About formal interpretation of architecture models

Denis Buzdalov, Alexey Khoroshilov
Institute for System Programming of Russian Academy of Sciences; email: {buzdalov, khoroshilov}@ispras.ru

Abstract

It is essential for nowadays architecture models to be able to be analyzed by automatic tools. This paper discusses what is needed for this and how robust interpretability can be achieved.

1 Introduction

This paper discusses issues of floating between abilities to express something complex in architecture models and abilities to formally interpret these models.

Generally, modelling languages are needed for both following usages:

- to *express* some ideas by some *author* (usually, a human) and
- to *perceive* these ideas by some particular *interpreter* (which can be a human or a machine).

These two usages of modelling languages contradict with each other. The more powerful a modelling language is, the easier to write models for author and the harder they are perceived by interpreters (in particular, the harder to implement a machine interpreter). Vice versa, the less powerful the modelling language, the easier to interpret it and the harder to use it by a model author.

If the intended interpreter of a modelling language is a machine, the language have to have formal semantics.

Sometimes modelling languages are used to solve a lot of tasks some of which are not automated yet. In such cases those parts that are intended to be used only by human may be left not so formalized. Generally this situation is normal but since models are tending to become more and more complicated, covering of analysis tasks by automation becomes more and more essential.

2 Formal interpretation

Every modelling language has bounds for set of domains where it is applicable. Domain defines aims and ways how such language would be used.

In particular, domain defines which characteristics are needed to be estimated by an architecture model. It defines precise enough semantics for each characteristic.

Characteristics example *For instance, we can consider such characteristics of architecture models like:*

- *latencies of data going though the system;*
- *power consumption of the whole system and its parts;*
- *what is the reaction of a system on particular stimulus end etc.*

Notice that some of such characteristics may be defined in some normative documents like international standards. Some other characteristics may be induced by particular not-standardized user definition. They can be set in user documents too.

AFDX in AADL *We can consider a document named* 'Recommendations of modelling AFDX nets in AADL models'. *This document is not a part of the AADL standard. But it definitely can define different domain-specific characteristics of AADL models: e.g. we can consider a formal interpretation with semantics of maximal and minimal AFDX switch buffers loading.*

This interpretation is not meaningful considering only AADL standard as a source of formal interpretations. But addition of user normative document widens set of interesting interpretations.

Talking about machine interpreters, those characteristics of architecture models need to be formalized. Formalization of getting machine-interpretable (i.e. formal) result with semantics of some particular characteristic (of an architecture model) is called *formal interpretation* of the architecture model.

There is a common issue for formalizations: formal interpretation may be or may be not *adequate* to the original (informal) semantics. Determination of adequacy of particular formal interpretation to its intended semantics (induced e.g. by domain of usage) is a task that have to be solved by human.

3 Underlying formal models

Lots of formal interpretations of some particular architecture modelling language need to perform a lot of more or less same actions for rearranging syntactic representation of models according to theirs semantics.

We would call formal objects that turn out after these actions *underlying formal models*. We can consider *formal objects* of a model (possibly, with some properties) and *relations* between them.

To make a machine to work with underlying models, all objects and relations have to have precise semantics. It means that set of operations and actions that can be done with such objects have to be formally defined.

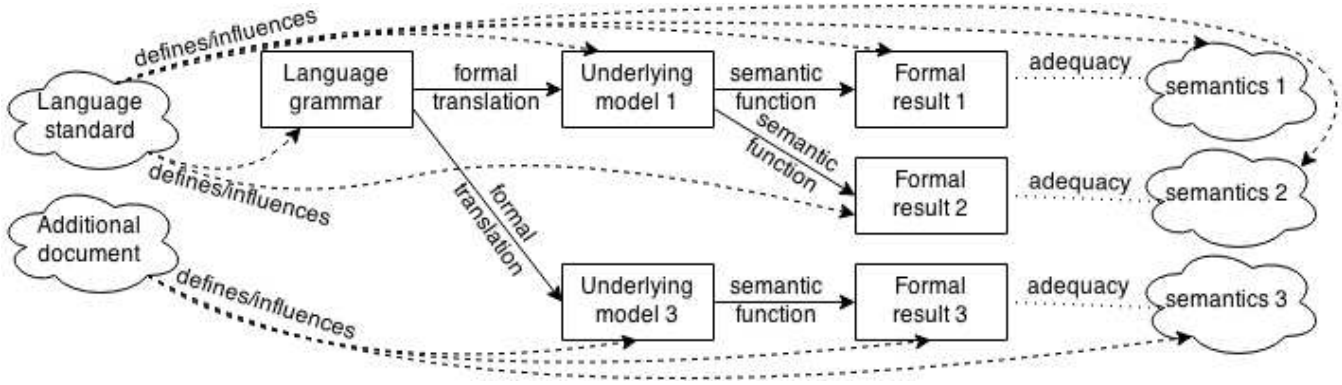

Figure 1: General view of formal interpretations

3.1 Different models

Notice, that single language may have different underlying models (which require different ways to get those underlying models for a single architecture model).

We can consider different relations between underlying models of a single modelling language. As an example, we will consider whether such models contradict with each other or not in some cases.

Let us consider two important different underlying model types.

Domain models We can call *domain model* an underlying model of a language which contains info for formal interpretation which have results that are meaningful in some particular domain.

Type models Domain model objects can have *types*. It means that some relation (which can be called 'is type of') exists between objects of domain model and some particular set of types.

Actually, we can notice that types also can have relations between each other. Since types can also be considered as formal objects, types with theirs relations can be organized into formal *types model*.

So, we can consider some joint formal model containing all objects and relations of both domain and types model and mentioned above 'is type of' relation. This model is also an underlying model of the same language.

AADL instance model *If the AADL* instance model *was formalized enough, it could be considered as the typed domain model for AADL.*

3.2 Formal translation

Both modelling language grammar and underlying model are formal. Underlying model is built upon statements of the modelling language.

This allows us to define *formal translation* of original modelling language to underlying formal model. Semantics of the language constructs have to be used during such translation.

AADL component instances *For example, AADL contains rules of filling properties of 'component instances' objects.*

We can consider this process as a part of formal translation of AADL text to the instance model (if it was formal).

AADL connections *Consider an example what objects could be in some underlying model.*

AADL grammar defines `connection` *constructs. They mean data and control transmission between model components. They can be contained in different components (which in their order can lay in different places in the containment hierarchy).*

The language standard defines semantically meaningful notion of 'semantic connections'. They are built upon a set of linked `connection` *instances with respect of components structure.*

The standard defines how to get semantic connections by the model definition. If this translation is formalized, it can be used as a part of formal translation to underlying model containing objects of type **SEMANTIC CONNECTION**.

Such objects are useful for machine interpreters that are working with semantics connections. If such objects exist in an underlying model, they do not have to perform the same job of building of similar formal objects upon simpler ones (e.g. connection instances).

3.3 Semantic functions

To interpret formally translated underlying model, we need to define first a *formal result* which we want to reach. We need to define the type and semantics of this result.

Functions that map an underlying model (with, probably, some additional information) to semantically meaningful formal result are called *semantic functions*.

Semantic functions finish formal interpretation process. It is one of the most important part of the formal interpretation because underlying models are intended to be used by lots of semantic functions and are meant to be adequate. It means

that semantic functions determine adequacy of formal interpretation it participates in.

Now let us consider several important examples of semantic functions.

3.3.1 Internal consistency

One particular and important case of semantic functions are functions of *internal model consistency*. They are needed because not all relations in the same underlying model are independent.

Internal consistency rules are functions that map a single underlying model into a boolean result.

AADL consistency rules Legality rules *and* semantics *sections (though the prism of instance model building rules) give consistency rules for instance models.*

For example, legality rule saying that `process` *components cannot contain other* `process` *components would transform to the consistency checking function defined on a formalized instance model in some way like*

$$\forall \text{PROC1} : \textbf{PROCESS}, \text{PROC2} : \textbf{PROCESS} \bullet \neg(\text{PROC1 `contains'} \text{PROC2})$$

Also, we can consider a rule induced by the semantics of properties: e.g. no object can have `Compute_Deadline` *property greater than* `Deadline` *because the first property means a period of time that is a part of period of time determined by the second one.*

3.3.2 Latency analysis

Latency of data and control passing though the system is a one of most important characteristics of real-time systems.

Semantic connection latency *We can consider a semantic function for formalized AADL instance model returning bounds of latency for each semantics connection using information of latencies in execution platform (considering speed of buses, execution schedules and etc.).*

3.3.3 Execution traces

If we are modelling systems that have some particular *behaviour*, we definitely can think of semantic functions aware of *traces of execution* of such systems.

One useful and not very complicated kind of such functions is a *execution trace checker* function. Besides the model, it requires additionally an execution trace (with outer world stimuluses, if needed). Formal result of such kind of functions is a boolean value with semantics of whether given trace can appear during the execution of the model.

This kind of semantic functions can be used both for deterministic and non-deterministic models. Other kinds of similar functions can be considered, e.g. those which generate model execution trace. They can require additional information.

4 Expressiveness and interpretability

4.1 Underexpressiveness

We can consider some particular (informal) interpretation semantics that is, for example, induced by the domain.

Consider the case when no formal interpretation that conforms this interpretation semantics, exists. In this case we say that the language is *underexpressive*.

Consider another situation when some particular underlying formal model is considered and no formal interpretation using this formal model exists for the informal interpretations semantics. In this case this underlying model is called *underexpressive*.

In other words, we cannot express something that we need to express using underexpressive language or its underlying model.

Processor component example *AADL legality rules set that* `processor` *components cannot contain other processor components. Considering* 'can contain' *relation of types, this rule can be formalized in way that* **PROCESSOR** *does not have relation* 'can contain' *with itself.*

Considering rules of components instantiation and subcomponents containment, the formalized rule above means that no AADL model with nested `processor` *components can satisfy instance models consistency rules.*

Since only `processor` *components can contain properties of hardware processors, we are not able to represent nested structure of hardware processors. Thus we are unable to represent multicore systems using* `processor` *components.*

Finally it means that no formal interpretations that are defined for multicore systems exists for AADL. This example reflects the situation of the AADL standard of version 2.1.

4.2 Overexpressiveness

We can consider another case, somewhat inverse to the previous one.

We call language or model *overexpressive* if it does not have *a single adequate* formal interpretation for some particular semantics.

Two reasons can lead to it:

1. no adequate formal interpretation exists for this semantics;
2. more than one contradicting and adequate formal interpretations exist for the semantics.

In other words, overexpressiveness is a situation when it is possible to express more than it can be formally interpreted.

Processors misinterpretation *Consider that we decided to solve the mentioned above underexpressiveness problem in a pretty simple way: we allowed objects of type* **PROCESSOR** *to contain other objects of type* **PROCESSOR**. *This decision influences at least on underlying model and formal translation rules.*

Consider a semantic function which maps an underlying model to count of operating systems in it. We can think of interpretation that is lying on the current semantics of instance type **PROCESSOR** *which incorporates both hardware chip and operating system model.*

When we look at formal interpretation that uses this semantic function we will find that this interpretation is not adequate to what we want to understand by several nested `processor` *components because three operating systems in a model with two-core CPU is not a thing meant to be expressed.*

We can imagine another semantic function that seems to be more adequate: for example, it defines that if we have a couple of objects CONTAINING PROC *and* NESTED PROC *of type* **PROCESSOR** *that have* 'contains' *relation between them, it means that* NESTED PROC *is considered to not to represent an operating system.*

Alas, this semantic function contradict with the current (both formal and informal) standard interpretations.

Sometimes the reason of second variant of overexpressiveness is the lack of semantics in normative documents, which allows to imagine lots of pairwise contradicting formal interpretations that conform to the original semantics requirements.

Connection binding *We can consider lack of semantics at the example AADL property called* `Actual_Connection_Binding`.

This property is a list of references to different execution platform components.

First of all, formal interpretations can differ on attitude to the order of items of this property association. Situation can become spicy when we remember that different ports and connections (that can have `Actual_Connection_Binding` *relation) can be bidirectional.*

But even if we would stop on the interpretation of this property as a set of references, we can have lots of different interpretations of a single semantic property.

For example, one interpretation can assume that all *referenced execution platform components have to be used for transmission of each message through bound port or connection. Another interpretation can think that* only *those components that are in the binding list can be used for transmission. Also, interpretation assuming* no strict obligations *can exist.*

Also, another point where interpretations can differ is how branching routes are managed. The same issues (whether all or only some branches have to be covered for each sent message) can be considered.

4.3 Underlying model refinement

To manage overexpressiveness not running into underexpressiveness, we need to refine the objects set and to change relations appropriately. Obviously, such changes lead to appropriate changes of formal translations.

In particular, we can consider a type **SUBJ** having a relation 'rel' with some other type **OBJ**. If this relation is overexpressive, i.e. we cannot interpret all such relations but only some of them, we can try to split the type **SUBJ** to two: **SUBJ+** and **SUBJ-**, where **SUBJ+** has the relation 'rel' with **OBJ** and **SUBJ-** has not.

This allows to have more granulated relations between objects in a corresponding typed domain model.

Overexpressiveness problems management *Now we can think of how we can manage overexpressiveness by the underlying model refinement, in particular by model objects separation.*

The problem is that there are too many operating systems in the formal model, more than we want to express. We need to refine the formal model to allow to express only what we can interpret leaving ability to express enough.

To do so, we can distinguish the following three types in the instance level: **HW PROCESSOR**, **HW CORE** *and* **OS**. *Only* **HW PROCESSOR** *has relation* 'can contain' *with* **HW CORE**. *To express that operating systems run on a processor and manages its cores, we need a relation* 'can run on' *between* **OS** *and* **HW PROCESSOR** *instance types. We should not allow an operating system to run on a single core of a processor. It is done by absence of* 'can run on' *relation between* **OS** *and* **HW CORE**.

We need the formal translation rules to be refined. For example, all upper-level `processor` *components can be translated to pair of objects of type* **HW PROCESSOR** *and* **OS** *and all nested* `processor` *components can be translated to* **HW CORE** *objects.*

5 Related works

There are several works related to the problem of formal semantics of modelling languages like AADL. Great classification of such works is given in [1].

Generally, it points to different kinds of behavioural semantics definitions. Some of them do not rely on additional specification of components behaviour [2,3,4]. Some approaches rely on non-standard behaviour specifications [5,6,7]. Other approaches define formal semantics for standardized behaviour specification [1,8].

This work is mpre general and differs from those works mainly in two points:

1. this work does not offer a concrete semantics for particular language standard parts; we propose a discipline of language standards writing which consequence is a formal semantics definition in the language standard; this frees the language users and instruments writers from a need to reinvent (possibly, contradictory) formal semantics for parts of the standard;

2. this work touches formal semantics not only of behavioural part of modelling languages; in the paper we try to offer a uniform way of working with different semantic aspects of a modelling language.

6 Conclusion

To analyse architecture models by a machine, domain-induced *formal interpretations* have to be defined. Underlying models can be used to reduce effort of formalization and implementation of such interpretations.

All transformations used in such interpretations are formal. These formal transformations should be checked for *adequacy* to informal semantics which is defined by normative documents.

AADL standard contains a lot of information needed for formal interpretation of AADL models. But a lot of things are left not formalized enough or even missing. Some examples were discussed above.

Way of thinking a language standard as a place of definition of formal interpretations seems to be the way for language to become formally interpretable that is easily and robustly analyzable by a machine.

The future work should focus on formalization of user interpretations, in particular, usage of multiple underlying models induced by multiple normative documents. This allows to build analyzers for user-defined characteristics which are formalized and can be implemented in different instruments with the same precise semantics.

Generally, such approach makes architecture models analysis robust and trustful.

References

[1] P. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral aadl models in real-time maude," in *Formal Techniques for Distributed Systems* (J. Hatcliff and E. Zucca, eds.), vol. 6117 of *Lecture Notes in Computer Science*, pp. 47–62, Springer Berlin Heidelberg, 2010.

[2] O. Sokolsky, I. Lee, and D. Clarke, "Process-algebraic interpretation of aadl models," in *Reliable Software Technologies – Ada-Europe 2009* (F. Kordon and Y. Kermarrec, eds.), vol. 5570 of *Lecture Notes in Computer Science*, pp. 222–236, Springer Berlin Heidelberg, 2009.

[3] S. Gui, L. Luo, Y. Li, and L. Wang, "Formal schedulability analysis and simulation for aadl," in *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, pp. 429–435, July 2008.

[4] P. Dissaux and O. Marc, "Executable AADL: Real-time simulation of AADL models," in *ACVI 2014 – Architecture Centric Virtual Integration Workshop Proceedings*, pp. 59–68, 2014.

[5] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens, "Virtual execution of aadl models via a translation into synchronous programs," in *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, (New York, NY, USA), pp. 134–143, ACM, 2007.

[6] T. Abdoul, J. Champeau, P. Dhaussy, P.-Y. Pillain, and J. Roger, "Aadl execution semantics transformation for formal verification," in *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pp. 263–268, March 2008.

[7] D. Buzdalov and A. Khoroshilov, "A discrete-event simulator for early validation of avionics systems," in *ACVI 2014 – Architecture Centric Virtual Integration Workshop Proceedings*, pp. 28–38, 2014.

[8] Z. Yang, K. Hu, D. Ma, and L. Pi, "Towards a formal semantics for the aadl behavior annex," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 1166–1171, April 2009.