A Model-Based Combination Language for Scheduling Verification*

Hui Zhao[1], Ludovic Apvrille[3], and Frédéric Mallet[12]

[1] Université Côte d'Azur, I3S, INRIA
[2] I3S Laboratory, UMR 7271 CNRS, France
[3] LTCI, Télecom Paris, Institut Polytechnique de Paris

Abstract. Cyber-Physical Systems (CPSs) are built upon discrete software and hardware components, as well as continuous physical components. Such heterogeneous systems involve numerous domains with competencies and expertise that go far beyond traditional software engineering: systems engineering. In this paper, we explore a model-based approach for systems engineering that advocates the composition of several heterogeneous artifacts (called views) into a sound and consistent system model. A model combination Language is proposed for this purpose. Thus, rather than trying to build the universal language able to capture all possible aspects of systems, the proposed language proposes to relate small subsets of languages in order to offer specific analysis capabilities while keeping a global consistency between all joined models. We demonstrate the interest of our approach through an industrial process based on Capella, which provides (among others) a large support for functional analysis from requirements to components deployment. Even though Capella is already quite expressive, it lacks support for schedulability analysis. AADL is also a language dedicated to system analysis. If it is backed with advanced schedulability tools, it lacks support for functional analysis. Thus, instead of proposing ways to add missing aspects in either Capella or AADL, we rather extract a relevant subset of both languages to build a view adequate for conducting schedulability analysis of Capella functional models. Finally, our combination language is generic enough to extract pertinent subsets of languages and combine them to build views for different experts. It also helps maintaining a global consistency between different modeling views.

Keywords: CPS, MDE, Combination Modeling Language, SysML, AADL, Multi-View Design

1 Introduction

CPSs (Cyber-Physical Systems) consists of various components and their interconnections [15]. Thus, the design of the CPSs span over numerous domains of the system. Handling requirements of different domains with different characteristics pushes model-based approaches to their limits.

* This work was financially Supported by the CLARITY project and by a UCN@Sophia Labex scholarship.

Model-Driven Engineering (MDE) is considered as a well-established software development approach that uses abstraction to bridge the gap between the problem space and the software implementation [7][23]. MDE uses models to describe complex systems at multiple levels of abstraction. In this paradigm, models are first-class elements that represent abstractions of a real system, capturing some of its essential properties. Models are instances of modeling languages which define their abstract syntax (e.g., using a metamodel expressed in a class diagram), concrete syntax (e.g., graphical or textual), and semantics (e.g., operational or denotational by means of a model transformation) [12]. As an important issue of MDE, multi-view modeling integrates different models using various DSMLs (domain-specific modeling languages) and abstract various aspects of systems and sub-systems, such as scheduling, behaviors and functionalities. Therefore, it is critical to understand the relationship among (meta) models. The modeling languages, such as Systems Modeling Language (SysML) [10] and Architecture Analysis and Design Language (AADL) [8], have been enhanced to better handle the CPS design, but, to the best of our knowledge, none of them cover all the necessary domains to handle all the characteristics of CPSs effectively. The increasing complexity of CPSs brings a critical challenge for developers to deal with different domains. Developers have to rely on domain-specific languages to handle different domains, which results in a proliferation of languages and increasing design complexity of CPS [9, 15]. Furthermore, the gaps between languages and platforms bring several problems, for example, the specification of the CPS that has problems with inconsistency and incoherency. All of those problems are exposed at integration and simulation stages, they also augment the complexity of CPS and make it skyrocketing.

To tackle these problems, a new approach is required to efficiently take advantage of each existing language and combine them together. To this end, the existing approaches can be classified into two types. The first type is to continuously integrate the necessary languages into an existing development platform, and then progressively build a comprehensive development platform. However, this type of approach could encounter a never-ending process and result in a gigantic framework, thus difficult to use, maintain, etc. The second type is to keep each language (or tool) isolated, and relate some of the elements from each language with (sub) meta-model, so as to allow different kinds of analysis offered by each method (e.g., scheduling analysis, safety analysis). Furthermore, each domain expert can work independently with the second type of approach. However, since each language has its own characteristic, such as syntax and semantics, the gaps between different languages have to be eliminated in order to handle the consistency issues.

Our previous work [27] introduced a formal approach to combine two modeling languages by defining how to link two (sub-)metamodels. More precisely, thanks to our approach, consider two models m_1 and m_2 of two different modeling languages: m_2 can automatically be augmented with some information of m_1 so as to perform verification on the enriched model (e.g., scheduling, timing, safety), and then verification results can be backtraced to m_1.

In order to validate our contribution, SysML and AADL are selected as two target languages, and their support environments (tools) Capella/Arcadia and OSATE2 [4] are used to show the design of example system.

The paper is organized as follows. In section 2, we first identify the workflow of the proposed approach. Then, we explicitly present the reinforced language and the operators in section 3. In section 4, we apply these operators on functional and physical views. To evaluate the proposed formal approach, train traction control systems are used to demonstrate the architecture and analyze scheduling in section 5. Section 6 illustrate the related work. Finally, section 7 concludes the paper and presents our future work. It should be noted that, in sections 2, 3, 4 and 5, all elements on the left of transformation rules belong to metamodels of Arcadia, and all elements on the right are from the AADL metamodels. These metamodels have been imported by default, and their prefix (e.g., *MM.Arcadia*.function) are omitted for conciseness.

2 Our Approach

In this section, we describe the workflow we propose using an example based on Arcadia and AADL, as shown in figure 1 [27]. Arcadia is well adapted to describe how to allocate functions, while AADL focuses on the concrete execution behaviors of components. In this paper, we use transformation to enhance Arcadia with the scheduling analysis features of AADL. The transformation is performed by proposing a set of rules and operators to specify the relationships at the M2 level. Those relations are used for model transformation purpose and a set of all relationships is called Transformation Rule Library (TRL). More specifically, these rules are used to establish a relationship between Arcadia and AADL metamodels in a Transformation Rule Library. We assume that Arcadia and AADL define concepts that can be put in relation thanks to the proposed rules.

As shown with the green part in the figure 1, an Arcadia function allocated to a processor can be related to a ”thread” in AADL. Additional attributes in Arcadia must be added (e.g., period and execution time) when one feature has no equivalence, as shown with a red part in figure 1). Then, the elements of metamodels are chosen manually depending on the requirements of the project. Finally, the workflow has four steps. In step one, we can get a temporary combinational metamodel (TCM) at run time by using TRL once the equivalence relations between the two metamodels have been settled. In step two, the TCM can be used to combine an AADL model with elements of an Arcadia model, then the new AADL model can be exported into OSATE for further editing. In step three, the Cheddar analysis tool [22] is used to conduct scheduling simulation. This tool can be used to detect designing flaws, time and resources conflicts. In step four, it traces back the results to the Arcadia model in order to help the designer enhancing the performance of his/her model.

[4] http://osate.org/index.html

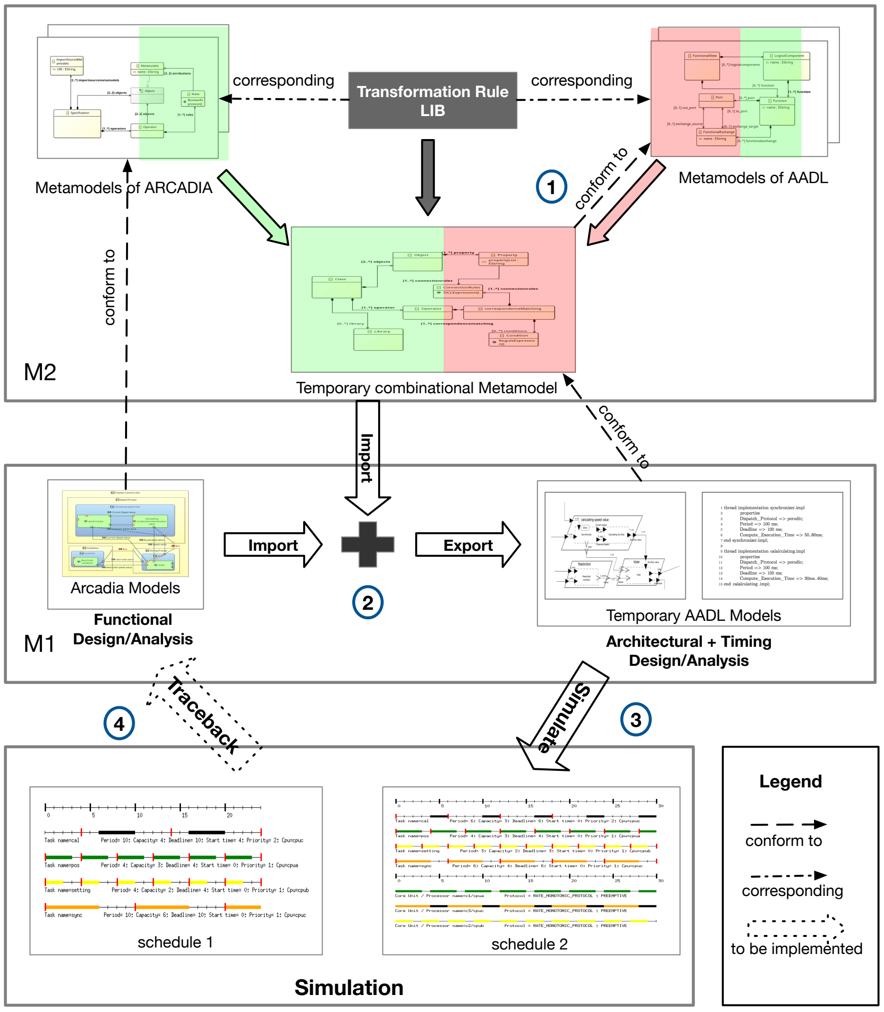


Fig. 1. Overview of Workflow

3 Model Combination Language

The proposed Language is a dedicated (meta) language to extend and enrich one DSML's capability by combining the other DSMLs. With this language, an integration engineer can explicitly capture combination scenarios at the language level. Combination pattern is used to specify different combination relationships. Specific operators are provided to build up Transformation Rule Expression (TRE), a set of TRE defines a TRL (Transformation Rule Library) which specifies how to combine different (meta) models' elements. Once the TRL is completed, it can be parsed by an automatic tool. Afterwards, the tool can perform the transformation automatically. The concept of combination language is illustrated in figure 2.

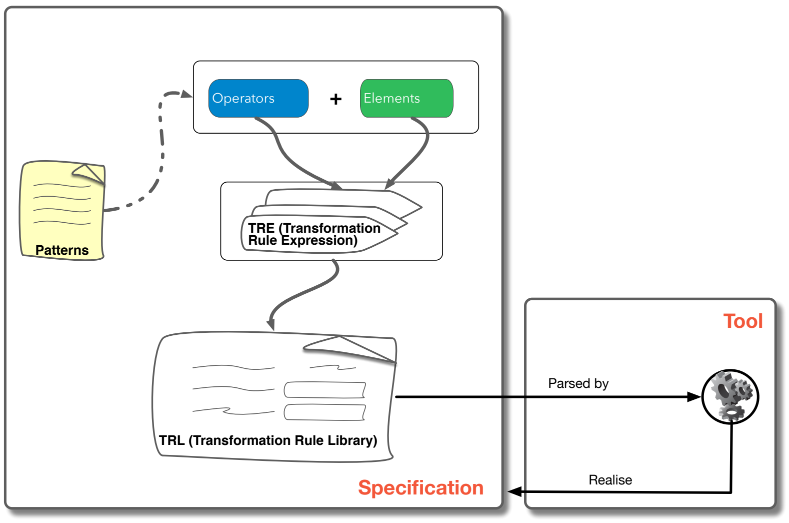

Operators
+
Elements
Patterns
TRE (Transformation Rule Expression)
TRL (Transformation Rule Library)
Specification
Tool
Parsed by
Realise

Fig. 2. Concept of Combination Language

3.1 Specification

A specification consists of combination patterns and corresponding TRL. It defines what and how elements from different models are combined. Once it is specified, integration experts can share this specification thus allowing the reuse and tuning of TRL. As a specification can explicitly describe combination relationship, it also can be used to decompose models by bi-directional techniques for some decomposition needs.

3.2 Combination patterns

Currently, We predefine a number of essential combination patterns, which provide all the declarations used in all the following examples. However, thanks to our language, designers can build other combination patterns depending on their problems and requirements. Certainly, they have to define some new combination patterns in the form of TRL.

1. ***Association***: The association pattern is the most common phenomenon and easier to understand. It is used to indicates one element associate to another element and their related sub-elements (for example, its embedded element or associated attributes).
2. ***Removal***: The removal pattern indicates the situation, where some element does not be needed for new models according to requirements.
3. ***Correspondence*** The Correspondence pattern indicates building an equivalence relationship among a set of elements.
4. ***Notation***: The notation pattern aims to hint people to add some extra information which is not existing in model. For example, the dependency relationship among the model's elements, and the nature of the elements.

3.3 Abstract syntax of Combination Language

We give an abstract syntax of Combination Language by using a metamodel expressed in a class diagram (shown in figure 3). The major element of Combination Language is a specification that contains Patterns, Operators and TRL. The specification requires importing at least two (meta) models. The imported (meta) models serve as a source of a set of candidate elements for following operations. An operator selects the elements and their attributes from imported (meta) models, and it also specifies how to combine selected elements with a clear relationship.

Each operator contains a Transformation Rule Expression which relies on a strict definition by EBNF (Extended BackusNaur Form). Symbols are used to construct the TRE. For instance, for adding security properties to a logical component of Capella, it has to specify the corresponding element and their related attributes in TTool by using TRE.

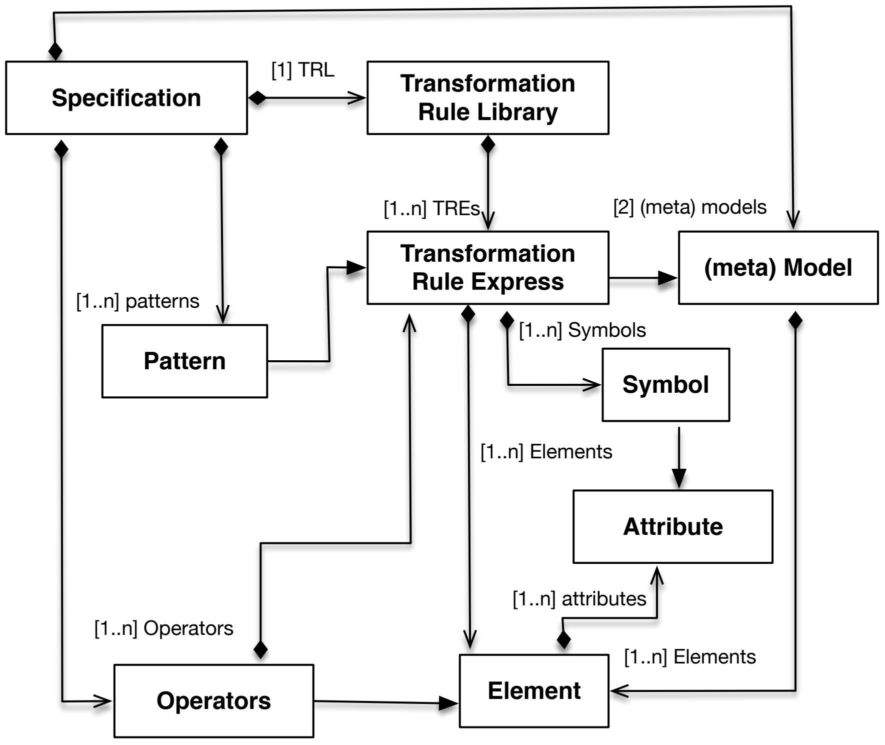


Fig. 3. A simplified view of abstract syntax of combination language

3.4 Meta symbol and notations rule expression

In this subsection, we firstly introduce some notations and meta symbols which are fundamental elements for constructing the well-defined Transformation Rule Expressions (see table 1). For the propose to obtain strict definition and non-ambiguous Transformation Rule Expression pattern, we use EBNF to define TRE. EBNF is a notation technique for context-free grammars [5], often used to describe the syntax of languages [17].

Symbol	Meaning
Γ	Transformation Rule
;	End of rule
:	Separate elements
$\rightsquigarrow$	Transfer
<>	Parent node
{ }	Attribute
[ ]	Optional value
|	Alternative
+	Object to be created
$\neg$	Ignorer
@	Notation

Table 1. Symbols of transformation rule expression

The detail literal meaning of symbols are as below:

1. A Transformation Rule Expression begins with "Γ" and ends with ";".
2. The symbol "$\rightsquigarrow$" indicates a transfer action.
3. A transfer action contains the source elements which in the left side of "$\rightsquigarrow$" and the target elements in the right side. A simple example is as bellow:

$$\Gamma < parent > source \rightsquigarrow target;$$

4. Symbol ":" separates each part of TRE.
5. An angle brackets "<>" encloses the parent node if the element has one or more parent nodes.
6. A parentheses "{ }" enclose attributes
7. A square braces "[ ]" delimit optional elements.
8. The alternative value is separated by a pipe "|". For example, The *port* has a directional attribute called *Direction* which could be *in* or *out* shown as:

$$P_{ort} : \{Direction[in|out]\}$$

[5] https://en.wikipedia.org/wiki/BackusNaur_form

9. Symbol "@" indicates the notations which are used to add some extra informations such as dependency and nature. The extra informations are handled as the same as operational value: enclosed in [ ]; separated by ",". For example, $P_{ort}@[ModelA, Security]$ means element Port belongs to ModelA and is used for Security purpose (view). In such situation, it makes tools automatically display or hide the element P_{ort} which is in modelA and for security view in the following process.

With those symbols, we can build up plentiful TREs. Some more detailed examples of Transformation Rule Expressions are shown in the listing 1.1.

3.5 Abstract syntax of rule expression in EBNF

As we mentioned in the previous subsection, the TRE consists of one or more sequences of symbols. We list here the context-free syntax in EBNF in this subsection.

$\langle expression\rangle$ ::= Γ $\langle term\rangle\rightsquigarrow\langle term\rangle$;|$\langle expression\rangle$:$\langle term\rangle$; |$\langle operator\rangle$ $\langle term\rangle$;

$\langle term\rangle$ ::= $\langle element\rangle$ | $\langle operator\rangle\langle element\rangle$ |$\langle operator\rangle\langle element\rangle\langle operator\rangle$

$\langle operator\rangle$ ::= '@' | '+' | '$\neg$' |'$\rightsquigarrow$'

$\langle element\rangle$ ::= $\langle element\rangle$|$\langle attribute\rangle$ |$\langle optional\ value\rangle$

3.6 Operators and semantics

The context-sensitive syntax and the operational rules could also be considered to be semantics instead of syntax. For example, the context-sensitive syntax is called static semantics in the UML specification documents from OMG [18]. In our case, it specifies how an instance of a construct can be meaningfully connected to other instances.

In order to make the TRE more clearly and precisely, we firstly present a set of relationships definitions formally. That is used to help users understand the semantics of the operator and to avoid ambiguity and misunderstanding. Secondly, we propose a set of operators to build up Transformation Rule Expression, which represents operations between (meta) models (e.g., transforming, creating, ignoring) in a systematic way. They may also help users to understand the following TRE examples.

We define a relation in the sense of set theory. Let A and B be a set of elements respectively, with a, b, c and x, y, z: elements of model, this is written as $A \supsetneq a, b, c$ and $B \supsetneq x, y, z$.

- **Relationship**: If the ordered pair (a,x) in our relation, we write $\mathcal{R}(a,x)$ or $a\mathcal{R}x$ for simplicity. It is also a boolean function. $\mathcal{R}(a,x)$ is true means existing a relation between a and x.
- **Equivalence**: $\mathcal{E}(a,x)$ is a boolean function that is true if and only if a semantically equals to x. By function $\mathcal{E}(a,x)$ holds $\mathcal{R}(a,x) \wedge \mathcal{E}(a,x)$.
- **NotIn**: $\neg a$ is a boolean function. If it is true, that means there are any corresponding elements in set of B (x,y,z) which either have a relationship with a, nor semantically equal to a. Formally,

$$\neg\mathcal{R}(a,\{x,y,z\}) \vee \neg\mathcal{E}(a,\{x,y,z\})$$

Operators:

(a) ***Transferring operator:*** We use $\rightsquigarrow$ indicates transferring operator, for example, $a \rightsquigarrow x$ it means that transfer from a to x, if and only if $\mathcal{E}(a,x)$ is true, in other words, a and x is *Equivalence* relationship.

(b) ***Creating operator:*** In the case of creating a new attribute, put the name of an attribute in the parentheses with plus ”{ }+”, that is used to present the option which is to be created. For example, $\Gamma a \rightsquigarrow x : \{y\}+;$, it means that transfer from a to x and add y attribute, if and only if $y \supset x$ and $\mathcal{E}(a,x) \wedge \mathcal{R}(a,y) \wedge \mathcal{R}(x,y)$ is true. An example in practice is below,

$$\Gamma Port \rightsquigarrow Port : \{Type[data|event|dataevent]\}+$$

A *port* will be transferred to another *port* element, and to create new attribute named *Type* that associates to *port* with three optional value (data, event, data and event).

(c) ***Ignoring operator:*** This operator is used for some ignored attributes and objects. It is denoted with symbol ”$\neg$” which is in front of the object. For example, $\neg a$, it means a is **NotIn** object of set B. Formally, $\neg\mathcal{R}(a,B) \vee \neg\mathcal{E}(a,B)$.

(d) ***Notation Operator:***This operator is used for tagging the nature of attribute of an element. There is an example: $P_{ort}@[ModelA, Security]$. It can present two attributes of element P_{ort} with two tags. One is *ModelA*, indicating that the element P_{ort} belongs to *ModelA*. In other words, It represents a dependency relationship between this element P_{ort} and element *ModelA*. Another is *Security*, represents an element P_{ort} for Security purpose. It would be used to catalog the elements for displaying or fast selecting purpose.

TRE examples with semantics: Transformation Rule Express (TRE) represents the transforming relationships. It would be used for guiding the integration engineer or for reading by automated transformation engine. We use some more detailed examples of Transformation Rule Expressions to explicitly explain how it works. Please refer to the TRE table which is in the listing 1.1.

In line 1 of this example, we firstly transfer an element *port* (it has direction attribute) of A to an object element *port* of B, and add a new attribute *Type* with three optional value (date, event or data event). These ”type value” can be recognised by model B’s DSML and the supported environment. The added attribute can be used to continue further design as well. In line 2, it is similar to previous one, but the object element *Port* has a parent node called *feature* which is enclosed in a pair angle brackets.

Secondly, in line 3, it shows an ignored element, in which the source element may not be found a corresponding one in the object model, or the source element is not needed by the object model. Finally, in line 4, it is a *Equivalence* relationship between the source element and the object element, in other words, it’s a set of one by one transformations which transfer ”Ex_{fun}” to ”*connection*”, ”Source” to ”source” and ”Target” to ”target”, respectively.

---

1 ΓP_{ort}:{Direction[in|out]}$\rightsquigarrow$ <feature>:Port:{Direction[in|out]}:{Type[data|event|data event]}+;

2 $\Gamma PP \rightsquigarrow$ <feature>:Port:{Direction[in|out]}+:{Type[data|event|data event]}+;

```
$\Gamma P_{ort}$:$\neg${ordering};
$\Gamma Ex_{fun}$:{Source}:{Target} $\rightsquigarrow$ <connections>:connection:{source}:{target};
```

Listing 1.1. The example of transformation Rule Expressions

4 Transformation Rule Library

As we described in the above section, the Transformation Rule Express play an important role in the transformation process. Hence, in this section, we will show how to construct a set of TREs called the Transformation Rule Library (TRL). We also respectively present the following views, functional view and physical view in Arcadia (SysML) and AADL. Each view contains one or more metamodels which represents as a x-tuples.

4.1 Functional view

Logical components in Arcadia The logical components in Arcadia contain a set of member elements, such as logical component containers, functions, ports, and functional exchanges. In the Arcadia, Functional diagrams consist of a set of SysML blocks and its interactions, named *Logical components*; The notion of Logical components enables better expression of system engineering semantics compared to SysML, and particularly, reduces the bias towards software. SysML block definition diagrams (BDDs) and internal block diagrams (IBDs) are assigned to different abstract and refined layers, respectively. The definition of a block in SysML can be further detailed by specifying its parts; ports, specifying its interaction points; and connectors, specifying the connections among its parts and ports. This information can also be visualized using logical components in Arcadia. In the definition 1, we present a metamodel of an instance of logical components.

Definition 1. *(Logical Component)*

A logical component (**LC**) is 5 tuples,

$$\mathcal{LC} =< C_{omp}, F_{un}, P_{ort}, Ex_{fun}, M_{cf} >$$

where,

$$C_{omp} = \sum_{i=1}^{\infty} F_{uni}$$

is a logical component container which contains a set of functional elements.

F_{un} is a finite set of functional block include their name and id attributes. P_{ort} is a finite set of functional ports including directions and allocation attributes. $Ex_{fun} \subseteq P_{ort} \times P_{ort}$ denotes a finite set of functional exchange (connection) between two functional ports, it must be pair, one is source, another is target. $M_{cf} : \Sigma F_{un} \rightarrow C_{omp}$ allocate functions to a logical component container.

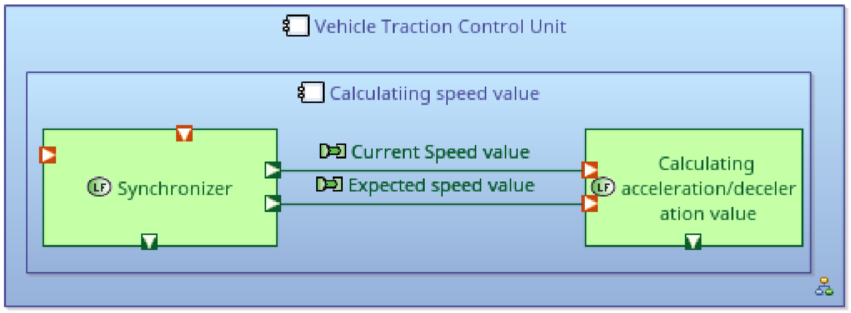


Fig. 4. An example of functional view of vehicle traction control unit in ARCADIA

In the figure 4, there is a functional instance model of a part of a vehicle traction control unit in ARCADIA as an example. The blue rectangle is named logical component in Arcadia, but we consider it as a function's container, we thus call it *logical component container* C_{omp} in this paper. The green rectangle are functions F_{un} which are contained by C_{omp}. The element M_{cf} has represented this allocation relationship between logical component containers and functions $M_{cf} : \Sigma F_{un} \rightarrow C_{omp}$. The deep green square with the white triangle is the outgoing port (P_{ort}), which connects to an incoming port (P_{ort}) that is drawn as a red square with white triangle and the green line is the functional exchange between two functional ports (Ex_{fun}).

The metamodels of software in AADL AADL is able to model a real-time system as a hierarchy of software components, predefined software component types in the category of the components such as thread, thread group, process, data, and subprogram are used to model the software architecture of the system.

Definition 2. *(Software Composition)*

A **SC** is a 4-tuples:

$$SC =< Type, Port, Connection, Annex >$$

where *Type* specifies the type of components (e.g, system, process, thread). *Port* is a set of communication point of component. Port could be different types such as **data** port, **event** port and **data event** port. And, port can specify the direction such as **in** port, **out** port, **in out** port. *Connection* is used to connect ports in the direction of data/control flow in uni- or bi-directional. *Annex* is defined for the refinement of component, in this paper, we used hybrid annex to explicitly describe the both discrete and continuous behavoir of train traction control system.

Hybrid Annex We use the HA to declare both discrete and continuous variables in the *Variables* section, and the initial values of constants are given in *constant* section.

Assert is used to declaring predicates which may be used with invariants to define a condition of operation. The *behavior* section is used to specify the continuous behavior of the annotated AADL component in terms of concurrently executing processes, and use continuous evolution — a differential expression to specify the behavior of a physical controlled variable of a hybrid system. The communication between computing units and physical components are an essential part of a hybrid system, Communication between physical processes uses the channels declared in the *channel* section, and communicate with an AADL component relies on ports that are declared in the component's type. Continuous process evolution may be terminated after a specific time or on a communication event. There are invoked through timed and communication interrupt, respectively. A timed interrupt preempts continuous evolution after a given amount of time. A communication interrupt preempts continuous evolution whenever communication takes places along any one of the named ports or channels. The definition 3 gives a metamodel of Hybrid Annex which does not exist in SysML-based environment.

Definition 3. *(Hybrid Annex)*

A *Hybrid Annex* is a 8-tuples:

$$\mathcal{HA} = < Ass, Ivar, Var_{hd}, Cons_{hd}, P_{roc}, ChP, Itr, B_{itr} >$$

where Ass is a finite set of assert for declaring predicates applicable to the intended continuous behavior of the annotated AADL component. $Ivar$ is associated with assert to define a condition of operation that must be true during the lifetime. Var_{hd} is a finite set of discrete and continuous variables. $Cons_{hd}$ is a finite set of constants which must be initiated at declaration. P_{roc} is a finite set of processes that are used to specify continuous behaviors of AADL components. ChP is a finite set of channels and ports for synchronizing processes. Itr is a finite set of time or communication interrupts. $B_{itr} : Itr \to P_{roc}$ binds interrupts to related processes.

Functional elements transformation rules The table 2 shows the correspondence between AADL and Arcade elements. The Additional attributes column are the attributes to be created during the transformation. According to this table, we can easily write the transformation rules to transforming Arcadia to AADL on functional parts, denoted $\mathcal{LC} \rightsquigarrow \mathcal{SC} + \mathcal{HA}$. An example as below (listing 1.2 [27]):

```
ΓC_omp ⇝ Type[ system|process]:{Runtime_Protection[true|false]}+;
ΓF_un ⇝ Type[abstract|thread]:{Dispatch_Protocol[Periodic|Aperiodic|Sporadic|Background|
        Timed|Hybrid]}+;
 ...
```

Listing 1.2. Functional elements transformation rules example

4.2 Physical view

Execution Platform in AADL Processor, memory, device, and bus components are the execution platform components for modeling the hardware part of the system. Ports and

Arcadia	**AADL**	**Additional attributes**	**Notation**
Logical component container (C_{omp})	System,Process	{Runtime_Protection[true|false]}+	@[function|AADL|scheduling]
Function (F_{un})	Abstract, Thread	{Dispatch_Protocol[Periodic|Aperiodic|Sporadic|Background| Timed|Hybrid]}+	@[function|AADL|scheduling]
Port (P_{ort})	Port	{Type[data|event|data event]}+	@[function|AADL|scheduling]
Functional Exchange (Ex_{fun})	Connection	∅	
∅	Annex	{Type[abstract| thread]}:{annex}+	@[function|AADL|scheduling]
Physical Node (N_{ode})	Device,Memory,Processor,Bus	{Dispatch_Protocol}+:{Period}:{Deadline}+:{priority}+	@[physic|AADL|scheduling]
Physical Port (PP)	∅	$\neg PP$	@[physic|AADL|scheduling]
Physical Link (PL)	Bus/BusAccess	{Allowed_Connnection_Type}+:{Allowed_Message_Size}+: {Allowed_Physical_Access}+:{Transmission_Time}+	@[physic|AADL|scheduling]

Table 2. Functional and Physical elements correspondence table

port connections are provided to model the exchange of data and event among components. Functional and non-functional properties like scheduling protocol and execution time of the thread can be specified in components and their interactions.

Definition 4. *(Execution Platform)*

A **EP** component is defined as a 3-tuples:

$$\mathcal{EP} =< EC, BA, C_{onn} >$$

where,*EC* defines the execution component such as processor, memory, bus and device. *BA* defines the BusAccess which is interactive approach between **bus** component and other execution platform components. $C_{onn} \subseteq EC \times EC$ denotes a finite set of connection between two components via bus device.

Physical components in Arcadia The physical component in Arcadia consists of physical Node, Port and Link. The Physical Port and Link correspond to port and bus connection in AADL. There are some choices when the physical Node is translated to AADL such as device, memory, and processor, hence the designer has to point out what type of target component during transformation by using transformation rule express.

Definition 5. *(Physical Components)*

A Physical components is 3-tuples,

$$\mathcal{PC} =< N_{ode}, PP, PL >$$

where, N_{ode} is a execution platform, named *node* in Arcadia, it could be different type of physical component (e.g, processor, board). *PP* is the physical component port. *PL* is physical link, it could be assigned a concrete type such as *bus*.

Figure 5 is shown as a part of physical instance model of vehicle traction control unit in ARCADIA. We can see the yellow parts are the physical node (N_{ode}) and the red line is the physical link (*PL*) named bus in this case which connects to two physical ports (*PP*), the small square in dark yellow.

Physical elements transformation rules According to the table 2, we can easily write the transformation rules for physical elements. Listing 1.3 [27] shown as a part of the code to transform the physical component from Arcadia to AADL.

```
ΓN_ode ⇝ [Device|Process|Memory|Bus]:{Dispatch_Protocol}+:{Period}:{Deadline}+:{
    priority}+;
ΓPP ⇝ ¬PP;
ΓPL ⇝ Bus/BusAccess:[{Allowed_Connnection_Type}+:{Allowed_Message_Size}+|{
    Allowed_Physical_Access}+:{Transmission_Time}+];
```

Listing 1.3. Physcial elements transformation rules example

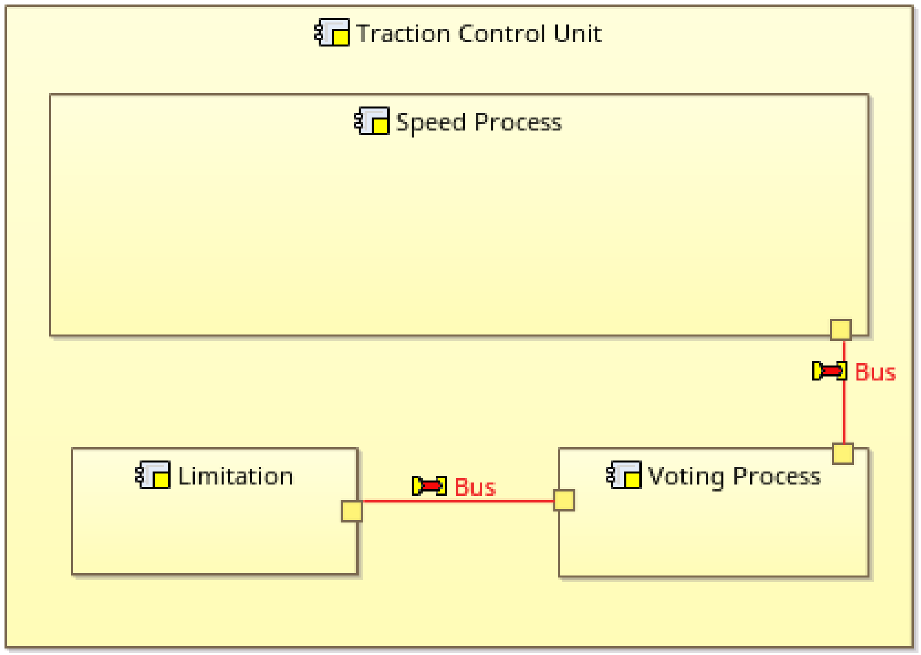


Fig. 5. An example of physical view of vehicle traction control unit in ARCADIA

What we have to especially explain is the physical link part (see line 3). The Bus device could be a logical resource or hardware component. Hence, the bus device has different properties depending on the role. When the bus is considered as a logical resource, it contains the properties *Allowed_connection_type and Allowed_Message_Size*. When the bus is hardware, it contains *Allowed_Physical_Access and Transmission_Time*. Therefore, we write the rules that either

$$\{Allowed\_Connnection\_Type\}+ : \{Allowed\_Message\_Size\}+$$

or

$$\{Allowed\_Physical\_Access\}+ : \{Transmission\_Time\}+$$

5 Case Study

To show the efficacy of our approach in transforming and using produced AADL models to analyze the properties, this section presents the experimental results of analyzing the traction controlling unit of railway signaling system. By using our proposed approach, we transfer and extend Arcadia metamodel, and design AADL using OSATE2 with the generated metamodel. once the concrete models have been created, the scheduling property is chosen to show analysis ability through Cheddar tool [22].

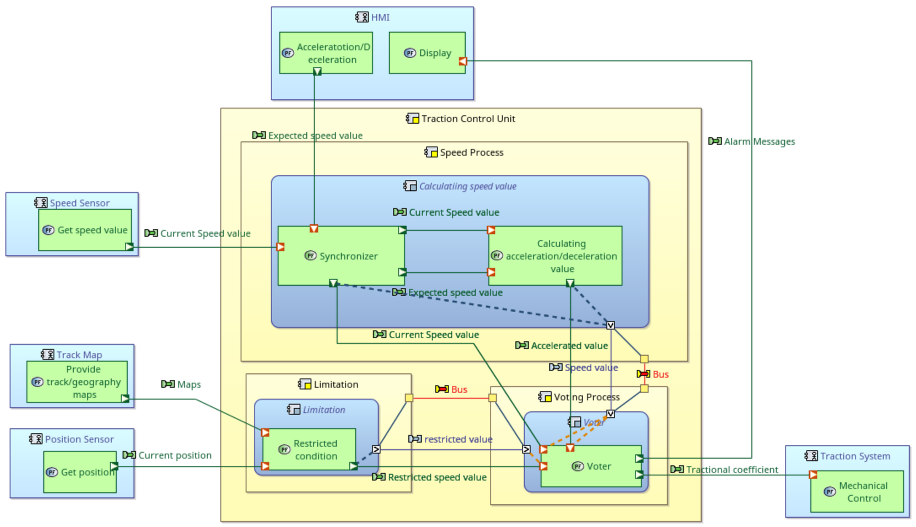


Fig. 6. Arcadia model of TCU system

5.1 Train Traction Control System

Train movement is the calculation of the speed and distance profiles when a train is traveling from one point to another according to the limitations imposed by the signaling system and traction equipment characteristics. As the train has to follow the track, the movement is also under the constraints of track geometry, and speed restrictions and the calculation becomes position-dependent. The subsystem of calculating the traction effective and speed restrictions is therefore critical to achieving train safe running. Nowadays, Communication Based Train Control (CBTC) system is the main method of rail transit (both urban and high-speed train) which adopts wireless local area networks as the bidirectional train-ground communication [28]. To increase the capacity of rail transit lines, many information-based and digital components have been applied for networking, automation and system inter-connection, including general communication technologies, sensor networks, and safety-critical embedded control system. A large number of subsystems consisting of modern signaling systems of railways, therefore, system integration is one of the key technologies of signaling systems; it plays a significant role in maintaining the safety of the signaling system [26].

This paper uses a subsystem which called Traction Control Unit system (TCU) from signaling system of high-speed railway. We use this TCU system to illustrate the model transformation from engineering level to detailed architectural level and verified the instance models. The functional modules such as calculation and synchronization will be transformed using our approach, and then non-functional properties such as timing correctness and resource correctness will be verified by schedule analysis tool Cheddar [22].

First, we start with component functional views and physical view analysis by designing system models in Arcadia (shown in figure of TCU 6 [27]). The functions of

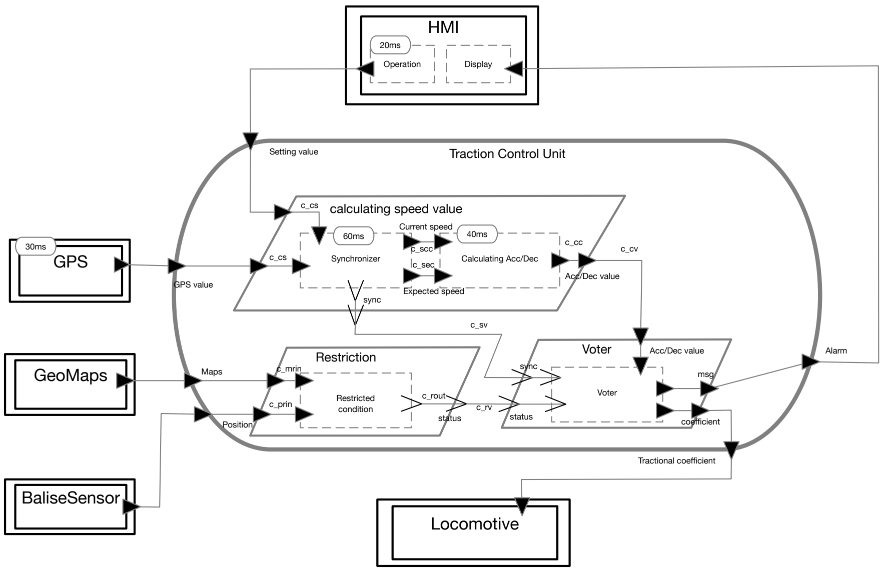


Fig. 7. AADL model of TCU system

the traction control system are to collect the external data by sensors such as a speed sensor. The data from Balise sensors is used to determinate the track block, and then it is going to seek the speed restriction conditions by matching accurate positioning (if the track blocks are divided fine enough) and digital geometric maps data. Meanwhile, calculating speed unit received the speed data from GPS and speed control commands from HMI (Human-Machine Interface) periodically. GPS data provides speed value periodically (we set a period of 30 seconds in this case), and HMI data sustainedly send the operation command with the period of 20 seconds till the value changed (e.g., expected speed value), then the calculating unit has to output an acceleration value and export to the locomotive mechanical system. Although they are periodic, the external data do not always arrive on time due to transmission delay or jitter. Therefore, we should use a synchronizer to make sure they are synchronized. Otherwise, the result would be wrong with asynchronous data. Similarly, to ensure the correctness of the command of acceleration (or deceleration), we applied a voting mechanism which can ensure the result is correct as much as possible. The voter must have the synchronized signal and restriction condition to dedicate to output the acceleration coefficient request to the locomotive system. The AADL diagram is shown in figure 7 [27].

5.2 Model transformation

Using the Arcadia2AADL tool, the metamodel of the TCU system in Capella is translated into the corresponding AADL metamodel with the rules and approach which describes in section 4. For instance, on the one hand, the function class is translated into the thread in AADL. To analyze the timing properties, several attributes also have been added such as protocol type, deadline, execution time, period.

On the other hand, the physical part element Node translates to the processor in this case. Differ from simple physical Node in Arcadia; the processor element attaches rich properties such as scheduling protocol (scheduler type), process execution time. The allocation relationships on both physical and functional parts are translated into AADL as well.

5.3 Schedule verification

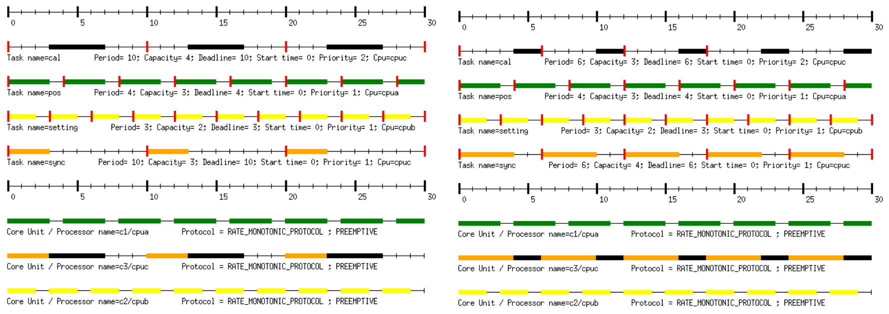
The external data and internal process work sequentially is an essential safety requirement of the system, and each task should be scheduled properly. However, in real-world, the risk of communication quality and rationality of scheduling must be taken into account. Therefore, the schedule verification is a way to evaluate system timing property. An Ada framework called Cheddar which provides tools to check if a real-time application meets its temporal constraints. The framework is based on the real-time scheduling theory and is mostly written for educational purposes [16].

```
thread implementation synchronizer.impl
        properties
        Dispatch_Protocol => perodic;
        Period => 100 ms;
        Deadline => 100 ms;
        Compute_Execution_Time => 50..60ms;
end synchronizer.impl;

thread implementation calalculating .impl
        properties
        Dispatch_Protocol => perodic;
        Period => 100 ms;
        Deadline => 100 ms;
        Compute_Execution_Time => 30ms..40ms;
end calalculating .impl;

thread implementation gps. position
        properties
        Dispatch_Protocol => perodic;
        Period => 40 ms;
        Deadline => 40 ms;
        Compute_Execution_Time => 30ms..40ms;
end gps. position ;

thread implementation HMI.setting
        properties
        Dispatch_Protocol => perodic;
        Period => 30 ms;
        Deadline => 30 ms;
        Compute_Execution_Time => 20ms..30ms;
end HMI.setting;
```

Listing 1.4. Setting of scheduling properties

Listing 1.4 shows a set of 4 periodic tasks (cal, pos, sync and setting) of TCU respectively, defined by the periods 100, 100, 40 and 30, the capacities 60, 40, 30 and 20, and the deadlines 100, 100, 40 and 30. These tasks are scheduled with a preemptive Rate Monotonic scheduler (the task with the lowest period is the task with the highest priority).

For a given task set, if a scheduling simulation displayed XML results in the Cheddar. One can find the concurrency cases or idle periods (see left of figure 8, comprise the software part and physical device part). People change the parameters directly and reload simulation; a feasible solution can be applied instead. After tuning, finally, the appropriate setting has displayed as in right of figure 8. According to this simulation result, people can correct the properties value in AADL, thereby ensure the correctness of system behavior timing properties.



```
Task name=cal          Period= 10; Capacity= 4; Deadline= 10; Start time= 0; Priority= 2; Cpu=cpuc
Task name=pos          Period= 4; Capacity= 3; Deadline= 4; Start time= 0; Priority= 1; Cpu=cpua
Task name=setting          Period= 3; Capacity= 2; Deadline= 3; Start time= 0; Priority= 1; Cpu=cpub
Task name=sync          Period= 10; Capacity= 3; Deadline= 10; Start time= 0; Priority= 1; Cpu=cpuc
Core Unit / Processor name=c1/cpua       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
Core Unit / Processor name=c3/cpuc       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
Core Unit / Processor name=c2/cpub       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
```

(a) Schedule 1 with idel time

```
Task name=cal          Period= 6; Capacity= 3; Deadline= 6; Start time= 0; Priority= 2; Cpu=cpuc
Task name=pos          Period= 4; Capacity= 3; Deadline= 4; Start time= 0; Priority= 1; Cpu=cpua
Task name=setting          Period= 3; Capacity= 2; Deadline= 3; Start time= 0; Priority= 1; Cpu=cpub
Task name=sync          Period= 6; Capacity= 4; Deadline= 6; Start time= 0; Priority= 1; Cpu=cpuc
Core Unit / Processor name=c1/cpua       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
Core Unit / Processor name=c3/cpuc       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
Core Unit / Processor name=c2/cpub       Protocol = RATE_MONOTONIC_PROTOCOL ; PREEMPTIVE
```

(b) Schedule 2 with compact time

Fig. 8. Simulation results of tasks schedule

6 Related work

We have presented our approach to extending SysML-based engineering framework Capella to AADL and analyzed the relationships among Arcadia and AADL models in different view at the metamodel level. Likewise, a considerable number of studies have been proposed on ”language extension, modeling languages integration and composable language components”. This section provides a brief introduction to these works.

The complexity of the development of CPS has been the significant problems which puzzle the developers. It is not only from the nature of problems but also from the develop languages. Elaasar et al. has discussed [6] about the limit of UML which exacerbate the complexity of development, and proposed an approach to reduce the complexity of UML tools by implementing and adapting the ISO 42010 standard on architecture description.

Efficient integration of different heterogeneous modeling languages is essential. Modeling language integration is onerous and requires in-depth conceptual and technical knowledge and effort. Traditional modeling language integration approaches require language engineers to compose monolithic language aggregates for a specific task or project. Adapting these aggregates to different contexts requires vast effort and makes these hardly reusable. Arne Haber et al [11] presented a method for the engineering of grammar-based language components that can be independently developed, are syntactically composable, and ultimately reusable.

In despite of existing a lot of studies on the combining SysML and AADL [4] or on the extending SysML with AADL [2]. Differ from the above studies, our approach dedicates to smoothly combine engineering platform Capella/Arcadia, AADL and its annex, and our approach can be easily applied to other languages through fine-tuning. In practice, one could design global system at a high level and then seamlessly refine the models within AADL and its annex for further analysis such as scheduling. In other words, our approach can properly extend Arcadia's design and analysis capabilities to AADL, while essentially keeping its independence.

An approach for translating UML/MARTE detailed design into AADL design has proposed by Brun et al. [3]. Their work focuses on the transformation of the thread execution and communication semantics and does not cover the transformation of the embedded system component, such as device parts. Similarly, in [25], Turki et al. proposed a methodology for mapping MARTE model elements to AADL component. They focus on the issues related to modeling architecture, and the syntactic differences between AADL and MARTE are well handled by the transformation rules provided by ATL tool, yet they did not consider issues related to the mapping of MARTE properties to AADL property. In [19], Ouni et al. presented an approach for transformation of Capella to AADL models target to cover the various levels of abstraction, they take into account the system behavior and the hardware/software mapping. However, the formal definition and rigorous syntactic of transformation rules are missed.

Behjati et al. describe how they combined SysML and AADL in [2] and provided a standard modeling language (in the form of the ExSAM profile) for specifying embedded systems at different abstraction levels. De Saqui-Sannes et al. [4] presented an MBE with TTool and AADL at the software level and demonstrated with the flight management system. Both of their works do not provide the description in a formal way.

In industrial domain applications, Suri et al [24] proposed a model-based approach for complex systems development by separating the behavior model and execution logic of the system. Moreover, they used UML based languages to model system behavior and connected the behavior models to external physical API of CPS. It focuses on providing a solution for the modularity and interoperability issues related to Industry 4.0 from a systems integration viewpoint.

S. Apel et al [1] also studied on different model driven methods for heterogenic systems for Electric vehicle. They have tried to evaluate how model-driven engineering (MDE) combined with generative frameworks can support the transfer from platform independent models to deployable solutions within the logistical domain.

The work of Kurtev [14] is used in the x-ray machine, it provided a family of domain-specific languages that integrate existing techniques from formal behavioral and time modeling. F. Scippacercola [21] have explored the application of model-driven engineering on the interlocking system (a subsystem of signaling system of the railway). They discussed how to reduce efforts and costs for development, verification, and validation in a critical system.

The modeling language scientists have proposed some specific methods to weave the models as well as metamodels formally such as [13], Degueule has proposed Melange, a language dedicated to merging languages [5], and similar works like [20]. However, the structural properties are not supported.

Compared with current studies, the approach proposed in this paper has the following features:

1. A proper subset of AADL has been chosen as the transformation target including functional software composition, execution platform. We use it to describe continuous behaviors of Cyber-Physical System.
2. All of the transformations is considered at metamodel level, and then a synthesized metamodel can be used to create concrete AADL models for further analysis.
3. Transformation rules are formally defined, and then it is readable by human and easier to verify the correctness of transformation.

7 Conclusions and future work

In this paper, we proposed a language-based design approach for combining different modeling design artifact (called views). At the beginning of this paper, we explicitly introduce the workflow of the proposed approach. Then we give the definition of syntax and semantics of our language. We selected system engineering methodology Arcadia (based on SysML) and architectural design language AADL as a vehicle for demonstrating the effectiveness of our approach and of model combination language for scheduling verification. We did so for two reasons. Firstly, the integrating of heterogeneous components and elaborate model integrity concept in system design are challenging problems while using numerous model language to describe different views of one system (or subsystem). Since our proposed language is generic enough to extract pertinent subsets of languages, if it works well for combining Arcadia and AADL, it should also work for the others, less demanding major modifications and extra cost of learning. Secondly, Enriching the functional design with scheduling ability can discover the conflicts in the early stage and improve the performance of CPSs in practice in a better way. Hence, Our language is competent for combining the composition of several heterogeneous artifacts (views) into a sound and consistent system model.

Especially, we give a formal description of the key modeling elements of Arcadia and AADL, respectively. Then we give some example of transformation rules which guide transforming from these Arcadia metamodels to AADL formally. Finally, a case study of train traction controlling system is used to demonstrate the transformation from engineering concerned design into an architectural refinement design which can be further analyzed by scheduling properties to find flaws of functional design.

Although our proposed language-based approach is effective and has been proven by many instances in practice, there are some drawbacks to use our approach: i) people have to spend times to learn the syntax of rules, and the writing of rule is error-prone. ii) the traceback function is not yet implemented automatically.

In our future work, we will try to build a graphic interface to write rules, and the writing errors of the rule can be detected. We also have to implement the traceback of simulation results, which is sketched in our workflow with the arrow in dotted line. The results must be used automatically by upstream modeling framework. To this end, we have to extract the critical information from cheddar outputting file and transform to an appended file of modeling tool which can be recognized by the tool and hint user in somehow. Secondly, we will study the transformation rules for more elements of Arcadia and also for comprehensive SysML elements, even for other UML-like profiles such as MARTE. At the same time, we will continue to explore the AADL and its annex to support more analysis and formal verification of system design. Besides, the safety-critical systems have become a trend in industrial files. We will study the extension of AADL with verification of safety properties with transformation methodology.

References

1. Apel, S., Mauch, M., Schau, V.: Model-driven engineering tool comparison for architectures within heterogenic systems for electric vehicle. In: 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 671–676 (Feb 2016)
2. Behjati, R., Yue, T., Nejati, S., Briand, L., Selic, B.: Extending SysML with AADL concepts for comprehensive system architecture modeling. In: European Conference on Modelling Foundations and Applications. pp. 236–252. Springer (2011)
3. Brun, M., Vergnaud, T., Faugere, M., Delatour, J.: From UML to AADL: an Explicit Execution Semantics Modelling with MARTE. In: ERTS 2008 (2008)
4. De Saqui-Sannes, P., Hugues, J.: Combining SysML and AADL for the design, validation and implementation of critical systems. In: ERTS2 2012 (2012)
5. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jezequel, J.M.: Melange: A meta-language for modular and reusable development of dsls. In: Conf on Software Language Engineering. pp. 25–36. ACM (2015)
6. Elaasar, M., Noyrit, F., Badreddin, O., Gérard, S.: Reducing uml modeling tool complexity with architectural contexts and viewpoints. In: MODELSWARD. pp. 129–138 (2018)
7. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Computer Languages, Systems & Structures **46**, 106–139 (Nov 2016)
8. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language. Addison-Wesley (2012)
9. Garlan, D.: Modeling challenges for cps systems. In: 2015 IEEE/ACM 1st International Workshop on Software Engineering for Smart Cyber-Physical Systems. pp. 1–1 (May 2015). https://doi.org/10.1109/SEsCPS.2015.8
10. Group, O.M.: OMG Systems Modeling Language (May 2017)
11. Haber, A., Look, M., Perez, A.N., Nazari, P.M.S., Rumpe, B., Vlkel, S., Wortmann, A.: Integration of heterogeneous modeling languages via extensible and composable language components. In: 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 19–31 (Feb 2015)
12. Harel, D., Rumpe, B.: Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff (Aug 2000)
13. Jezequel, J.M.: Model driven design and aspect weaving. Software and Systems Modeling **7**(2), 209–218 (2008)
14. Kurtev, I., Schuts, M., Hooman, J., Swagerman, D.J.: Integrating interface modeling and analysis in an industrial setting. In: MODELSWARD. pp. 345–352 (2017)

15. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing. pp. 363–369. IEEE (2008)
16. Marcé, L., Singhoff, F., Legrand, J., Nana, L.: Scheduling and Memory Requirements Analysis with AADL. In: SIGAda. pp. 1–10. ACM (2005)
17. McCracken, D.D., Reilly, E.D.: Backus-naur form (bnf) (2003)
18. OMG: OMG Unified Modeling Language (Apr 2015)
19. Ouni, B., Gaufillet, P., Jenn, E., Hugues, J.: Model Driven Engineering with Capella and AADL (2016)
20. Ramos, R., Barais, O., Jezequel, J.M.: Matching model-snippets. In: Conf on Model Driven Engineering Languages and Systems. pp. 121–135. Springer (2007)
21. Scippacercola, F., Pietrantuono, R., Russo, S., Zentai, A.: Model-driven engineering of a railway interlocking system. In: 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD). pp. 509–519 (Feb 2015)
22. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Cheddar - a flexible real time scheduling framework. SIGAda pp. 1–8 (2004)
23. Stahl, T., Voelter, M., Czarnecki, K.: Model-driven software development: technology, engineering, management. John Wiley Sons, Inc. (2006)
24. Suri, K., Cuccuru, A., Cadavid, J., Gérard, S., Gaaloul, W., Tata, S.: Model-based development of modular complex systems for accomplishing system integration for industry 4.0. In: MODELSWARD. pp. 487–495 (2017)
25. Turki, S., Senn, E., Blouin, D.: Mapping the MARTE UML profile to AADL. In: ACES-MB. pp. 11–20 (2010)
26. Wang, J., Wang, J.: A New Early Warning Method of Train Tracking Interval Based on CTC. IEEE Transactions on Intelligent Transportation Systems pp. 1–7
27. Zhao, H., Apvrille, L., Mallet, F.: Meta-models combination for reusing verification techniques. In: 7th International Conference on Model-Driven Engineering and Software Development. pp. 39–50. SCITEPRESS-Science and Technology Publications (2019)
28. Zhu, L., Zhang, Y., Ning, B., Jiang, H.: Train-ground communication in CBTC based on 802.11 b: Design and performance research. In: CMC'09. pp. 368–372. IEEE (2009)