, we only need to consider the transformation of the extended elements instead of the entire transformation from the RM-RNL to AADL.

The metamodel of RAInterM is presented in Fig. 6. RAInterM is a component-based model, and the top concept is *Model*, which represents an entire abstract system. *Component* represents entities in *Model*, and type of Component is defined in glossary. *Statemachine* can be classified as *MTStateMachine* and *BHVStateMachine*. *MTStateMachine* refers to both mode transitions in RM-RNL and AADL, and *BHVStateMachine* refers to both Functional Requirement in RM-RNL and *BA* in AADL. Connection and Port refer to the concepts of *port* and *connection* in AADL, representing the interactions among components.

Algorithm 1: Transformation From the RM-RNL to the RAInterM Model.

Require:
RM-RNL
Ensure:
RAInterM

```
1:  for each Template t in RM-RNL.getTemplates do
2:    Component c = new Component (t.getType);
3:    for each Port p in t.getPorts do
4:      p.gen(RAInterM, c, p.PortType);
5:    end for
6:  end for
7:  for each Port p in RAInterM.getPorts do
8:    if p.NoSameNamePort then
9:      p.type = DATAACCESS;
10:     RAInterM.add(data = newComponent(DATA));
11:     RAInterM.addDataAccessConnections(p, data);
12:   else
13:     RAInterM.addConnections(p,
          p.getSameNamePort);
14:   end if
15: end for
16: for each Sentence s in t.getRequirements do
17:   s.transform(RAInterM, c);
18:   //Each type of requirements transformed into
        different parts in RAInterM
19: end for
```

3) Transformation from the RM-RNL to the RAInterM Model: To facilitate the transformation from the RM-RNL to RAInterM, we develop a transformation algorithm (see Algorithm 1). This algorithm can be divided into three parts.

1) *Transformation of the structure of RM-RNL:* The structure of RM-RNL is transformed first, including the hierarchical relationship among templates and *Input/Output*. SRT, SSRT, FRT, SFRT, and SFBRT are transformed into Components with corresponding types. *Input/Output* are transformed into *Port* in RAInterM.
2) *Generation of connection:* Requirement specifications in the RM-RNL may be incomplete. For each *Port*, we create a series of *Connections* to link it with other ports, which have the same port type and opposite direction. If no matchable port exists, we create a data component in top *Component* connecting with single ports.
3) *Transformation of the elements in templates:* The Transformation of sentence pattern is realized in "*gen*()" operation. Thus, *Functional Requirements, Performance Requirements, Mode Transitions, Interface Requirements*, and *Hardware Constraints* are mainly transformed into *BHVStateMachine, Property, MTStateMachine, Property of Port*, and *Components* with hardware types in RAInterM, respectively.

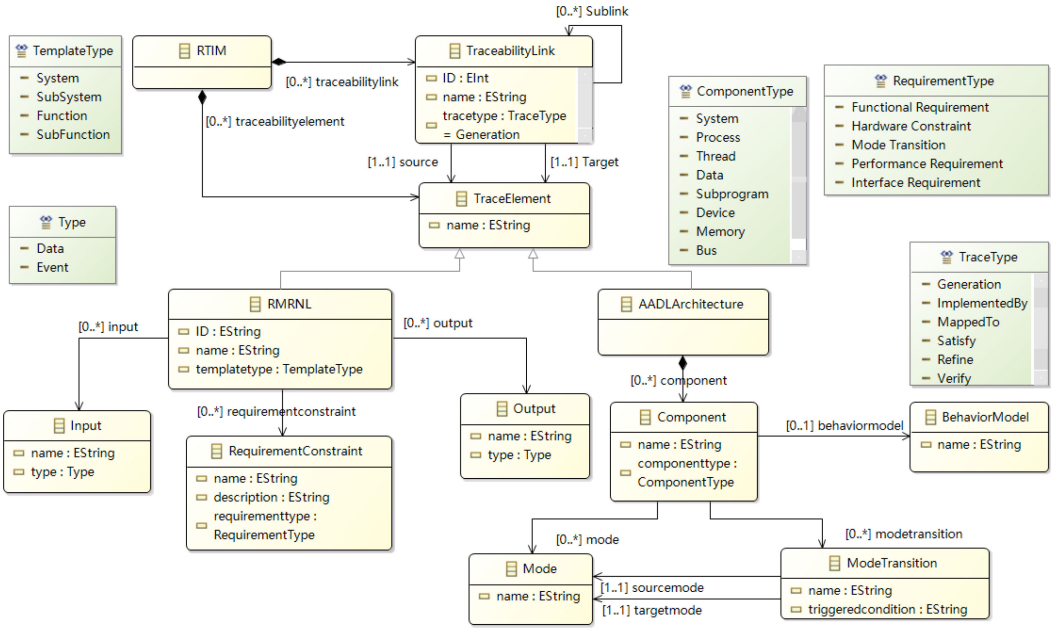

Fig. 7. Requirement traceability information model.

Algorithm 2: Transformation From the RAInterM Model to the AADL Models.

Require:
RAInterM
Ensure:
AADL models

```
for each Component c in RAInterM.getComponents do
  if c.isSystem then
    AADL.add(new System n);
  else if c.isProcess then
    AADL.add(new Process n);
  else if c.isThread then
    AADL.add(new Thread n);
  else if c.isSubprogram then
    AADL.add(new Subprogram n);
  else
    AADL.add(new Abstract n);
  end if
  AADL.addInstance(n.newInstance);
  c.getPorts → n.features;
  c.SubComponents → n.instance.subcomponents;
  c.Connections → n.instance.connections;
  c.MTStateMachine.getStates → n.instance.modes;
  c.MTStateMachine.getTransitions
    → n.instance.transitions;
  c.Properties → n.instance.properties;
  c.BHVStateMachine → n.instance.BA;
end for
```

4) Transformation from the RAInterM model to the AADL models: Since we implement the metamodel of RAInterM and the metamodel of AADL in the source code of OSATE, we develop another algorithm (see Algorithm 2) to facilitate the transformation from RAInterM to AADL.

The transformation from RAInterM to AADL can be divided as the generation of AADL-type declarations and of AADL implementation declarations; the first one includes AADL *features* such as *ports*, and the second one includes other information of AADL models, such as subcomponents, connections, properties, mode transitions, annexes, etc. AADL components are generated based on *Component* in RAInterM, and *ports, connections, MT-StateMachine, BHVStateMachine*, and *properties* in RAInterM are transformed into *features, connections, mode transitions, BA*, and *properties* of AADL components, respectively.

C. Traceability

In this section, we propose a requirement traceability information model, which can standardize the process of establishing the requirement traceability links, shown in Fig. 7. In the following paragraphs, we elaborate on the key elements of the requirement traceability model.

1) *TraceabilityLink* in Fig. 7 connects traceability artifact to define their dependence relations. One end of a traceability link is attached to a set of requirement elements, and the other end of the traceability link is attached to a set of elements of design model. In other words, traceability links build the relations between the elements of requirement and design model. Each traceability link has an attribute "id," which uniquely identifies the link. Traceability link types define the type of a link and give the semantics of the link. The sublink relation of traceability links indicates that the traceability links are not independent of each other. Some links are the sublinks of other links. In this article, the traceability relationship refers to the set of all traceability links.

2) *TraceabilityElement* refers to the elements that are supposed to be linked in order to ensure traceability. So, there are two kinds of traceability elements: the element of the RM-RNL and the element of AADL models.
3) *RTIM* is the root element, which contains traceability elements and all the generated traceability links.
4) *TraceType* enumeration is used to specify links types, which are represented as follows.
   a) *Generation* automatically relates the component to a requirement through model transformation. In this article, each traceability link type is generation by default.
   b) *ImplementedBy* relates requirement to system fragments, implementation plans, code source, etc.
   c) *MappedTo* relates requirement to a particular attribute, operation, state, or value of the artifact.
   d) *Satisfy* relates requirement to the component that fulfills it.
   e) *Refine* relates a requirement to its refined requirement.
   f) *Verify* relates requirements to test cases.

Model transformation is the process of converting a source model into a target model based on a set of transformation rules. The rules are defined with a model transformation language [40]. These transformation rules manipulate elements defined in metamodels. In our approach, the process of automatically generates the requirement traceability links actually consist of three steps.

1) Traceability links are established between the RM-RNL and the RAInterM model generated from it, while Generate_RAInterM is performed.
2) Traceability links are established between a RAInterM model and the AADL models generated from it, while Generate_AADL is performed.
3) Traceability links are established between the RM-RNL and the AADL models through merging the generated traceability links in the former two step.

V. Maintain Requirement Traceability During the Refinement of AADL Models

The initial AADL models automatically generated by model transformations cannot fully express all the properties of an embedded software system, for instance, nonfunctional properties such as schedulability, and the related feature of the execution platform. Thus, the requirement traceability links may not be complete. Therefore, we study how to realize the change of requirements in the refinements of AADL models, and how to maintain the requirement traceability links at the same time.

A. Global View the AADL-Based Development

Practically, the AADL models require iterative refinement before the system synthesis. As shown in Fig. 8, we give a global view of our AADL-based development approach. The refinement can be done at different phases, for instance, software requirement specifications, design, and coding. First, we rewrite the requirements of embedded software through RM-RNL and

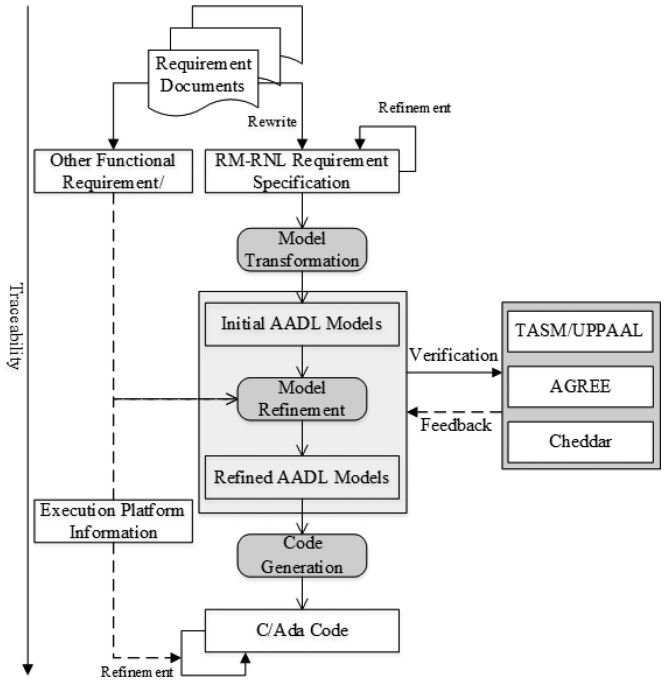

Fig. 8. Global view of the AADL-based development.

then automatically generate the initial AADL models by model transformations, as well as the requirement traceability links. It needs to maintain the consistency between requirements and AADL models by regenerating the AADL models and the traceability links when a requirement change is happened. Second, the generated initial AADL models are incomplete. Thus, it needs a refinement process, which may include several steps. In our work, we refine the AADL models from two aspects. One is to enrich the functional/nonfunctional expression of AADL models. It mainly includes adding some description of functional and nonfunctional requirements to the AADL models that cannot be specified in the RM-RNL or are difficult automatically transformed to the AADL models. The other is structure reorganization of AADL models; it mainly considers the factors of safety and reliability, or automatic code generation. Moreover, the AADL models should be formally verified, in which we use TASM and UPPAAL to verify the individual properties of each component, use the compositional verification tool AGREE to verify the system properties of the hierarchical components, and use Cheddar [41] to verify the schedulability of the system. The verification results will be further feedback to the AADL models and, thus, to guide the modification of AADL models.

Finally, the executable C and Ada code can be generated. Similarly, we also need to perform the code refinement to make the code executable, by adding some platform-specific information to C/Ada that cannot be automatically generated from the AADL models, such as the *watchdog* in the VxWorks operating system, etc.

In this article, the refinement of requirements (or requirement change) focuses on the expression ability of the RM-RNL. We maintain the consistency between requirements and the AADL models, as well as the effectiveness of traceability links through

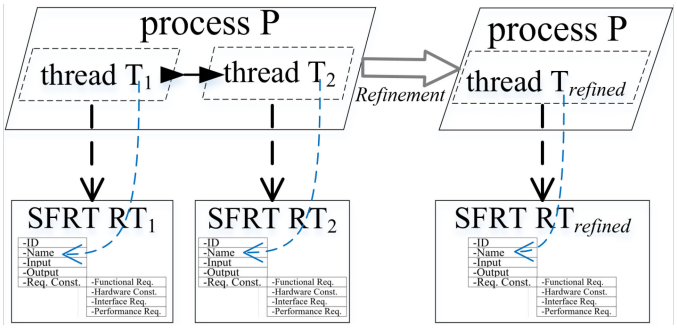

Fig. 9. Merge pattern for refinement of AADL models.

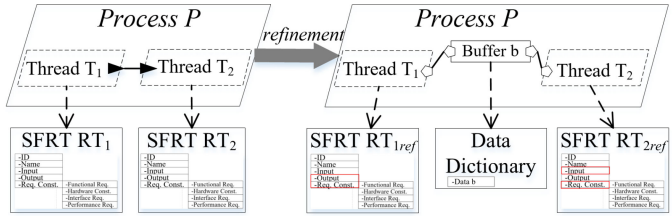

Fig. 10. Spilt pattern for refinement of AADL models.

model transformations, which will not be discussed here. In this section, we focus on the refinement of AADL models. However, it is a very complex process of generated a complete AADL models, which requires several iteration refinements.

Traceability can effectively support change impact analysis, coverage analysis, test optimization, and so on. Therefore, it is very important to establish and maintain the traceability links (path) in the refinement of AADL models, especially in the safety-critical domains.

B. Maintain Requirement Traceability Through AADL Refinement Patterns

The refinement of AADL models is an iterative design process. In this section, we describe the *refinement patterns*, i.e., general refinement scenarios, and how to maintain the requirement traceability in this refinement patterns.

First, we consider the *merge pattern*. This pattern is taking two or more components aggregated into a single one that performs the same function. The purpose of such refinement is to provide a single functionally equivalent component, from which it will be simpler to generate implementation code.

As shown in Fig. 9, before refinement, process P contains two threads T1 and T2; thread T1 processes the data received at its input port and sends the computation results to its output port. The data are then received at the input port of thread T2 through the port connection between the threads. Thread T2 then processes the data and sends the result to its output port. In addition, T1 and T2 have a traceability relationship with the subfunction requirement template SFRT1 and SFRT2 in the RM-RNL, respectively.

After refinement, the two threads are aggregated into a single one that performs the same function, and the traceability links between thread T1 (or T2) and SFRT1 (or SFRT2) are broken since thread T1 (or T2) does not exist anymore in the new AADL models. Then, we must refine the original requirements in the RM-RNL and fix the traceability links between the refined requirements and the AADL models.

Furthermore, considering the factors of reliability and safety, it is necessary to optimize the AADL models, such as increasing the redundancy of the components that realize the safety-critical functions, or increasing the shared data access mechanism to improve the reliability of data interaction, and so on. Thus, we introduce the *split pattern*. As shown in Fig. 10, we use a shared data access mechanism to refine the initial AADL model. Before refinement, the process P contains two threads T1 and T2; thread T1 processes the data received at its input port and sends the computation results to its output port. The data are then received at the input port of thread T2 through the port connection between the threads. Thread T2 then further processes the data and sends the result to its output port. After refinement, the port connection between two threads is replaced by a data subcomponent (*buffer*), which is shared by the sender and receiver threads via two data access connections. Therefore, the original port connection is split into three elements. Additionally, subprogram calls are also added to each thread to implement the refined communication mechanism (not shown in the figure).

Similarly, the original traceability links between the requirements and the initial thread are broken since a new data subcomponent is added to the process. In order to fix the traceability links and requirements, we need to add a new data item to the data dictionary in the RM-RNL to describe the information of *buffer* data components; we also need to modify the data interaction between SFRT1 and SFRT2 and the corresponding requirement constraints in the requirement template.

Before code generation, it is necessary to further refine the AADL models based on the features of target code and the execution platform. Therefore, we review additions to attach some platform-specific details or code representing an interrupt service routine. For instance, we can further to refine the AADL models by adding a property association to data, such as we can use the Allowed-Memory-Binding and Allowed-Memory-Binding Class properties to indicate the memory (or device) hardware the port resources reside on.

Finally, the target code (such as C or Ada) can be generated based on the refined AADL models.

VI. Case Studies

We illustrate our approach through two industrial case studies: Attitude and Orbit Control System (AOCS) [42], [43] and Rocket Launch Control System (RLCS), including the specification of the requirements with RM-RNL and the generation of the traceability links through the MACAerospace toolset.

A. AOCS

AOCS is a subsystem in the GNCC system on board a satellite. Its task is to ensure that the satellite attitude and orbit remain stable and follow prespecified profiles by ground control. In the case

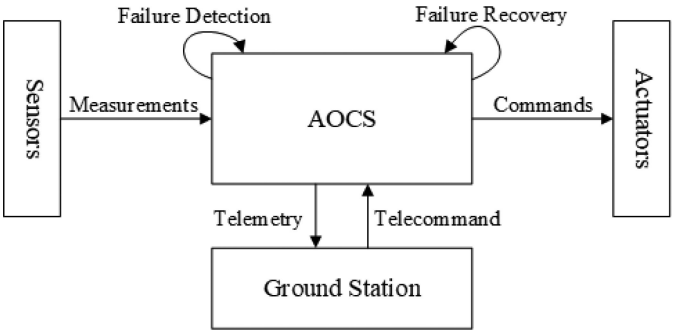

Fig. 11. Conceptual structure of AOCS.

of geostationary telecommunication satellites, for instance, the AOCS is responsible for ensuring that each particular satellite retains its position over the Earth's equator at a given longitude and keeps its antennas pointed toward the ground station.

The conceptual structure of the AOCS is shown in Fig. 11. The AOCS is a typical embedded hard real-time control system. It periodically collects the measurements from a set of sensors and converts them into commands for a set of actuators. The AOCS interacts with a ground control station, from which it receives commands (telecommands), and to which it forwards housekeeping data (telemetry).

Our requirement document of AOCS has 200 pages, and it is obtained from the industrial partner. It has nine sections (such as Attitude Determination, Orbit calculation, Attitude Control, Orbit Control, etc.) and 124 modules. We manually extract the structure of the AOCS requirements and specify them using the RM-RNL modeling function with MACAerospace. For confidentiality reasons, we are only allowed to present a sanitized portion of the AOCS case study in this article.

First, we construct a data dictionary and a glossary of AOCS based on the original textual documents. Then, we use the RM-RNL to specify the AOCS requirements. The decomposition structure of a system should be accomplished by requirement engineers, who have a wide knowledge on the object system. In this case study, the structure of decomposed systems is given in the requirement documents. The corresponding hierarchical structure described in the RM-RNL plug-in of MACAerospace is shown in Fig. 12. After the requirement specification with the RM-RNL, we can automatically generate initial AADL models and the traceability links between RM-RNL and AADL models.

The RM-RNL of AOCS includes 12 system/subsystem requirement templates, ten function requirement templates, and one subfunction template, and so on. The statistical data are shown in Table IV, and the statistical data of generated AADL models and traceability links are shown in Table V.

All types of templates except Shared Function Block and all sentence patterns except Interface-Sentence are covered by this case study. There are 19 types of sentence patterns in the RM-RNL, which include three types of Interface-Sentence. Thus, the coverage of RM-RNL template elements and sentence patterns are 80.0% and 84.2%, and we believe this case study can cover enough elements for evaluation. The hierarchical templates refer to system, process, thread components in AADL;

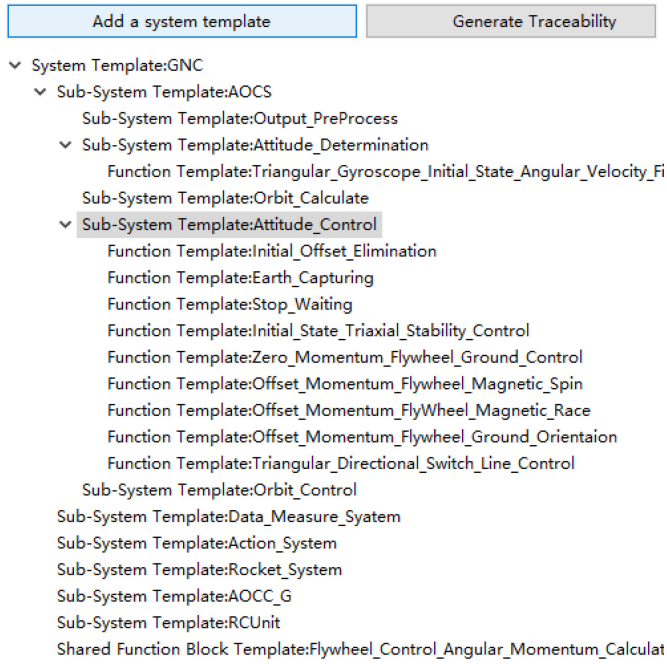

Fig. 12. Structure of AOCS requirements in the MACAerospace toolset.

TABLE IV
STATISTICAL DATA OF RM-RNL FOR AOCS CASE STUDY

Item	Number	Item	Number
Data in Data Dictionary	56	Terminologies in Glossary	62
System/sub-System Template	12	Function Template	10
Sub-Function Template	1	Data port	44
Functional sentence	6	Mode_Transition sentence	12
Performance sentence	3	Hardware_Constraint sentence	29

TABLE V
STATISTICAL DATA OF GENERATED AADL MODELS AND TRACEABILITY LINKS FOR AOCS CASE STUDY

Item	Number	Item	Number
Data component	56	System component	12
Process component	10	Thread component	1
Port	92	Connection	66
Behavior Annex	8	Mode Transition	10
Property	15	Hardware component	29
Line of AADL code	856	Trace Link	190

In/Out refers to port and condition in AADL; Mode Transition refers to mode transition in AADL; and Functional Requirement refers to BA in AADL. As we explained in Section III-A, each sentence pattern has its own transformation rules; some of them can be transformed into two or more transitions in BA, such as sentence pattern "$< Condition > + < Behavior > + < else > + < Behavior >$" is transformed into two transitions representing "if" and "else," respectively.

A part of requirement traceability links is shown in Fig. 13, and the traceability table presents a set of one-to-many mappings from the elements in the RM-RNL to the elements in the AADL

TraceInfo.traces

NO	RM-RNL Elements	Model Element
R7	Module: Pose_Control	System Pose_Control
R7-1	AOCS_init port: D3_A9	AOCS_System connections:C_14
R7-2	Pose_Control_init port: D3_A9	AOCS_init features:D3_A9
		AOCS_init connections:C_19
R7-3	Req: Pose_Control-Mode Transitions-1	Pose_Control modes: mode
R7-4	Req: Pose_Control-Mode Transitions-10	Pose_Control modes: T_8
R7-5	Req: Pose_Control-Mode Transitions-11	Pose_Control modes: T_9
R7-6	Req: Pose_Control-Mode Transitions-12	Pose_Control modes: T_10
R7-7	Req: Pose_Control-Mode Transitions-3	Pose_Control modes: T_1
R7-8	Req: Pose_Control-Mode Transitions-4	Pose_Control modes: T_2
R7-9	Req: Pose_Control-Mode Transitions-5	Pose_Control modes: T_3
R7-10	Req: Pose_Control-Mode Transitions-6	Pose_Control modes: T_4
R7-11	Req: Pose_Control-Mode Transitions-7	Pose_Control modes: T_5
R7-12	Req: Pose_Control-Mode Transitions-8	Pose_Control modes: T_6
R7-13	Req: Pose_Control-Mode Transitions-9	Pose_Control modes: T_7

Fig. 13. Part of generated requirement traceability links of the attitude control subsystem.

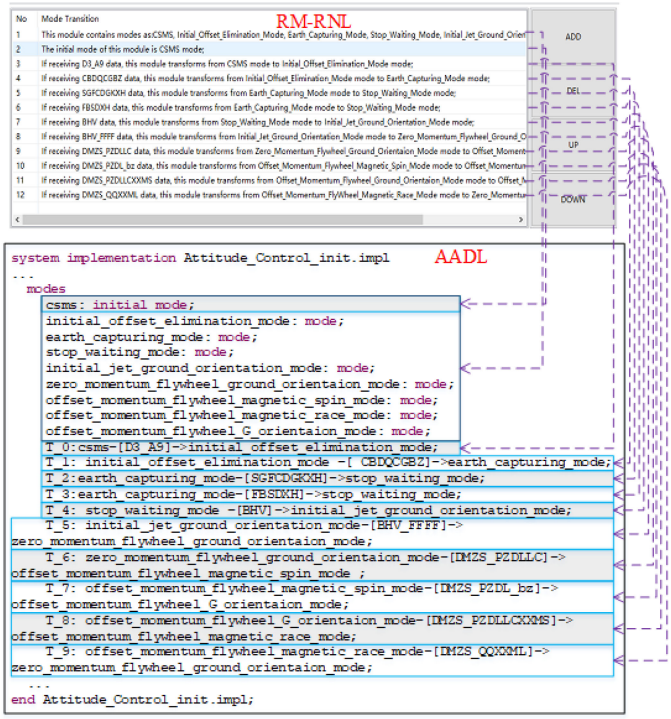

Fig. 14. Traceability link of the attitude control subsystem.

models. To take the mode and mode transition of attitude control subsystem as an example, the upper part of Fig. 14 is the RM-RNL, and the lower part is the corresponding AADL code. The dashed lines in this figure that describe the traceability links between the elements of the RM-RNL and the AADL code.

B. RLCS

The RLCS is a critical subsystem of the rocket launcher system (RLS). The function of the RLS is to control the rocket to perform various operations and automatically execute the function of rocket launch during the period from receiving the launch command to the rocket leaving the launcher. RLCS running on the launch control unit (LCU) computer, and the LCU computer interacts with other modules of the RLS through the bus and/or network. The RLCS can ensure the normal execution of the launch function of the rocket through a serious of hardware–software interaction.

TABLE VI
STATISTICAL DATA OF THE RM-RNL FOR RLCS CASE STUDY

Item	Number	Item	Number
Data in Data Dictionary	107	Terminologies in Glossary	140
System/sub-System Template	13	Function Template	14
Shared function block requirement template	76	Data port	66
Functional sentence	111	-	-

TABLE VII
STATISTICAL DATA OF THE GENERATED AADL MODELS AND TRACEABILITY LINKS FOR RLCS CASE STUDY

Item	Number	Item	Number
Data component	107	System component	11
Port	569	Subprogram	76
Connection of Subprogram	137	Connection of Data	176
Data Port	104	Data Access	219
Event Port	5	Behavior Annex	44
Line of AADL code	5855	Traceability Link	857

The requirement document of RLCS has more than 300 pages. In this article, we only show the requirements of the launch and control subsystem, which has 56 pages. For confidentiality reasons, we consider 24 subfunction modules in the main control layer and process layer of the RLCS in this article.

Similarly, we first construct the data dictionary and the glossary of RLCS and then specify the RLCS requirements through the requirement modeling function of MACAerospace. After the requirement specification with the RM-RNL, the initial AADL models and the requirement traceability links can be generated automatically by the model transformation function of MACAerospace.

The requirements of the RLCS are specified in the RM-RNL, including 13 system/subsystem requirement templates, 14 functional requirement templates, and 76 shared function requirement templates, and so on. The statistical data are shown in Table VI; the statistical data of generated AADL models and traceability links are shown in Table VII.

The traceability links are presented in tabular form in the MACAerospace, and the elements of the RM-RNL and the AADL models are one-to-many mapping, that is, one element in the RM-RNL can be traced to many elements in the AADL models.

Since these traceability links are automatically generated along with the model transformation in our approach, all the generated traceability links should be correct if the transformation rules and algorithms are correct. However,we still manually check these traceability links with industry partners to ensure the effectiveness of traceability links. We ensure that at least one traceability link exists for each element in the RM-RNL and the AADL models.

Then, we improve the initial AADL models through model refinement, which mainly include improving the expression of functional and nonfunctional requirements, reorganizing the structure of the AADL models based on refinement patterns,

and adding the information of platform-specific, and so on. At this time, the original requirement traceability links and the consistency between the requirements and the AADL models will be broken. Therefore, we must maintain the consistency between the requirements and the AADL models, that is, we need to change the requirements and traceability links during the refinement of the AADL models.

The refined AADL models are verified by the verification tools, such as TASM and UPPAAL. If the verification fails, the AADL models will be further modified according to the counterexample, and if so, we can automatically generate the architecture C/Ada codes with the support of the developed code generation tool. Before that, we should perform the second-round refinement, which is required to derive the platform-specific model. MACAerospace provides an user interface for the PSM refinement to assist users adding information that is OS-related, hardware-related, programming-related, communication-related, etc. For example, we need to add the platform features of the VxWorks operating system in the AOCS case, such as watchdog, etc., because the AOCS is developed on the VxWorks operating system. However, we have not consider the characteristics of the operating system in the refinement of the RLCS case study, because it is developed in the Ada language and runs without an operating system.

Requirement traceability can effectively support the analysis and verification of software artifacts and can effectively reduce the maintenance cost of software artifacts. In traditional software development, traceability links between requirements and design models need to be established manually, is often time-consuming and error-prone. To improve this situation, we provide an automatic method to generate the traceability links between requirements and the design models.

VII. Evaluation

In this section, we will evaluate our approach and summarize the threats to its validity. Finally, we also discuss the lessons learned and limitations of our approach.

A. Evaluation Experiments

We mainly evaluate the approach from two aspects: the practicability of RM-RNL and the quality of requirement traceability links.

We conducted a deep investigation when designing the RM-RNL, including fully understanding the requirements description in the actual projects through close interaction with engineers, and verified the RM-RNL using actual industrial cases. The RM-RNL has been modified and adjusted several times. Therefore, we consider that the RM-RNL can meet the basic requests of industrial requirement description. In addition, the RM-RNL has good scalability. We evaluate the practicability of the RM-RNL in this article via a questionnaire. This questionnaire includes four measures: understandability, usability, effectiveness, and restrictiveness. Four statements are designed for these four measures and are presented in Table VIII. We requested the subjects to rate each question relevance on a scale between 0 (completely disagree) and 9 (completely agree).

TABLE VIII
Evaluation the Practicability of the RM-RNL

Measure	Question
Understandability	I understood the RM-RNL and was able to properly apply it.
Usability	RM-RNL is straightforward to apply.
Effectiveness	RM-RNL can effectively reduce the ambiguity of requirements.
Restrictiveness	Did you feel the restriction rule of RM-RNL was too restrictive?

The requirement traceability links are generated by using model transformation in this article, and therefore, these links should be correct in the premise of the transformation rules, and algorithms are correct. Based on this theory, we design the following experiments to illustrate the advantages of our approach.

We set up two groups of comparison experiments. One group manually establishes the traceability links between the RM-RNL and the AADL models, and the other group manually establishes the traceability links between the original requirement documents and the AADL models. Then, we compare the traceability links established by manually with those generated automatically using the MACAerospace toolset.

In addition, we observe the impact of engineering experience in the experimental results by comparing the traceability links established by engineers and students (master's students and fourth-year undergraduate students) in each set of experiments. The experimental results are measured using two metrics, namely, accuracy and recall. In this article, we use $\text{links}_{\text{MACAerospace}}$ to represent the traceability links generated by the MACAerospace toolset and $\text{links}_{\text{manually}}$ to represent the traceability links established by subjects. Then, we define matched links as the links, which appear in both $\text{links}_{\text{MACAerospace}}$ and $\text{links}_{\text{manually}}$. Therefore, the definitions of the accuracy and recall are as follows.

Accuracy is the ratio between the number of matched links and the number of links that are established by subjects

$$\text{accuracy} = \frac{|\{\text{links}_{\text{MACAerospace}}\} \cap \{\text{links}_{\text{manually}}\}|}{|\{\text{links}_{\text{manually}}\}|}. \quad (1)$$

Recall is the ratio between the number of matched links and the number of links that are automatically generated by the MACAerospace toolset

$$\text{recall} = \frac{|\{\text{links}_{\text{MACAerospace}}\} \cap \{\text{links}_{\text{manually}}\}|}{|\{\text{links}_{\text{MACAerospace}}\}|}. \quad (2)$$

B. Evaluation Results

The subjects consist of 12 students (seven fourth-year undergraduate students and five master's students) and six engineers from industry. A lecture is given to the subjects regarding the RM-RNL and the MACAerospace toolset before conducting the experiment. The evaluation results of the RM-RNL obtained by the questionnaire are as shown in Fig. 15.

Fig. 15 compares the evaluation scores given by the 18 subjects for the practicability for the RM-RNL. All measures of RM-RNL are rated highly with a median of either 7 or 8

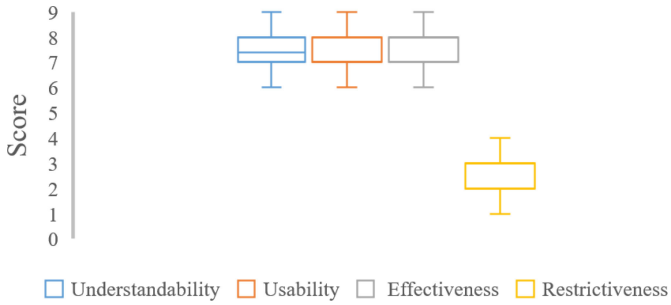

Fig. 15. Evaluation results of practicability for the RM-RNL.

TABLE IX
EVALUATION RESULTS OF THE QUALITY OF REQUIREMENT TRACEABILITY LINKS

Subject	Case Study	Group A		Group B	
		Accuracy	Recall	Accuracy	Recall
Students	AOCS	0.81	0.611	0.935	0.821
	RLCS	0.683	0.544	0.808	0.702
Engineers	AOCS	0.882	0.747	0.961	0.9
	RLCS	0.742	0.616	0.891	0.798

except for restrictiveness (avg: 2.61, med: 3, and std: 1.04). Individual statistics are given follows: understandability (avg: 7.34, med: 7.5, and std: 0.85), usability (avg: 7.28, med: 7, and std: 0.89), and effectiveness (avg: 7.50, med: 4, and std: 0.79). All subjects found RM-RNL to be practicable. We believe that the MACAerospace toolset has improved the understanding and cognition for RM-RNL.

In the evaluation experiment of the quality of traceability links, we divided the 18 subjects into two groups, with each group consisting of six students and three engineers. The subjects established the traceability links between the original requirement documents and the AADL models in Group A, and the subjects established the traceability links between the RM-RNL and the AADL models in Group B. The evaluation results are shown in Table IX.

Comparing the two sets of experiments results, we find the following:

1) The accuracy and recall of Group B are significantly higher than those of Group A.
2) In the same case study, the accuracy and recall of engineers are significantly higher than those of students in Group A, but this advantage is not obvious in Group B.
3) In both groups, the accuracy and recall of the RLCS cases study are lower than in the AOCS case study.

Therefore, we draw the following conclusions.

1) The RM-RNL can effectively improve the understanding of the requirements for subjects and also indirectly prove the effectiveness of the RM-RNL.
2) Our approach can generate more complete requirement traceability links than the manual method, that is, our approach is effective.
3) With the increasing complexity of the software system, it is not reliable to manually establish the traceability links between requirements and the design models.

In addition, requirement traceability links are automatically generated along with model transformations to generate the design models in our approach, and thus, there is no additional time or costs associated with establishing the traceability links between requirements and the design models.

C. Threats to Validity

Besides inheriting all limitations of the underlying software quality engineering and model-driven traceability techniques, our approach exhibits some threats to validity. In order to reduce the possible threat to validity, we communicate with industry partners iteratively to obtain more information and try to make each case study more real. However, we still find some threats to validity of our approach.

1) Correctness: The input of the RM-RNL is given by the engineers that define the system decomposition and other elements based on their understanding. This means that not every input combination is valid, and it becomes increasingly unlikely that the input remains consistent, especially if the input is provided by different engineers. It is important for future work to provide correctness checks based on the consistency of the input, despite the fact that consistency does not imply correctness. The result of the evaluation experiment shows that there is no obvious difference among RM-RNLs created by different subjects if the original requirement document is more standardized.

2) Granularity: It is difficult to establish at what level of granularity traces between requirements and design models should be generated. The trace links between requirements and design models can be created in different level of granularity, e.g., the requirement traceability links can be created at the system level or at the (sub)function level, or even on a requirement sentence, and so on. Then, the engineer has the choice to establish traceability between the model elements, and it is unrealistic to keep under control all requirements and design models at all levels of abstraction.

3) Requirement Specification Based on the RM-RNL: The AOCS and RLCS case studies may not cover all possible situations for safety-critical systems. We see opportunities and needs to apply MACAerospace in other systems of different domains such as the vehicle control system in the automotive domain. Further investigation of the applicability of MACAerospace through larger scale case studies will be conducted in the near future in a real industrial setting.

D. Discussions

From the experiment results reported, we observe that the RM-RNL is overall easy to understand and apply. At the same time, the applicability of the RM-RNL can be further improved since the MACAerospace toolset can be used to enforce the proper usage of keywords specified in the restrictions on the use of control structures.

It is worth noting that inconsistent requirement specification in the RM-RNL may lead to low-quality AADL models and traceability information. However, we believe that inconsistencies among requirement specifications can be reduced if the RM-RNL restrictions are properly applied. In summary, the

RM-RNL can be better applied in practice due to support from the MACAerospace toolset.

During the collaboration with our industrial partner for devising the methodology, developing the tool, and conducting the industrial case studies, we learned the following lessons and identified some challenges when applying our approach in real industrial contexts.

1) In the safety-critical domain, a number of standards (e.g., DO-178B/C and ISO26262 [44]) are recommended to be followed when developing requirement specifications. However, even though guided by such standards, we observed that there are hidden guidelines (i.e., implicit domain knowledge) followed by domain experts and requirements engineers but not documented anywhere. Such hidden rules were hard to obtain, and we spent a lot of effort eliciting such information. We believe there are still more of these rules to discover. Understanding, formalizing, and enabling automated analyses of such requirements require domain knowledge. Furthermore, these aforementioned hidden rules need to be identified and embedded as part of the methodology and tool. We believe that our approach can be easily tailored for accommodating such domain-specific rules. During the domain analysis, a lot of effort was spent on identifying and clarifying such rules with our industrial partner, and the whole process was highly iterative.
2) We observed that some of the requirements need to describe data flows. Doing so with restricted natural languages is not as straightforward as describing control flows. Therefore, our approach is mainly designed for describing control flows, which is similar to AADL. Although our approach supports basic descriptions of data processes, such as data transitions and assignments, it requires more effort compared with the capability of any programming language. Thus, in the future, we will consider how to describe the data flows in a better way.
3) Sentence patterns in RM-RNL (see Section IV) are adequate for describing the requirements of the case studies. However, the case studies are only for an aerospace control system and RLS. If one wants to apply the RM-RNL to other domains, it may not be enough for the requirements be expressed by the predefined sentence patterns. To accommodate this challenge, we implemented our toolset as an extensible framework, which will make it easier to introduce new sentence patterns as well as to generate trace links in the future.
4) Traceability is rarely directly supported by current software development processes [45]. There is a lack of guidance, both with respect to traceability planning and the evaluation of cost and benefit and with respect to when and how to actually carry out traceability-related tasks during software development. Although our approach supports extending model-driven design ideas to the requirement phase, the traceability between requirements and design can be generated automatically when deriving the design model through model transformation. But in MDD, traceability should be regarded not only as an output of model transformation, but also considered in the context of the larger development process. This calls for addressing pre and postmodeling traceability and for traceability of transformation specifications.
5) The autogenerated requirement trace links of our approach are only a coarse-grained traceability relationship. The more complete and accurate requirement traceability relationship needs to be artificially perfected in the refinement of the AADL model or through formal derivation and other methods. In addition, this automatic approach to trace recording only supports one general type of trace link. Furthermore, there is disagreements with the requirement traceability community, which has an understanding of semantics as the meaning of a link, and the modeling community in semantics is comprehended more in the concrete technical context of conditions, events, and actions. Merging both views could be beneficial for both communities. There is also a tradeoff between applying resource extensive, but semantically more accurate, manual techniques, and cost-efficient, but inaccurate, automatic approaches.

VIII. Related Work

We carefully searched for previous work with relation to the study reported in this article and classify the discussion into three topics: requirement specifications in restricted natural language, automated traceability, and model refinement.

A. Requirement Specifications in Restricted Natural Language

Requirements should be easy to understand, since they are usually written as a means for communication between different stakeholders (e.g., users and developers). There are many different ways to document requirements. One common way is to use textual descriptions only. Other ways to document requirements include use cases and customized document templates. For some systems (e.g., safety-critical systems), requirements may even be documented as formal specifications.

In most cases, requirements are represented as natural language, which is easy to understand. However, there are some shortcomings, such as ambiguity and difficult to be processed automatically. In order to solve these problems, restricted natural language came into being. A restricted natural language is also called a controlled natural language (CNL). It is a subset of natural language obtained by restricting the grammar and vocabulary. It aims to reduce ambiguity, redundancy, size, and complexity of requirements, and to facilitate automated analysis. Mavin *et al.* proposed an Easy Approach to Requirement Syntax (EARS) [46], [47]. The EARS provides specific keywords to support the specification of four different types of normal operation and one unwanted behavior. It is simple to use and leads to clear and expressive descriptions of the desired functionality. The EARS has been successfully applied to a variety of complex safety-critical systems, e.g., aero-engine control systems. Holtmann *et al.* proposed a CNL, which describes the requirements of embedded software in automobile areas. CNL can reduce the ambiguity of natural language; it can also detect

inconsistency and incompleteness of requirements, as well as support the automate verification of requirements [48], [49]. CNL is mainly used to describe system requirements of automotive systems. The EARS and CNL were developed primarily for stakeholder requirements, as opposed to technical system requirements.

Use case is another common description for requirements in requirements engineering, and its expressions mainly include use case diagram and use case textual specification. The use case diagram has the exact definition and corresponding modeling standard in UML; the use case textual specification is often described in the form of a template for the use of natural language. The use case textual specification generally includes options such as use case names, descriptions, basic flows, and optional flows. Yue *et al.* developed a modified use case modeling method, namely restricted use case modeling (RUCM) [50]–[52], which contains a relatively perfect with using case textual specification template and a series of natural language used to constraint template writing limit rules (restricted rules). This made the use case description more easy to understand, reduced the ambiguity, and allowed for the automatic generation of the analysis model. In addition, the authors propose a method and a tool called aToucan, to automatically generate a UML analysis model comprising class, sequence, and activity diagrams from a use case model and to automatically establish traceability links between model elements of the use case model and the generated analysis model.

RUCM is a general description method of requirements. It has some shortcomings in the description of safety requirements in SCS, but the RUCM method has good scalability. Therefore, Wu *et al.* proposed a safety RUCM by extending some templates and restricted rules into the existing RUCM. The safety RUCM supports the standardized description of safety requirements [53]. However, it still describes the requirements through use cases, and the description of the domain features of the embedded system is not comprehensive.

Gu *et al.* [54], [55] proposed a formal modeling approach SPARDL as a concise way to specify embedded systems. It can improve the quality of embedded systems in aerospace. SPARDL is a formal requirements modeling method that can accurately describe software requirements, but it is difficult for engineers to use it directly.

AADL provides *ReqSpec* [56], a textual requirement specification language Annex of AADL. *ReqSpec* supports the refinement of requirements along with system designs, qualitative and quantitative analysis of the created requirement specifications, and verification of the associated system architecture models, and thus ensures that requirements are met. *ReqSpec* mainly focuses on the consistency between requirements and AADL models, which is recommended for requirement specifications along with the establishment of AADL models, instead of the transformation of AADL models from requirements.

To summarize, the existing modeling methods for restricted RNLs have their own advantages and disadvantages. For instance, the EARS and RUCM are directly used for writing the text requirement and were developed primarily for stakeholder requirements. AADL *ReqSpec* is primarily used to express traceability between the requirement document and the AADL model. The RM-RNL can be better for specifying the input–process–output features of embedded software. The RM-RNL is a requirement modeling method for SCS and AADL. It combines the advantages of RMCM, EARS, and other methods, which can eliminate the ambiguity of NLRs and barely change engineer's habits of requirement specification. The RM-RNL can also promote the application of MDD in the stage of NLRs.

B. Automated Traceability

Traceability has been defined as the ability to describe and follow the life cycle of software artifacts [30]. van Lamsweerde [57] reported several applications of formal specifications relevant to traceability: refinement of specifications, derivation of test cases, and extraction of specifications from source code are transforming activities, which can produce trace links as by-products. Of course, deriving a trace link as part of a generative or transforming activity is rather intuitive: there is usually a link *derived-from* from the product to the source artifact.

In the context of MDD, traces partially fulfill the same purpose as in requirements engineering because in many tasks, MDD is simply an automation of software engineering. The special characteristic of MDD is the usage of models and automated transformations. Therefore, the artifacts under study are mainly (intermediate) models. This context influences the definitions and semantics of the terms known from requirement traceability and software engineering in general.

Gervasi and Zowghi [58] explored the automatic transformation from NLRs into formal logic. In their approach, they analyzed NLRs with a part of speech tagger and then produced equivalent formulas in predicate logic. The source and target artifacts were stored in a database in order to make the transformation traceable. A similar transformation from requirements to UML models was described by Ilieva and Ormandjieva [59], [60]. While their model generation method could easily support traceability, they did not mention it explicitly.

In addition, results from existing empirical studies [61]–[63] demonstrate that traceability information has beneficial effects on the effectiveness and efficiency of understanding changes, performing requirements inspections, and evolving software artifacts. Ghabi and Egyed introduced a language for expressing uncertainties in traceability relationships between models and code, which is the main benefit of this technique compared with other traceability approaches [64], [65]. They also considered artifacts with different natures that are architectural elements, and extra-functional results utilized a similar approach [66], [67].

Holtmann *et al.* proposed a semiautomatic method for establishing and maintaining requirement traceability in the process of automated development based on the CNL requirement specification method [50], [68], [69]. Meanwhile, Yue *et al.* realized an RUCM tools—aToucan [52]; it can automatically generate a UML analysis model and automatically establish trace links between model elements of the use case model and the generated analysis model. The aToucan is rule based and, thanks to a modular design, facilitates modifications to the set

of transformation rules to accommodate different contexts (e.g., use different parsers).

C. Model Refinement

There are several works studying the specific nature of the relationship between requirements and architectures and the more general problem of model corefinement. In [70], Tang *et al.* proposed an interesting traceability metamodel taking into account the characterization of requirements and architecture elements in terms of problem and solution spaces and capturing design outcomes and decisions. An ontology supporting the designer in corefinement is provided. However, only traceability is managed automatically, and requirements and architecture must be corefinement manually. In [71], Rahimi and Cleland-Huang proposed a pattern of corefinement between requirements and source code. Such patterns provide building blocks for automating traceability maintenance, but, again, corefinement of requirements and architecture is not addressed. In [72], a co-evolution of use cases models and feature model configurations is proposed and implemented with a bidirectional transformation language. In [73], Blouin *et al.* present a semiautomated approach to evolve nonfunctional requirements and their trace links following system's architecture refinement in the context of design space exploration and automated code generation. The approach has been prototyped for AADL models refined with the RAMSES tool and model transformations implemented as story diagrams. In [74], Rahmoun *et al.* proposed an approach that automates the identification of model transformation alternatives (MTAs) taking into account their dependencies, and selections of MTAs based on evolutionary algorithms that produce the best output models with respect to nonfunctional properties.

IX. Conclusion

In this article, requirement traceability was broadly recognized as a critical element of any rigorous software development process, especially for building high-assurance and SCS systems. MDD provided new opportunities for establishing traceability links through model transformations. However, requirement modeling was not involved in the MDD life cycle. To promote the gap between NLRs and the AADL models, we proposed a new requirement modeling method, which was named as RM-RNL. The RM-RNL can promote the application of MDD in safety-critical domains. In the context of MDD, we proposed a method to automatically establish traceability links between the elements of the RM-RNL and the generated AADL models. In addition, we needed to maintain the traceability links when the requirement change and/or AADL models refined. Therefore, we proposed "refinement patterns" to achieve the change of requirements and traceability links. Finally, we demonstrated the effectiveness of our approach with industrial case studies and evaluation experiments and discussed potential threats to its validity.

In the future work, we will further improve the description capability of the RM-RNL and automatically generate more fine-grained requirement traceability links. At the same time, we will realize the automatic maintenance of the requirement traceability during the refinement (and evolution) of the AADL models. In addition, we will consider automatically creating Glossary and Data Dictionary through artificial intelligence technologies.

References

[1] N. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*. Cambridge, MA, USA: MIT Press, 2011.

[2] O. C. Z. Gotel and C. W. Finkelstein, "An analysis of the requirements traceability problem," in *Proc. Int. Conf. Requirements Eng.*, Colorado Springs, CO, USA, 1994, pp. 94–101.

[3] P. Rempel and P. Mäder, "A quality model for the systematic assessment of requirements traceability," in *Proc. 23rd Int. Requirements Eng. Conf.*, Ottawa, ON, Canada, 2015, pp. 176–185.

[4] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA Standard DO-178C, 2011.

[5] J. Cleland-Huang, O. C. Z. Gotel, J. H. Hayes, P. Mäder, and Z. Andrea, "Software traceability: Trends and future directions," in *Proc. Future Softw. Eng. Conf.*, Hyderabad, India, 2014, pp. 55–69.

[6] Z.-B. Yang, "AADL formal semantics and verification & analysis of safety-critical real-time systems," Ph.D. dissertation, School Comput. Sci. Eng., Beihang Univ., Beijing, China, 2012.

[7] C. He and G. Mussbacher, "Model-driven engineering and elicitation techniques: A systematic literature review," in *Proc. 24th Int. Requirements Eng. Conf. Workshops*, Beijing, China, 2016, pp. 180–189.

[8] OMG, "OMG Unified Modeling Language (UML)," 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[9] OMG, "OMG System Modeling Language (UML)," 2017. [Online]. Available: https://www.omg.org/spec/SysML/1.5/PDF

[10] MathWorks, "Simulink," 2018. [Online]. Available: https://ww2.mathworks.cn/products/simulink.html

[11] P. H. Feiler and D. P. Gluch, *Model-Based Engineering With AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Upper Saddle River, NJ, USA: Addison-Wesley, 2012.

[12] P. H. Feiler, B. A. Lewis, and S. Vestal, "The SAE architecture analysis & design language (AADL) a standard for engineering performance critical systems," in *Proc. IEEE Conf. Comput. Aided Control Syst. Des./IEEE Int. Conf. Control Appl./IEEE Int. Symp. Intell. Control*, Munich, Germany, 2006, pp. 1206–1211.

[13] CMU/SEI, "Architecture analysis and design language," 2019. [Online]. Available: http://www.aadl.info/aadl/currentsite/

[14] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni, "Model traceability," *IBM Syst. J.*, vol. 45, no. 3, pp. 515–526, Jul. 2006.

[15] T. Yue, L. C. Briand, and Y. Labiche, "A systematic review of transformation approaches between user requirements and analysis models," *Requirements Eng.*, vol. 16, no. 2, pp. 75–99, Jun. 2011.

[16] S.-L. Kan and Z.-Q Huang, "Detecting safety-related components in statecharts through traceability and model slicing," *Softw.: Pract. Experience.*, vol. 48, no. 3, pp. 428–448, Mar. 2018.

[17] B. Turban, *Tool-Based Requirement Traceability Between Requirement and Design Artifacts*. Wiesbaden, Germany: Springer Vieweg, 2013.

[18] F. Wang *et al.*, "Generating the AADL model based on restricted natural language requirement template," *Ruan Jian Xue Bao/J. Softw.*, vol. 29, no. 8, pp. 2350–2370, 2018.

[19] *Architecture Analysis & Design Language (AADL)*, SAE Standard AS5506C, 2017.

[20] *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface, Annex E: Error Model Annex*, SAE Standard AS5506/1, 2011.

[21] J. Delange and P. Feiler, "Architecture fault modeling with the AADL error-model annex," in *Proc. 40th EUROMICRO Conf. Softw. Eng. Adv. Appl.*, Verona, Italy, 2014, pp. 361–368.

[22] *SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Data Modeling Annex, Annex D: Behavior Model Annex, Annex F: ARINC653 Annex*, SAE Standard AS5506/2, 2011.

[23] P. Dissaux, J. P. Bodeveix, M. Filali, P. Gaufillet, F. Vernadat, "AADL behavioral annex," in *Proc. DASIA Conf.*, Berlin, Germany, 2006, pp. 361–368.

[24] J.-M. Xu *et al.*, "Hierarchical behavior annex: Towards an AADL functional specification extension," in *Proc. 16th ACM/IEEE Int. Conf. Formal Methods Models Syst. Des.*, Beijing, China, 2018, pp. 1–11.

[25] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics," *Sci. Comput. Program.*, vol. 16, no. 2, pp. 103–149, Sep. 1991.

[26] Z.-B. Yang *et al.*, "From AADL to timed abstract state machines: A verified model transformation," *J. Syst. Softw.*, vol. 93, pp. 42–68, Jul. 2014.

[27] G. Behrmann, A. David, K. G. Larsen, "A tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems.* Berlin, Germany: Springer, 2004, pp. 200–236.

[28] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. Lavalley, and L. Sha, "Compositional verification of architectural models," in *Proc. NASA Formal Methods Symp.*, Norfolk, VA, USA, 2012, pp. 126–140.

[29] F. A. Pinheiro, "Requirements traceability," in *Perspectives on Software Requirements*, vol. 753. Boston, MA, USA: Kluwer, 2004, pp. 91–113.

[30] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia , E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[31] P. Lago, H. Muccini, and H. V. Vliet, "A scoped approach to traceability management," *J. Syst. Softw.*, vol. 82, no. 1, pp. 168–182, Jan. 2009.

[32] S. Winkler and J. V. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Softw. Syst. Model.*, vol. 9, no. 4, pp. 529–565, Sep. 2010.

[33] *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Standard 830-1998, 1998.

[34] B. Ramesh and M. Edwards, "Issues in the development of a requirements traceability model," in *Proc. IEEE Int. Symp. Requirements Eng.*, San Diego, CA, USA, 1993, pp. 256–259.

[35] P. Rempel, P. Mäder, and T. Kuschke, "An empirical study on project-specific traceability strategies," in *Proc. 21st IEEE Int. Requirements Eng. Conf.*, Rio de Janeiro, Brazil, 2013, pp. 195–204.

[36] P. Rempel and P. Mäder, "Preventing defects: The impact of requirements traceability completeness on software quality," *IEEE Trans. Softw. Eng.*, vol. 43, no. 8, pp. 777–197, Aug. 2017.

[37] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and Systems Traceability*. London, U.K.: Springer-Verlag, 2012.

[38] S. A. Bohner, "Impact analysis in the software change process: A year 2000 perspective," in *Proc. Int. Conf. Softw. Maintenance*, Monterey, CA, USA, 1996, pp. 42–51.

[39] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards, "Implementing requirements traceability: A case study," in *Proc. IEEE Int. Symp. Requirements Eng.*, York, U.K., 1995, pp. 89–95.

[40] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proc. 33rd Int. Conf. Softw. Eng.*, Honolulu, HI, USA, 2011, pp. 633–642.

[41] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, "Cheddar: A flexible real time scheduling framework," in *Proc. ACM SIGAda Int. Conf. Ada*, Atlanta, GA, USA, 2004, pp. 1–8.

[42] X. Liu, J. Guo, and E. Gill, "Towards model-driven development of AOCS/GNC for small satellite missions," in *Proc. 65th Int. Astronaut. Congr.*, Toronto, ON, Canada, 2014, pp. 3807–3816.

[43] Z.-B. Yang, K. Hu, Y.-W. Zhao, D.-F. Ma, J. P. Bodeveix, "Verification of AADL models with timed abstract state machines," *J. Softw.*, vol. 26, no. 2, pp. 202–222, 2015.

[44] *Road Vehicles Functional Safety*, ISO Standard 26262, 2011.

[45] R. Ramsin and R. F. Paige, "Process-centered review of object oriented software development methodologies," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 202–222, Feb. 2008.

[46] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, "Easy approach to requirements syntax (EARS)," in *Proc. 17th IEEE Int. Requirements Eng. Conf.*, Atlanta, GA, USA, 2009, pp. 317–322.

[47] S. Gregory, "Easy EARS: Rapid application of the easy approach to requirements syntax," in *Proc. 17th IEEE Int. Requirements Eng. Conf.*, Trento, Italy, 2011, pp. 1–2.

[48] J. Holtmann, J. Meyer, and M.V. Detten, "Automatic validation and correction of formalized, textual requirements," in *Proc. 4th Int. Conf. Softw. Testing, Verif. Validation Workshops*, Berlin, Germany, 2011, pp. 486–495.

[49] M. Fockel, J. Holtmann, and J. Meyer, "Semi-automatic establishment and maintenance of valid traceability in automotive development processes," in *Proc. 2nd Int. Workshop Softw. Eng. Embedded Syst.*, Zurich, Switzerland, 2012, pp. 37–43.

[50] T. Yue, L. C. Briand, and Y. Labiche, "A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, Denver, CO, USA, 2009, pp. 484–498.

[51] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, Feb. 2013, Art. no. 5.

[52] T. Yue, L. C. Briand, and Y. Labiche, "aToucan: An automated framework to derive UML analysis models from use case models," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, p. 13, May 2015.

[53] X. Wu, C. Liu, and Q.-X Xia, "Safety requirements modeling based on RUCM," in *Proc. Comput., Commun. Appl. Conf.*, Beijing, China, 2014, pp. 217–222.

[54] B. Gu, Y.-W. Dong, and Z. Wang, "Formal modeling approach for aerospace embedded software," *Ruan Jian Xue Bao/J. Softw.*, vol. 26, no. 2, pp. 321–331, 2015.

[55] Z. Wang *et al.*, "SPARDL: A requirement modeling language for periodic control system," in *Proc. 4th Int. Symp. Leveraging Appl. Formal Methods, Verif. Validation*, Heraklion, Greece, 2010, pp. 594–608.

[56] P. H. Feiler, J. Delange, and L. Wrage, "A requirement specification language for AADL," Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2016-TR-008, Jun. 2016.

[57] A. van Lamsweerde, "Formal specification: A roadmap," in *Proc. Future Softw. Eng. Track ICSE*, Limerick, Ireland, 2000, pp. 147–159.

[58] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, pp. 277–330, Jul. 2005.

[59] M. G. Ilieva and O. Ormandjieva, "Models derived from automatically analyzed textual user requirements," in *Proc. 4th Int. Conf. Softw. Eng. Res., Manag. Appl.*, Seattle, WA, USA, 2006, pp. 13–21.

[60] M. G. Ilieva and O. Ormandjieva, "Automatic transition of natural language software requirements specification into formal presentation," in *Proc. Int. Conf. Appl. Natural Lang. Inf. Syst.*, Alicante, Spain, 2005, pp. 392–397.

[61] A. V. Knethen, "Change-oriented requirements traceability: Support for evolution of embedded systems," in *Proc. Int. Conf. Softw. Maintenance*, Montreal, QC, Canada, pp. 482–485, 2002.

[62] L. Briand, D. Falessi, S. Nejati, M. Sabetzadeh, and T. Yue, "Traceability and SysML design slices to support safety inspections: A controlled experiment," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, Feb. 2014, Art. no. 9.

[63] P. Mäder and A. Egyed, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" *Empirical Softw. Eng.*, vol. 20, no. 2, pp. 413–441, Jun. 2015.

[64] A. Ghabi and A. Egyed, "Exploiting traceability uncertainty between architectural models and code," in *Proc. Joint Working IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, Helsinki, Finland, 2012, pp. 171–180.

[65] A. Ghabi and A. Egyed, "Exploiting traceability uncertainty among artifacts and code," *J. Syst. Softw.*, vol. 108, pp. 178–192, Oct. 2015.

[66] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and performance analysis results," in *Proc. Eur. Conf. Softw. Archit.*, Cavtat, Croatia, 2015, pp. 305–321.

[67] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and extra-functional results," *J. Syst. Softw.*, vol. 125, pp. 15–34, Mar. 2017.

[68] M. Fockel and J. Holtmann, "A requirements engineering methodology combining models and controlled natural language," in *Proc. 4th IEEE Int. Model-Driven Requirements Eng. Workshop*, Karlskrona, Sweden, 2014, pp. 67–76.

[69] M. Daun, M. Fockel, J. Holtmann, and B. Tenbergen, "Goal-scenario-oriented requirements engineering for functional decomposition with bidirectional transformation to controlled natural language: Case study "body control module" Inst. Comput. Sci. Bus. Inf. Syst., Univ. Duisburg-Essen, Essen, Germany, ICB Res. Rep. 55, 2013.

[70] A. Tang, P. Liang, V. Clerc, and H. van Vliet, "Traceability in the co-evolution of architectural requirements and design," in *Relating Software Requirements and Architectures.* Berlin, Germany: Springer-Verlag, 2011, pp. 35–60.

[71] M. Rahimi and J. Cleland-Huang, "Patterns of co-evolution between requirements and source code," in *Proc. 5th IEEE Int. Workshop Requirements Patterns*, Ottawa, ON, Canada, 2015, pp. 25–31.

[72] W.-Z. Zhao, H.-Y. Zhao, and Z.-J. Hu, "A framework for synchronization between feature configurations and use cases based on bidirectional programming," in *Proc. 24th IEEE Int. Requirements Eng. Conf. Workshops*, Beijing, China, 2016, pp. 170–179.

[73] D. Blouin *et al.*, "A semi-automated approach for the co-refinement of requirements and architecture models," in *Proc. 25th IEEE Int. Requirements Eng. Conf. Workshops*, Lisbon, Portugal, 2017, pp. 36–45.

[74] S. Rahmoun, E. Borde, and L. Pautet, "Multi-objectives refinement of AADL models for the synthesis embedded systems (mu-RAMSES)," in *Proc. 20th Int. Conf. Eng. Complex Comput. Syst.*, Gold Coast, QLD, Australia, 2015, pp. 21–30.

Fei Wang was born in 1990. He is currently working toward the Ph.D. degree in software engineering with the Nanjing University of Aeronautics and Astronautics, Nanjing, China.

His main research interests include requirement engineering, safety-critical embedded software, model-driven development, and software engineering.

Zhi-Bin Yang received the Ph.D. degree in computer science from Beihang University, Beijing, China, in 2012.

He is currently an Associate Professor of Software Engineering with the Nanjing University of Aeronautics and Astronautics, Nanjing, China. From April 2012 to December 2014, he was a Postdoctoral Researcher with the Institut de Recherche en Informatique de Toulouse, University of Toulouse, Toulouse, France. His research interests include safety-critical real-time system, formal verification, Architecture Analysis and Design Language, and synchronous languages.

Zhi-Qiu Huang received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1999.

He is currently a Professor and the Director of Software Safety in Computer Science with the Nanjing University of Aeronautics and Astronautics. His current research interests include software safety, formal methods, requirement engineering, and software engineering.

Cheng-Wei Liu was born in 1994. He is currently working toward the master's degree in software engineering with the Nanjing University of Aeronautics and Astronautics, Nanjing, China.

His research interests include safety-critical real-time systems, model-driven development, and Architecture Analysis and Design Language.

Yong Zhou received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2006.

From 2002 to 2009, he was a Lecturer. He is currently an Associate Professor of Computer Science with the Nanjing University of Aeronautics and Astronautics. He regularly teaches discrete mathematics and programming language. His research interests include software engineering, formal methods, as well as artificial intelligence.

Jean-Paul Bodeveix received the Ph.D. degree in computer science from the University of Paris-Sud 11, Orsay, France, in 1989.

Since 1989, he has been an Assistant Professor of Computer Science with the University of Toulouse III, Toulouse, France, where he has been a Professor of Computer Science, since 2003. He has participated in European and national projects related to these domains. His current activities are linked to the study of real-time and synchronous language semantic properties within proof assistants and to real-time modeling and verification. His main research interests include formal specifications, and automated and assisted verification of protocols as well as of proof environments.

Mamoun Filali received the Ph.D. degree in computer science from the Université Paul Sabatier, Toulouse, France, in 1983.

Since 1983, he has been an Associate Professor in Computer Science with ENSEEIHT, Toulouse, France, where he has been a Full-Time Researcher of Computer Science with the Centre National de la Recherche Scientifique, Université de Toulouse, Toulouse, France, since 1985. During the past years, he has been mainly involved in the French nationwide TOPCASED project, where he was concerned by the verification topic. He has also participated in the proposal of the Architecture Analysis and Design Language (AADL) behavioral annex, which has been adopted as part of the AADL SAE standard. His main research interests include the certified development of embedded systems, formal methods, model checking, and theorem proving.