Integrating AADL within a multi-domain modeling framework

Ivano Malavolta, Henry Muccini, and Patrizio Pelliccione
Dipartimento di Informatica, Università dell'Aquila,
Via Vetoio, 67100 L'Aquila,
ivano.malavolta@univaq.it
{henry.muccini | patrizio.pelliccione}@di.univaq.it

Abstract

DUALLY *is a framework that allows architectural languages interoperability through automated model transformation techniques. Any transformation among ADLs is defined in* **DUALLY** *by passing through* A_0 *that is an extensible semantic core set of modeling elements. In this paper we describe the integration of AADL and the corresponding OSATE tool-set in* **DUALLY**. *Once AADL is hooked to* A_0*, it is automatically integrated with the network of languages already integrated in* **DUALLY**. *In particular, we show how it is possible, in an easy way, to obtain a UML specification and to model check AADL and behavioral annex specifications through LTSA.*

1. Introduction

Increasingly, Architecture Description Languages (ADLs) [1] are defined by stakeholder concerns [2]. The large amount of possibly different stakeholder (e.g., a developer, a domain expert, a maintainer) implies a variety of different needs, requirements and domain specific information to be incorporated into an architectural language. Obviously, it is impossible to capture all such domain concerns with a single, narrowly focused notation and building a "universal" notation for modeling software architectures is impractical. This is one of the major causes of the flourishing of different ADLs.

While a unique language for modeling all such concerns is not reasonable, the software architecture community has created a quite clear boundary between those core architectural concepts shared among most software architecture languages, and other modeling elements that are domain- or analysis- specific. This observation is one of the leitmotif of **DUALLY** [3], an automated framework that allows architectural languages and tools interoperability. Given any number of architectural languages and tools, **DUALLY** allows interoperability among them through automated model transformation techniques. Any transformation among ADLs is defined in **DUALLY** by passing through what we refer to as A_0, a semantic core of architectural concepts, providing the infrastructure upon which to construct semantic relations among different ADLs. It acts as a bridge among the different architectural languages to be related together. Extensibility mechanisms are provided by **DUALLY** in order to augment A_0 with domain specific concerns. For instance, A_0 could be extended with performance concerns by obtaining an A_{perf} or it can be extended with real-time concerns by obtaining an A_{RT}. Therefore, A_0 is the root of a hierarchy tree that allows **DUALLY** to support the horizontal abstraction [4], i.e., an abstraction that takes place at a same level of definition and emphasizes certain system concerns or complementary viewpoints.

In this paper we analyze the feasibility in integrating AADL [5] and the corresponding OSATE tool-set in **DUALLY** following a multi-domain approach. The integration is performed in a *scalable*, *time-saving* and *effective* way. In fact, the number of transformations is linear and adding AADL means just to relate it to A_0 or to one of its extensions (without the need to relate AADL with each ADL previously integrated). This process is time-saving since we need only to relate AADL to A_0. This process is effective since the software architect neither needs to be skilled in model transformation techniques nor needs to know all the notations within **DUALLY**. We show how AADL (and its behavioral annex), once integrated in **DUALLY**, can be transformed into both a UML profile for SA modeling [6] and Darwin/FSP [7]. Thanks to the integration with Darwin/FSP we can, in an easy way, model check AADL behavioral specification through the Darwin/FSP LTSA tool [7].

The paper is structured as follows: Section 2 describes how **DUALLY** can be used as a multi-domain modeling framework. Section 3 presents the integration of AADL in **DUALLY** and Section 4 shows how AADL models can be automatically transformed to UML and Darwin/FSP models. In Section 5 we present related work, while Section 6 concludes the paper highlighting future research directions.

2. DUALLY as a multi-domain modeling framework

DUALLY [3] allows different formal ADLs and/or UML-based languages and tools for software architecture modeling to interoperate. The interoperability is implemented via model transformation and passing through A_0, a semantic core of architectural concepts, acting as a central pillar of the

model transformation network. A_0 becomes the staging point for the complete and consistent migration for architectural information across any number of description technologies already integrated within **DUALLY** (in the form of meta-models or profiles), by means of the process we call **DUALLY**zation. A_0 has been defined as general as possible to ensure that **DUALLY** is able to potentially represent and support any kind of architectural representation (i.e., formal ADLs or UML-based languages). A_0 is a MOF compliant meta-model whose main elements are the concept of component, connector, interface, channel, architectural type, property and group. Please refer to [3] for further details about A_0 and its meta-model.

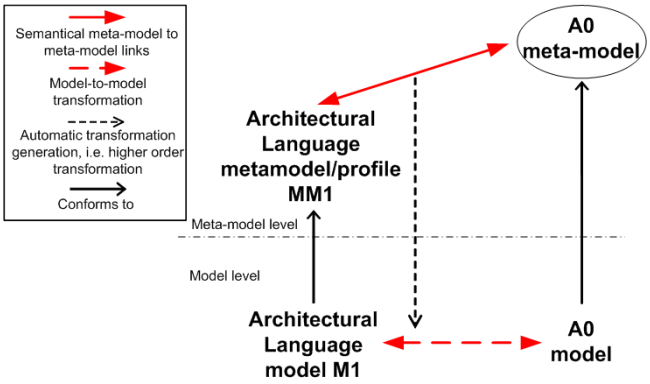

Figure 1. **DUALLY** conceptual view

As shown in Figure 1, the semantic mappings (and the corresponding generated transformation) relates an architectural language representation MM1 (via a meta-model/profile) to the A_0 meta-model (as well as M1 to A_0). Transformations are defined between a meta-model/profile and the A_0 meta-model. Model-to-model transformations are then automatically instantiated by executing a higher-order transformation, thus providing the possibility to automatically reflect modifications made on a model designed with a language to one or even all of the other languages connected with **DUALLY**. The semantic correctness of the generated transformations can be checked by providing properties that must hold while passing from the source to the target models [3].

A_0 has been defined as extensible in order to give the possibility to software architects to customize it for each specific domain. These kinds of extensions are realized by means of the inheritance mechanism. Each element of the A_0 meta-model can be extended.

As shown in Figure 2, ADLs can be "hooked" to the **DUALLY** framework at each level of specialization of A_0. We call **DUALLY**zed an ADL hooked with **DUALLY**. The *level 0* is A_0 and a generic ADL, e.g., ADL_y in figure, can be **DUALLY**zed directly to A_0. Figure 2 shows also ADL_x and ADL_z **DUALLY**zed at *level 1* and *level 2*, respectively. As previously seen, each transformation between two ADLs is performed always through A_0 or one of its extensions.

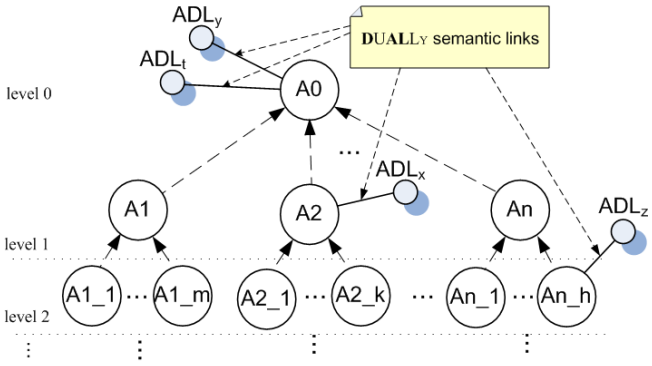

Figure 2. Hierarchy of domain specific extensions of A_0

If two ADLs are hooked at the same A', where A' is A_0 or an extension of A_0, it is clear that the transformation will be defined on elements of A'. Whereas, if two ADLs are hooked on two different versions of A_0, A'' and A''' respectively, the transformation will be based on elements of A' that is the first common "ancestor" of A'' and A'''. Thus, typically there exists a set of elements that cannot be translated from one ADL to a different one. The dimension of this set depends on the similarity of the pair of ADLs. The existence of these elements is unavoidable. In fact ADLs could be specific of two different domains and could contain domain specific aspects. However, **DUALLY** provides synchronization mechanisms such that changes made on a specific (generated) model can be propagated back to the others when closing the round-trip journey.

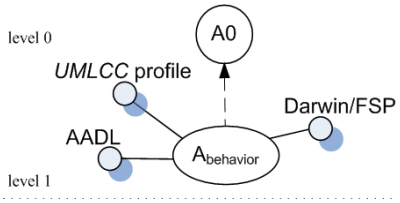

Figure 3. Hierarchy of domain extensions of A_0 instantiated on the illustrative example

In the context of what illustrated in Figure 2, this paper describes an attempt to realize an instance of the hierarchy, by considering the **DUALLY**zation between AADL and its behavioral annex, Darwin/FSP, and the UMLCC profile for software architecture modeling, and by passing through a common extension named $A_{behavior}$ (as shown in Figure 3).

3. Integrating AADL in DUALLY

This section presents the mappings between AADL (along with its behavior annex) and the A_0 meta-model. A mapping refers to semantic links among meta-models elements, and describes a transformation relationship between (AADL and A_0) architectural concepts and vice versa. The mapping is

implemented via a *weaving model* and its logic will be reflected upon the automatically generated transformations at the modeling level. Due to space limitations, we focus on the most relevant elements of AADL and their counterparts in the A_0 meta-model only.

3.1. Mapping core AADL concepts.

AADL components can be divided into software, hardware and system architecture. Since the concepts of A_0 focus on software architecture and this work is meant to abstract from AADL low-level specificity, we restrict our experimentation only on software and system AADL categories. Nevertheless, we had to pay special attention to the concept of AADL Device; in [8], a device is defined as a generic entity "that interfaces with the external environment of an application system" and that can be considered as both a software and an execution platform component. So, in order to complete our approach and cover all possible software entities of AADL, we consider also the concept of device during the mapping process.

Software component implementations. Each software component meta-class (namely AADL *ProcessImpl*, *ThreadImpl*, *ThreadGroupImpl* and *DataImpl*) is mapped to the generic A_0 *SAcomponent*. The promotion of these kinds of components to a generic entity is one of the means by which we manage to abstract from AADL specificity. However, this may cause a round-trip problem, i.e. while transforming an A_0 model into an AADL one, an SAcomponent can be transformed either into a process, a thread or some other AADL component; this problem could be avoided by extending the generated transformations so that model synchronization is performed during the round-trip, instead of creating a new AADL model. We are currently investigating this and other similar issues. Also *Threadgroup* is mapped to an *SAcomponent* because in AADL it natively serves to logically collect threads into a single, coarse-grained entity. Both a *ThreadGroup* and its internal *threads* are thus grouped into a single A_0 component.

Software component types. AADL software component types are mapped to *SAtype* in A_0, along with the related properties and features (whose mappings will be explained later in this section). Thus, AADL *ProcessType*, *ThreadType*, *ThreadGroupType* and *DataType* are mapped to A_0 *SAtype*. In case the source AADL concept can contain nested AADL entities, the target A_0 element is *SAstructuredType*. The only difference between an *SAtype* and an *SAstructuredType* is that it is allowed to define a subarchitecture within the latter one. Component types can be hierarchically structured in AADL through the "extend" structural reference.

System. In the specification of AADL, a system component is defined as an assembly of software and execution platform components. From an abstract point of view, it can be considered as a generic, coarse-grained component. Consequently, we mapped *SystemImpl* and *SystemType* to A_0's *SAcomponent* and *SAstructuredType*, respectively.

Device. An AADL device can be considered as a low-level software component executing on a processor that accesses the physical device. A device represents also the software that access and manage the underlying physical device. In the mapping process we linked AADL *DeviceImpl* and *DeviceType* meta-classes to A_0's *SAcomponent* and *SAtype*.

Features. An AADL feature (i.e., *Port*, *PortGroup*, *Parameter*, *Subprogram* and *ComponentAccess*) is part of a component type and represents its point of interaction with other components of the system. Based on this definition of AADL features, we mapped them to A_0 *SAinterface*. Moreover, ports and port groups directions are linked to the "direction" attribute of the target A_0 *SAinterface*, while other auxiliary attributes (like properties) are related to the corresponding A_0 properties. The "direction" attribute of an AADL *ComponentAccess* can be either $provided$ or $required$. In the former case the direction of the corresponding *SAinterface* is out, while in the latter one is in.

Connections. AADL port connections are mapped to *SAchannel* in A_0 if the connected elements are not contained within each other. Another mapping links AADL port connection and *SAbinding*. It is guarded with an OCL condition that is evaluated true if one of the connected elements is nested within the other one. So, similarly to the mapping between component implementations, this mapping links many different kinds of connections (e.g., *DataConnection*, *EventConnection*, *EventDataConnection*) into a single, generic A_0 entity reaching a higher level of abstraction. If the source AADL port connection is a DataConnection, then its "connectionTiming" attribute is mapped into an A_0 Property initialized with the corresponding value (i.e., delayed or immediate). *ParameterConnection*, *DataAccessConnection* and *SubprogramCall* are linked to either *SAchannel* or *SAbinding* in the same manner of port connections.

Properties. AADL *Property* is mapped to A_0 *Property*, the corresponding "value" feature is also linked to the *PropertyValue* meta-class in the A_0 meta-model. The types of AADL properties are also mapped to either an A_0 primitive type or a predefined A_0 *SAtype*, accordingly to the type of the source AADL property. Bridging the concept of property set to A_0 consisted in linking the *PropertySet* meta-class of AADL to A_0 Group and setting the "members" reference of the latter to the property declarations contained by the AADL property set.

Package. An AADL *Package* is defined as a construct to organize collections of component declarations into separate units. In the A_0 meta-model the concept of Group is semantically close to that of AADL package, so we mapped it (either it is public or private) to A_0 Group. This mapping helps producing A_0 models with different namespaces and designed in a modular way.

Modes. Modes and mode transitions are managed as well and, since their concepts intersect with that of behavior, their mapping is described in the following section.

3.2. Mapping behavioral concepts

The AADL behavioral annex is composed of three main sections: states, composite states and transitions declaration. The AADL behavioral annex contains also sections to declare state variables, their initialization and auxiliary connections to component ports. They are not part of this presentation because their semantics is not represented in A_0. Therefore, such elements are not mapped to any member of the A_0 meta-model.

The AADL *BehaviorAnnex* meta-class is mapped to a *StateDiagram* in A_0 and the contained references to states and transitions are mapped to the *ownedState* and *ownedTransitions* references, respectively.

AADL *State* and *CompositeState* are both mapped to A_0 *State*. In case the source AADL element is a composite state, there is also a link to an inner A_0 *StateDiagram* and the references to its contained elements is mapped as well. However the concept of history in a composite state is ignored because there is no semantic counterpart in A_0. An AADL state corresponds by default to a *State* in A_0. However, if the type of the AADL state is *initial*, then the mapping refers to *InitialState* in A_0, and if it is *complete* or *return*, the mapping refers to *FinalState*.

Transition in AADL corresponds to A_0 *Transition* and the condition of the guard is mapped to the "guard" attribute in A_0. The label attribute in the A_0 transition corresponds to the action of the AADL transition and both source and destination state attributes are mapped to the corresponding source and target attributes of A_0's state. This will help in preserving the topology of the AADL behavioral specification into A_0 state diagrams. Real-time aspects like delays, computations or timeouts are not considered during the mapping process so that they will be abstracted out while passing to A_0 models.

Since AADL modes are considered states in the behavioral annex, then the *Mode* meta-class is also mapped to *State* A_0. Similarly, the *ModeTransition* meta-class is mapped to A_0 *Transition*.

It is important to note that elements that cannot be mapped are stored by the **DUALLY** framework so that these parts of the model can be retrieved when coming back to the source notation in a round-trip scenario. In fact, it is unacceptable to use a transformation technique supporting the horizontal abstraction [4] that loses the parts of the model that cannot be translated from one ADL to another.

3.3. Model transformations generation

Once the semantic links between AADL and A_0 meta-models have been designed and implemented via a weaving model, model-to-model transformations can be automatically generated. We generate two transformations from the weaving model: (i) $Aadl2A_0$ and (ii) A_02Aadl. The former takes as input an AADL model and produces an A_0 one, while the latter performs the opposite task. In the following we will focus on the generation (and subsequent execution, see Section 4) of $Aadl2A_0$ only, presenting how **DUALLY** can be exploited to either raise AADL's level of abstraction (via UML) or formally verify it (via Darwin/FSP). The logic of the $Aadl2A_0$ reflects that of the the weaving models.

However, there are some concepts peculiar of the AADL language whose mapping has to be handled by advanced constructs in the $Aadl2A_0$ model transformation; thus we need to manually refine $Aadl2A_0$ in order to correctly preserve their semantics also in A_0. Each peculiar AADL concept and a high-level description of how we handle it in the model transformation are described below:

Inheritance between implementations: AADL allows the definition of hierarchies also at the implementation level. In this case all the features, properties and flows of a super-component are also implicitly defined in its sub-components. This creates a semantic mismatch between AADL and A_0 because in A_0 only hierarchy among type elements (e.g., *SAtype*) are allowed. To overcome this issue, we instructed the $Aadl2A_0$ transformation in order to "simulate" AADL implementations hierarchy so that if an AADL implementation component C_1 extends another component C_2, then in A_0 all the features and properties of C_2 are replicated into the component C_1. The various connections to and from C_2 are replicated as well into C_1.

Behavior communication actions: one of the aim of our work is to verify AADL models via LTSA. In FSP two transitions synchronize if they have exactly the same action name (e.g., p). Whereas, in AADL, transitions synchronize if they are coupled by the $p!$ (sender) and $p?$ (receiver) notation. The $Aadl2A_0$ model transformation is modified so that for each couple of actions, the corresponding action names are stripped to p.

4. Illustrative Example

To illustrate the execution of the **DUALLY** transformations we decided to consider a Flight Control System (FCS, taken from [9]) and to model its behavior through the OSATE AADL editor. This model mainly focuses on the autopilot and status display system. From a high-level perspective, it is composed of a LAN network connecting two main subsystems: a NAP_system that comprises a network of sensors and actuators managed by a navigation controller and an HCI_system that represents the Human Control Interface (HCI) that manages the moving map display and takes input from the pilot for autopilot parameters. Due to space limitation, we present the HCI_system only; its internal structure is represented in Figure 4.

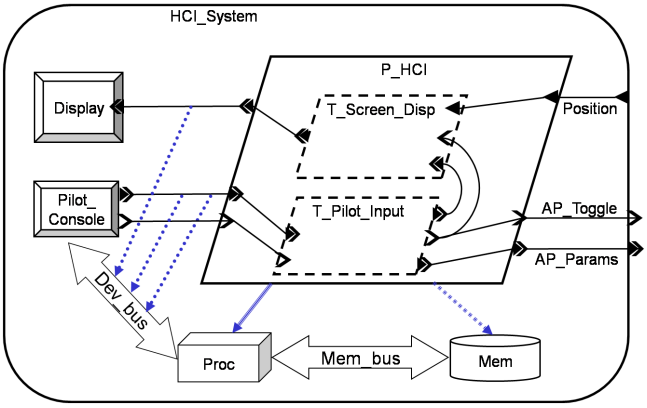

Figure 4. AADL model of the HCI system.

The HCI_System has a standard computing architecture: there are a processor and a memory component bound by a bus and devices connected to the processor via a dedicated bus. *Display* shows the moving map, and the status of the aircraft, while *Pilot_Console* is used to set the autopilot configuration and turn on/off the autopilot system through a toggle switch. The software part of the system is composed of a main process (P_HCI) containing two concurrent threads:

1) T_Screen_Disp: periodically updates the display device with the current position of the aircraft; it is also triggered if the pilot either inputs configuration data or switches on/off the system;
2) T_Pilot_Input: a background thread that waits for the pilot to input a new configuration or hits the toggle switch and transmit this information to T_Screen_Disp and to the NAP_system. If the system is off, the configuration is sent also to T_Screen_Disp only. A sketch of its internal behavior is shown in the following listing.

```
thread implementation T_Pilot_Input.PowerPC_G4
  properties
    Dispatch_Protocol => Background;
  annex behavior_specification {**
    state variables
      aux : Nav_Types::Position.GPS;
    states
      s_Off : initial complete state;
      s_On : complete state;
      s1, s2, s3, s4, s5 : state;
    transitions
      s_Off -[AP_Position_In?]-> s2;
      s2 -[]-> s_Off {AP_Position_Out_Disp!;};
      s_Off -[AP_Toggle_In?]-> s1;
      s1 -[]-> s_On {AP_Toggle_Out!;};
      s_On -[AP_Toggle_In?]-> s3;
      s3 -[]-> s_Off {AP_Toggle_Out!;};
      s_On -[AP_Position_In?(aux)]-> s4;
      s4 -[]-> s5 {AP_Position_Out_Disp!(aux);};
      s5 -[]-> s_On {AP_Position_Out_Nav!(aux);};
  **};
  ...
end Pilot_Input_Thread.PowerPC_G4;
```

Listing 1. Behavior of T_Pilot_Input

The whole HCI_System is composed of 10 hardware/software components, 21 ports (event, data and event/data), 19 connections (port connections and bindings). The behavioral descriptions comprise 14 states and 16 transitions.

Applying the $Aadl2A_0$ transformation introduced in Section 3.3 we obtain the A_0 representation of the HCI_System containing 6 SAcomponents (hardware ones have been abstracted), 21 SAinterfaces and 14 connections. The behavioral part contains the same number of elements as the AADL one.

From now on, we can exploit the transformations generated from past **DUALLY**zations in order to obtain models conforming to other notations. Our intention is to produce both a UML model and a Darwin/FSP specification of the HCI_System (as pointed out before, UMLCC and Darwin/FSP are already integrated in **DUALLY**). In the case of UMLCC, we execute the corresponding $A_02UmlCC$; a component diagram (represented in Figure 5) and a set of state machines are generated.

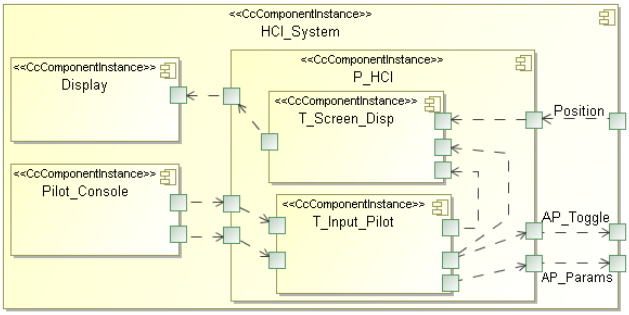

Figure 5. UML model of the HCI_System.

At this point we reconsider the A_0 model of HCI_System and execute the $A_02DarwinFSP$ transformation to produce its Darwin/FSP counterpart. The output specification is composed of two sub-models: the first one conforms to Darwin and describes the static configuration of the software components, it is conceptually similar to the UMLCC diagram in Figure 5; the second one is an FSP model and can be imported into the corresponding LTSA tool. Figure 6 graphically represents the behavior of T_Pilot_Input.

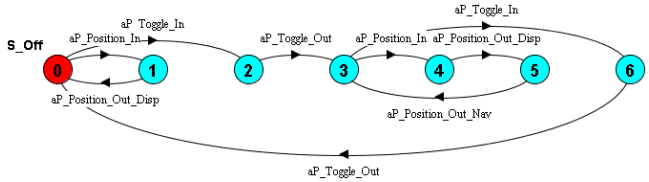

Figure 6. FSP specification of T_Pilot_Input.

One of the key features of LTSA is that it allows to compose the various behavioral description in order to analyze the system as a whole. So, the HCI_System resulted in a

main labeled transition system composed of 126 states and 459 transitions. In addition, transforming an the HCI_System model from AADL to FSP allowed us to check if the composed behaviors are in a deadlock. The result is negative, i.e. there are no deadlocks in the initial AADL specification. Moreover, customized properties may be checked on the produced FSP specification, like reachability analysis. LTSA provides also a facility to simulate the system, this helps to get an interactive idea on its functioning.

5. Related Work

Related work mainly regards automatic derivation of model transformations and semantic integration. The authors of [10] present ModelCVS, in which semantic links are defined between ontologies, that will serve as a basis for the generation of model transformations. **DUALLY** differs since it implies the A_0-centered star topology and the preliminary step of meta-model lifting is not performed.

The role of **DUALLY**'s A_0 is similar to the Klaper language in the field of performability. Grassi et al. in [11] propose the Klaper modeling language as a pivot meta-model within a star topology. However in the Klaper-based methodology model transformations are not "horizontal" and model transformations are not derived from semantic bindings: they must be manually developed.

In the field of software architectures, the ACME initiative [12] is famed for being one of the very first technologies to tackle the interoperability problem. Differently from **DUALLY**, it is neither MOF compliant nor automatized. Further on, a programming effort (rather than graphically designed semantic links) is needed every time a notation must be related to the ACME language.

Finally, the Eclipse project named Model Driven Development integration (MDDi[1]) presents an interesting approach based on the concepts of model bus and semantic bindings, but it is still in a draft proposal state.

6. Conclusions and further work

This paper presented as **DUALLY** can be considered a multi-domain modeling framework. This is possible thanks to extension mechanisms of **DUALLY** that allows us to extend it with domain specific concerns. The paper presents also the **DUALLY**zation of AADL and then the enabled transformations among AADL, UML, and Darwin/FSP.

As future work we plan to better investigate the use of **DUALLY** as a multi-domain modeling framework. In fact, one limitation of the current **DUALLY** implementation is that the only relation among different **DUALLY** extensions is the inheritance and that the allowed inheritance is only the "single inheritance". In order to deal with this limitation we are following three different ways: (i) allowing the "multi inheritance" (i.e., to build a **DUALLY** extension by extending two or more different extensions of A_0), (ii) investigating the definition of model transformations also among **DUALLY** extensions and (iii) investigating on merging techniques to build a new **DUALLY** extension from already existing **DUALLY** extensions.

1. http://www.eclipse.org/proposals/eclipse-mddi/

Acknowledgment

The authors wish to thank Gianluca Croce who provided an initial version of the AADL_A_0 weaving model.

References

[1] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, January 2000.

[2] P. Pelliccione, P. Inverardi, and H. Muccini, "CHARMY: A framework for designing and verifying architectural specifications," *Transactions on Software Engineering (TSE). IEEE Computer Society*, 2009.

[3] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri, "Providing architectural languages and tools interoperability through model transformation technologies," *IEEE Transactions on Software Engineering*, to appear, 2009.

[4] S. Gérard, P. Feiler, J.-F. Rolland, M. Filali, M.-O. Reiser, D. Delanote, Y. Berbers, L. Pautet, and I. Perseil, "UML&AADL '2007 grand challenges," *SIGBED Rev.*, vol. 4, no. 4, pp. 1–1, 2007.

[5] H. P. Feiler, B. Lewis, and S. Vestal, "The SAE Architecture Analysis and Design Language (AADL) Standard," in *IEEE RTAS Workshop*, 2003.

[6] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. R. O. Silva, "Documenting Component and Connector Views with UML 2.0," Carnegie Mellon University, Software Engineering Institute, Tech. Rep. CMU/SEI-2004-TR-008, 2004.

[7] J. Magee, "Behavioral analysis of software architectures using LTSA," in *Proceedings of the 21st international conference on Software engineering, IEEE Computer Society Press*, 1999, pp. 634–Ű637.

[8] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," Tech. rep. CMU/SEI-2006-TN-011, Tech. Rep., 2006.

[9] I. Hamid, "Flight Control System," 2005.

[10] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Lifting metamodels to ontologies - a step to the semantic integration of modeling languages," in *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MODELS'06)*, 2006.

[11] V. Grassi, R. Mirandola, E. Randazzo, and A. Sabetta, *The Common Component Modeling Example: Comparing Software Component Models*. Springer-Verlag, 2008, ch. KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability, pp. 327–356.

[12] D. Garlan, R. T. Monroe, and D. Wile, "Acme: Architectural description of component-based systems," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.