Developing next generation ADLs through MDE techniques

Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, Alfonso Pierantonio
Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, L'Aquila, Italy
{davide.diruscio,ivano.malavolta,henry.muccini,
patrizio.pelliccione,alfonso.pierantonio}@univaq.it

ABSTRACT

Despite the flourishing of languages to describe software architectures, existing Architecture Description Languages (ADLs) are still far away from what it is actually needed. In fact, while they support a traditional perception of a Software Architecture (SA) as a set of constituting elements (such as components, connectors and interfaces), they mostly fail to capture multiple stakeholders concerns and their design decisions that represent a broader view of SA being accepted today. Next generation ADLs must cope with various and ever evolving stakeholder concerns by employing *semantic extension mechanisms*.

In this paper we present a framework, called BYADL – Build Your ADL, for developing a new generation of ADLs. BYADL exploits model-driven techniques that provide the needed technologies to allow a software architect, starting from existing ADLs, to define its own new generation ADL by: i) adding domain specificities, new architectural views, or analysis aspects, ii) integrating ADLs with development processes and methodologies, and iii) customizing ADLs by fine tuning them. The framework is put in practice in different scenarios showing the incremental extension and customization of the Darwin ADL.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.2.11 [**Software Engineering**]: Software Architectures; D.2.10 [**Software Engineering**]: Design;
D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Modeling.

Keywords

Software Architecture, ADL, model driven, metamodeling.

1. INTRODUCTION

Early Architecture Description Languages (ADLs) [25], proposed during the 1990s, had the main purpose to design an "*ideal*" ADL [15] enabling the specification of any feature and element constituting a software architecture. These languages enabled the specification of components, connectors and their overall interconnection [28, 15], as well as composition, abstraction, reusability, configuration, heterogeneity, and analysis [32]. Medvidović and Taylor in [25] tried to define the requirements an ADL shall satisfy. Then ADLs evolved into a new generation of notations, each one dealing with more specific features [29, 8], such as configuration management, distribution, and product line modeling support.

A broader view of SA is being accepted today, which goes far beyond the traditional perception of an SA as a set of constituting elements and looks at multiple stakeholders concerns and their design decisions [24, 16, 20, 34, 23]. Based on this broader view of SA, what is evident nowadays is that an ideal and general purpose ADL cannot exist, and architectural languages must be extensible so to be defined by stakeholder's concerns. Building on the conclusions of [24, 16], the main emergent requirements of future (next generation) architecture modeling approaches are:

R1. *Domain specific concerns*: suitable mechanisms are required to allow software architects to specialize general architecture elements to a particular domain by adding new elements and leaving out unnecessary details and constructs;

R2. *Multiple views*: multiple views are required to keep the software architecture description cognitively manageable, each view represents a subset of the modeled concerns while hides the others;

R3. *Analysis features*: assuring as early as possible the correctness of an SA is fundamental in order to produce quality software [27]. Typically each analysis technique requires analysis-specific notations that should be part of the selected ADL;

R4. *Interoperability with other ADLs*: often a single ADL can provide features that (only) partially satisfy the stakeholders needs: future ADLs should be able to interoperate so to globally satisfy all the stakeholders needs [23]. Sometimes could be preferable (also for legacy purposes) to keep separated some features instead of concentrating everything in only one notation;

R5. *Promote architecture-centric development*: SAs should be used as a high-level design blueprint of the system to be used during system development and later on for maintenance and reuse. Therefore, ADLs should be integrated into development processes that consider also requirements, implementation, maintenance, etc;

R6. *Tool support*: as stated in [24], "tools are as important as notations". Tools should support several features, as editing, visualization, extensions creation, interoperability, and analysis.

Extensible and domain- and style-specific ADLs have been proposed to cope with those emerging requirements: Acme [14], ADML [18], and xADL [7] are first attempts of extensible ADLs that can be adapted to new and evolving concerns but, as pointed out in [24, 23] and analyzed in Section 2, they do not address many

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

challenges of next generation ADLs and they do not deal with semantic aspects of extensions in a satisfactory manner.

In this paper we present a framework, called BYADL (staying for *Build Your ADL*), for developing next generation ADLs according to the requirements listed above. We exploit model-driven technologies and tools which are more powerful with respect to techniques used in the past, such as normal programming languages or XSLT that may suffer from code maintainability and scalability issues [6]. In BYADL, the specification of an ADL mainly consists of a metamodel whose semantics is given in terms of the **D**U**AL**LY [23] semantic core. In particular, BYADL is fully integrated and built on top of **D**U**AL**LY, a framework that provides ADLs and tools interoperability. Moreover, BYADL provides composition mechanisms (presented in Section 4.1) enabling a software architect to create its own ADL by customizing or extending existent ADLs. In BYADL interoperability and extensibility mechanisms coexist since we believe that it is not practical to put everything in one architectural language: once the new ADL has been built by using BYADL, it can interoperate with all the other notations that are already in the **D**U**AL**LY environment. For this reason we do not promote, even though this could be technically possible, the composition and merging of two different ADLs. Contrariwise, with BYADL, an existing ADL can be extended with (i) domain specific concerns, (ii) analysis notations, (iii) new architectural views, and (iv) methodologies and processes that support other life-cycle activities. Furthermore, BYADL allows software architects to cut-off part of an ADL definition and to tune it by adding references and so on. ADL-specific tools are build on top of BYADL that enables functionalities like visualization, consistency checking, and multi-view handling of architectural descriptions. BYADL is implemented as Eclipse plugins that can be integrated with other technologies available in the Eclipse community.

Summarizing, the contributions of this paper are: $i)$ the analysis and definition of what a new generation of ADLs should have, $ii)$ an incremental approach to build customized and customizable ADLs via a set of well formalized metamodel composition operators, $iii)$ a tool that automates the approach of BYADL, $iv)$ the integration of architecture descriptions with software development processes and methodologies, $v)$ some evidence on the applicability of BYADL on the field.

The paper is organized as follows. Section 2 provides an analysis of existing towards-next generation ADLs. Section 3 provides background information of **D**U**AL**LY as required to understand the description of BYADL and introduces an illustrative example that is used throughout the paper. Section 4 describes the BYADL framework which is applied in different scenarios showing the extension and customization of the Darwin ADL [22]. Section 5 summarizes the illustrative example and opens a discussion about advantages and limitations of BYADL. Section 6 concludes this paper and outlines future work directions.

2. INITIAL EFFORTS IN THE DIRECTION OF NEW GENERATION ADLS

Some initial effort has been done in the direction of a new generation ADL through the definition of extensible and domain- and style-specific ADLs [12, 7, 14, 18, 26].

Acme [14] is famed for being one of the very first technologies to tackle the problem of interchange in ADLs. Acme was born as a simple, multi-style ADL framework also providing foundations and mechanisms to extend itself. More precisely, Acme provides tooling extension points introduced to allow other tools to physically read and write Acme descriptions while its semantics can be extended by means of properties that can decorate each element. These properties are uninterpreted and it is up to user-written tools to properly parse and use them. Therefore, the most challenging aspects of the extension, i.e., the semantic extension, is not properly supported. Similar description can be provided for ADML [18] that is an XML-based language derived by Acme. The main addition with respect to Acme is the possibility to add meta-properties, i.e., a mechanism to define properties and property types of particular elements. The extensibility mechanism is the same of Acme.

xADL [7] as well as its core xArch, are based on XML and thus fully extendable [8]. However, XML schemas do not provide facilities to define the semantics of individual elements but only the syntax. The semantics is fundamental in order to define a common and well defined means to interpret the syntactic concepts. In xADL the semantics can be encoded into comments in the XML schemas themselves, in a project documentation, in visualization tools that stakeholders use or, finally, in analysis tools associated to a given feature [24].

UML 2.0 [26] has been used for modeling architectures and different profiles have been proposed. Even thought UML is a rich language composed of 13 different diagrams, they are not enough for modeling every possible architectural concern. While extension mechanisms of UML allow software architect to define stereotypes and tagged values, which can be used to extend UML elements to better capture domain specificities, these mechanisms cannot be used to define new diagrams. These extensions cannot fully represent all concepts of every ADL and on the other side, as already claimed in [25, 24], it is impractical to have a "universal" notation.

AADL [12] was born as an avionics focused Domain Specific Language (DSL) and later on has been revised in order to represent and support embedded real-time systems. AADL was designed as an extensible language supporting modeling from multiple aspects and viewpoints. The extension mechanisms of AADL include the definition of custom properties to specify additional ADL-specific analysis and/or generic information to be attached on the architectural design. The extensibility mechanisms defined through constructs such as the standard "annex" plus the mentioned property set extensions are closely related to our view of "semantic" enhancement, which we consider very important in any technology. Unfortunately, AADL does not provide automated support to its extensibility possibilities.

All these attempts to create extensible and domain- and style-specific ADLs do not sufficiently take into account the problem of reusing already defined extensions. In particular each extension is coupled and specifically defined for the particular ADL and cannot be reused for extending other ADLs.

Even though out of the scope of software architectures, in [11] the XTEAM framework has been proposed for creating Domain Specific Development Infrastructures (DSDIs). In XTEAM, metamodel composition is the initial step for creating modeling infrastructures. Metamodels are linked to a proposed metamodeling language (called ACT) in order to establish their semantics. The approach has been implemented within the GME environment[1] which gives the possibility to have an unrestricted number of levels in the modeling architecture. In fact, ACT is defined as a metamodeling language in terms of another metamodeling language. This aspect locks the specified ACT models with GME, since other technologies like EMF[2], or MOF[3] do not give the possibility to define

[1]GME - Generic Modeling Environment: `http://www.isis.vanderbilt.edu/projects/gme`

[2]Eclipse Modeling Framework (EMF) project Web site: `http://www.eclipse.org/emf`

[3]OMG MetaObject Facility (MOF): `http://www.omg.org/mof`

other metamodeling languages prior the definition of metamodels. XTEAM promotes domain-specific analysis through the use of extensible Model Interpreter Frameworks (MIFs) that transform the composed metamodel to analysis-specific notations. However this is done programmatically and not through a dedicated architectural interoperability framework.

Summarizing, the major limitations of these approaches with respect to the six requirements highlighted in the introduction are:
1. These approaches propose extension mechanisms that only partially satisfy requirements **R1**, **R2**, and **R3**. For instance, as described before, Acme supports the extension of its semantics only by means of properties that cannot be easily interpreted by tools. The lack of semantics can lead to misunderstanding and specifically, as described in [11], the lack of semantics within metamodeling languages can lead also to several practical problems: *i)* imprecise determination of semantic relationships between the ADL and the elements that must be integrated within the ADL, *ii)* onerous manual composition of metamodels, and *iii)* lack of rigorous and automated validation mechanisms;
2. Requirement **R4** is not addressed properly by any language. Acme provided some initial ideas and more mature results can be found in the **D**U**AL**LY approach [23];
3. Concrete and scalable solutions for **R5** are missing;
4. Mature tools supporting the features described by requirement **R6** are missing.

A comprehensive discussion on how BYADL satisfies the **R1-R6** requirements will be provided in Section 5.

3. BACKGROUND

3.1 DUALLY

DU**AL**LY [23] is a tool-supported framework to create interoperability among different architectural languages. Given a number of architectural languages and tools, **D**U**AL**LY allows interoperability among them through automated model transformation techniques. Any transformation among ADLs is defined in **D**U**AL**LY by passing through A_0, a core set of architectural concepts. Therefore, A_0 provides the infrastructure upon which to construct semantic relationships among different ADLs. In other words, it acts as a bridge among the different architectural languages to be related together. A_0 has been defined as general as possible to ensure that **D**U**AL**LY is able to potentially represent and support any kind of architectural representation (i.e., formal ADLs or UML-based languages). The selection of the elements within A_0 has been performed by studying existing architectural languages and UML. The authors exploited and inherited features they judged satisfactory, overcoming identified limitations (e.g., xArch is extensible but makes use of XML, UML is very expressive but is ambiguous, etc.). The main elements of A_0 are `Architecture`, i.e., a collection of components (`SAcomponent`) and connectors (`SAconnector`) instantiated in a configuration, `SAinterface` which specifies the interaction point between an `SAcomponent` or an `SAconnector` and its environment, `SAtypes` to define architectural types, its specialization `SAstructuredType` which can contain also a definition of sub-architectures, and so on. We refer to [23] for further details about A_0 and its metamodel.

3.2 Illustrative example

This section introduces an illustrative example that will be used as running example during the description of each aspect of BYADL. This example is organized in three scenarios that permit to cover the features required by a new generation ADL:
• *Extending ADLs with Domain Specific & Analysis concerns or with additional views*: in this scenario the idea is to extend a given ADL with domain specificities, analysis notations, or additional views. The result of the extension is an extended ADL. We collected these three challenges in one scenario which covers the requirements **R1**, **R2**, and **R3**;
• *ADL & Development process*: in this scenario the idea is to create interoperability among the ADL and a development process in order to show how BYADL promotes architecture-centric development. This scenario covers the requirement **R5**;
• *ADL customization*: the macro-requirement of satisfying stakeholders concerns sometimes requires to leave out unnecessary concepts or to add specific constructs. This scenario is devoted to this objective and is orthogonal to the analyzed requirements.

These scenarios are applied incrementally. They do not take into account the **R4** requirement that is natively supported via **D**U**AL**LY while the **R6** requirement will be discussed in Section 4.3.

Let us suppose now that our company has invested time and effort for acquiring the required knowledge and experience on the Darwin ADL [22], and would like to use such an ADL as much extensively as possible. Let us assume that we have to model a fault-tolerant system where it is extremely important to explicitly model (i) connectors, (ii) both normal and exceptional behaviors, and (iii) the development strategies related to the SA. Unfortunately, all those features are not supported by Darwin. The first step is to consider the Darwin ADL [22] as the staging point for the composition. Now we cover all the concerns of the current system being developed (e.g., fault tolerance, direct link to the development process, and so on) by instantiating the three scenarios introduced at the beginning of this section.
DarwinFT: Extending Darwin with Fault Tolerance. In order to generate DarwinFT we compose the Darwin ADL with the *IdealComponent* UML profile presented in [9] for specifying software components according to the idealized fault tolerant component model [13]. Each ideal component is composed of both normal and exceptional parts and exceptions can either be signaled or handled through specific interfaces.
DarwinFT+BPMN: DarwinFT & Development process in BPMN. In this scenario, we show how DarwinFT can be composed with the BPMN metamodel[4]. Upon doing so, software architects can associate structural parts of the system to specific development activities. The design/development process is directly linked to its business context, including development strategies, costs, risks, etc.
$(DarwinFT+BPMN)_{cc}$: Darwin customization. We customize the DarwinFT+BPMN language by adding software connectors as first-class elements. Components may communicate also through connectors now and connectors have associated portals that act as architectural roles.

4. THE FRAMEWORK

BYADL provides software architects with extensibility features to build, starting from an existing ADL, their own *ideal* ADL. It is important to note that extending an ADL does not mean to compose and merge two different ADLs. Even though this could be technically possible this could lead to the creation of a "chaotic" and "vague" language. Therefore, BYADL supports the extension of an existing ADL with domain specificities, architectural views, and analysis aspects. Furthermore, BYADL supports also the integration of the selected ADL with development processes and methodologies. The high-level design of BYADL is described in Figure 1. An ADL can be considered a DSL specialized in the domain of software architectures, so it is essentially composed of (i) *abstract*

[4]BPMN specification: `http://www.bpmn.org/`

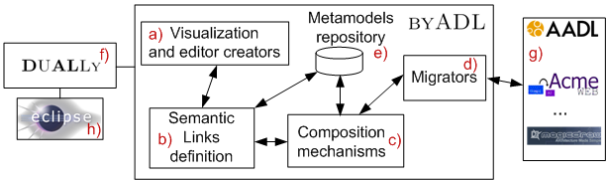


Figure 1: The high-level design of BYADL

syntax, i.e., the set of language concepts and their relationships, (ii) a set of *concrete syntaxes*, i.e., textual and graphical notations to visualize and edit the models, and (iii) *semantics* describing the meaning of the language's constructs [31]. The *abstract syntax* of the ADL is represented in terms of the metamodel obtained by means of the composition mechanisms shown in Figure 1 part *c* and explained in Section 4.1. The metamodels representing the extensions are stored in a repository of metamodels for further uses (Figure 1 part *e*). For each ADL different *concrete syntaxes* can be specified and their definition is managed by the "Visualization and editor creators" component in Figure 1 part *a*; Section 4.3 will go into the details on how BYADL supports it.

The semantics of the new ADL is provided by means of **DUALLY** (Figure 1 parts *b* and *f*). More precisely, this is realized through A_0, the semantic core of **DUALLY** that is, as explained in Section 3.1, a centralized set of elements with respect to which relationships must be defined. Once a software architect provides relationships between elements of a metamodel and elements of A_0, the elements of the ADL implicitly inherit the built-in semantics of A_0. Note that this way of providing semantics is (conceptually) the same proposed in [31] in which the semantics of DSLs, called translational semantics, is provided by means of model-to-model transformations from the DSL to another language. In our case the target language is A_0. For this reason we oblige the software architect to define these semantic relationships between the ADL to be extended and A_0, and between the metamodel that represents the extension and A_0 as a pre-requisite to operate with BYADL. Then, for each element of the ADL and of the metamodel that represents the extension, the software architect selects, through the graphical interface of BYADL, the most appropriate metaclass of A_0. The result is that each element of the ADL has one and only one *type-of* [11] relationship with an element of A_0. For instance, referring to the metamodel of Darwin shown in Figure 4 and to the background of **DUALLY**, the Darwin `ComponentInstance` is typed `SAComponent`, the Darwin `ComponentDeclaration` is typed `SAStructuredType`, and the Darwin `Portal` is typed `SAInterface`, and so on. In the next sections we will discuss the importance of the semantic links to A_0 by describing how the semantic links help designers during the composition phase and improve the quality of the concrete syntaxes of the composed ADLs.

Finally, as shown in Figure 1, BYADL is integrated into the Eclipse platform and inherits from **DUALLY** also architectural languages and tools interoperability. Given any number of architectural languages and tools, they can all interoperate thanks to automated model transformation techniques provided by **DUALLY**.

The *Migrators* building block of BYADL (see Figure 1 part *d*) automatically generates (for each composed ADL) model transformations able to reflect the architectural models defined within the newly created ADL, back to the original tools and notations. In this way, compatibility with previous editing and analysis tools is guaranteed (see Section 4.2).

Figure 2 provides an overview of the composition mechanisms and of the model migrators. The whole BYADL framework relies on the Atlas Model Management Architecture (AMMA) [3]. We

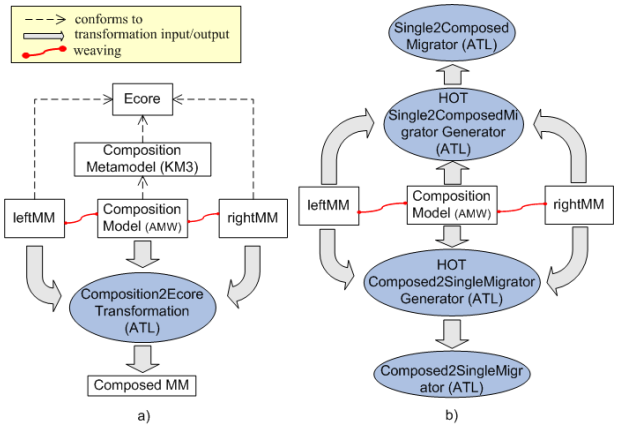


Figure 2: a) Composition mechanism; b) Model migrators

chose AMMA since it best fits to our technical needs, like metamodel independence, high flexibility of the composition behavior, integration to Eclipse and its modeling facilities [2] [17]. The technology we use for metamodel composition is the Atlas Model Weaver (AMW) [19]. It allows the definition of correspondences among (meta)models and links among model elements. Such links are captured by weaving models conforming to an extensible weaving metamodel. Model transformations are specified using the Atlas Transformation Language (ATL) [19], an hybrid model transformation language with declarative and imperative constructs. Both models and metamodels are expressed via Ecore, the metamodeling language of EMF, and serialized into XMI files.

The composition mechanism (Figure 2.a) relies on the `Composition` weaving model; it conforms to the `Composition Metamodel` which specifies the types and structure of the BYADL composition operators. This metamodel is given in KM3 (Kernel MetaMetaModel)[5], a domain specific language for specifying metamodels. The composition model allows the composition of two metamodels, (`leftMM` and `rightMM` in figure) into a metamodel `composedMM` by executing the *Composition2Ecore* transformation. This transformation takes as input `leftMM`, `rightMM`, and the `Composition Model` and produces `composedMM` by executing the semantics of the applied operators.

An overview of the migrators infrastructure is reported in Figure 2.b. More precisely, starting from the metamodels to be composed (i.e., `leftMM` and `rightMM`) and the specification of their composition (i.e., `Composition Model`), two transformations are automatically generated: *Composed2SingleMigrator* and *Single2ComposedMigrator*. The former is able to generate two models conforming to `leftMM` and `rightMM` from a model conforming to the composed metamodel. On the other way round, the *Single2ComposedMigrator* transformation generates a model conforming to the composed metamodel, starting from models conforming to the single ones. Such transformations are automatically generated by means of the execution of two Higher-Order Transformations (HOT): *Composed2SingleMigrator Generator* and *Single2ComposedMigrator Generator* which are also defined in ATL. The BYADL migrators engine is fully metamodel-independent because the HOTs are generic, i.e., they do not depend neither on the composed metamodel nor on the single ones. Section 4.2 details the migrators engine.

BYADL is able to deal also with UML profiles. More precisely, BYADL reuses the **DUALLY** mechanism for importing a UML profile [23] by considering them as Ecore metamodels.

[5]KM3 Web page: `http://www.eclipse.org/gmt/am3/`

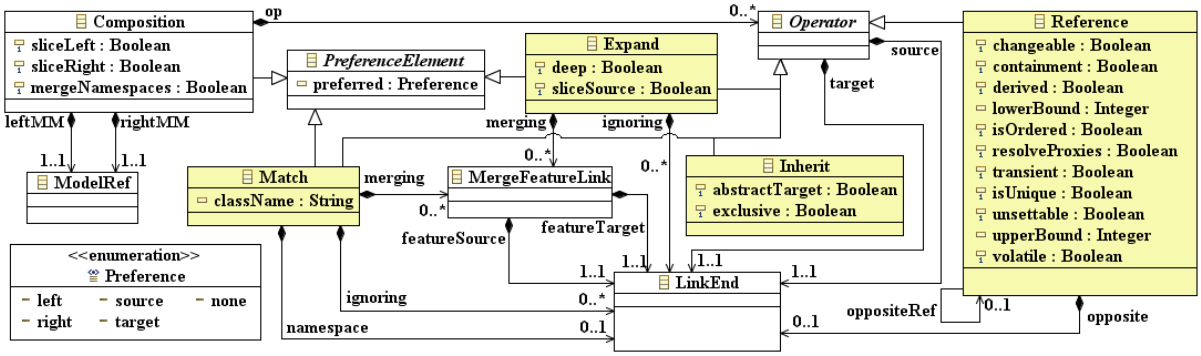


Figure 3: Definition of BYADL metamodel composition operators

4.1 Composition mechanisms

Within the BYADL framework, metamodel composition is the means by which the software architect creates an architectural language. This task is performed by applying specific composition operators to metamodels extracted from the Metamodels Repository. The BYADL operators, namely *Match*, *Inherit*, *Reference*, and *Expand*, have been defined building on those presented in [9, 33]. In particular, we made the operators proposed in [9] applicable on Ecore metamodels and we refined those introduced in [33] in order to lighten the composition process. For example, the deletion of elements is implicitly performed in BYADL by a slicing algorithm. Our approach is somehow parallel to that proposed by Ledeczi et al. [21]: in that approach the structure of the metamodels to compose cannot change (so that original models are still valid). BYADL's migrators permit to relax this constraint, providing more flexibility while designing and applying the composition operators. Aside from the composition operators, the BYADL conflict resolution mechanism is based on the work presented in [30].

The BYADL composition operators are defined in terms of the weaving metamodel reported in Figure 3. The *Composition* metaclass represents the root of each composition model. It references the metamodels to compose (i.e., `leftMM` and `rightMM`) and contains a set of operator applications (*op* in Figure 3) which can be given in any order without changing the result. The *mergeNamespaces* attribute specifies if the composition will preserve the packages structure of the metamodels to compose or if the composed metamodel will contain a single root package.

The default behavior of the composition is to consider the union of all the metaclasses in both `leftMM` and `rightMM` and to interrelate them by applying the specific operators. Such behavior may vary depending on the *sliceLeft* or *sliceRight* attributes. If *sliceLeft* is true, then only the portion of the left metamodel involved in some operator application will be part of the composed metamodel. The same mechanism holds for *sliceRight*. The relevant portion of metamodel is computed by executing a slicing procedure before calculating the union of the metaclasses in `leftMM` and `rightMM`. This slicing procedure is based on a slicing algorithm for UML models [1]. Let MM be a metamodel and let SC be a subset of the elements in MM; $slice$ is defined as follows:

$$slice(SC) = SC \cup \bigcup_{c \in SC} slice(neighbour(c))$$

where $neighbour(c)$ is the set of all superclasses of c, of all classes referred (both with association and aggregation) by c, and of all types of attributes in c. It is important to note that even though $slice$ is defined as a set of classes, since each class contains also references to other classes, the final result is a part of the metamodel MM with both classes and their relationships.

In *PreferenceElement*, the optional *preferred* attribute is used to solve conflicts during composition: if the composition engine has to make an arbitrary choice, then it chooses the option specified by the preferred element. If *preferred* is defined both in *Composition* and in an operator, then the operator's preferred "overrides" the one of *Composition*. Each operator is always applied on two metaclasses that we refer to as source (s) and target (t) in the remainder of this section. The composition operators of BYADL are:

Match. s and t semantically overlap and then they are merged into a single metaclass c in `composedMM`. c contains the union of all the structural features (i.e., both attributes and references) of s and t. The features referenced by the *ignoring* aggregation are discarded. The *merging* reference specifies if two features must be merged before calculating the union of the features of s and t. This is necessary in case two features are semantically the same, but they are syntactically different. The supertypes and subtypes references are merged. The *className* optional attribute represents the name of the merged metaclass. If this attribute is not specified, then the preferred metaclass's name is chosen, otherwise the name will be the concatenation of the names of s and t (the latter case is signalled to the user with a warning). The *namespace* attribute specifies the package that will contain the c metaclass. If it is not set, then the root package of `composedMM` is chosen. This operator is subject to semantic checks (and corresponding solving mechanisms) that handle possible conflicts between properties of s and t. For example, if both metaclasses are abstract then the resulting metaclass c is abstract, otherwise it is not abstract. The same mechanism holds for the interface property.

Inherit. This operator specifies that s will be a subtype of t in the resulting composed metamodel. If its application results in a cycle in the inheritance tree, then it is not executed and a warning is raised. The *abstractTarget* attribute specifies if the t metaclass will be abstract (then it cannot be instantiated) in the composed metamodel. If the *exclusive* attribute evaluates to *true*, then the metaclass corresponding to s in `composedMM` will extend only t, discarding all its previous inheritances in its original metamodel.

Reference. In the composed metamodel s references t. All features of the new reference (e.g., lowerbound, name, opposite) can be set while applying this operator. If the new reference *ref* overlaps an existing one, then it replaces the old one. The *oppositeRef* feature is used to set the opposite of *ref* to the result of another application of the *Reference* operator.

Expand. The attributes of s (leaving out the attributes specified in the *ignoring* aggregation) are copied into t. The standard merge operation is executed to manage attributes with the same name. If the *deep* attribute is true, then all attributes of s (including the inherited ones) are recursively copied into t. The *sliceSource* attribute specifies if the s metaclass is still part of the composed metamodel.

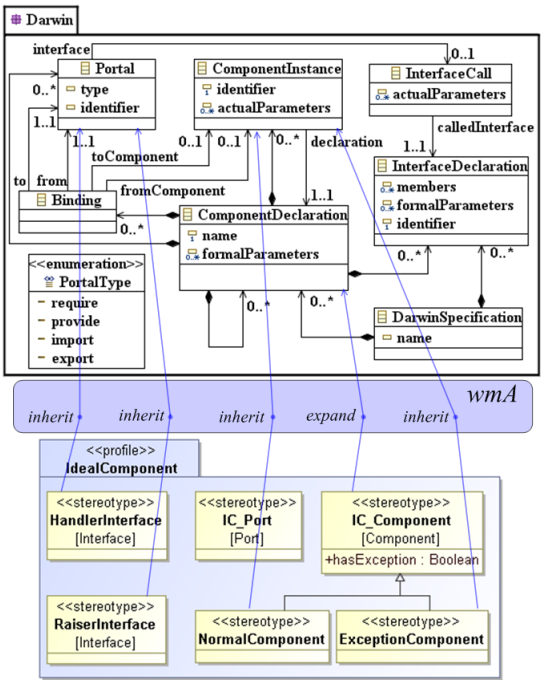


Figure 4: DarwinFT: Extending Darwin with Fault Tolerance

If this property evaluates to true, then: i) the *s* metaclass is deleted from `composedMM`, ii) all the metaclasses extending *s* will extend the direct supertypes of *s*, iii) all the references whose type is *s* are deleted from the composed metamodel. The *ignoring* and *merging* aggregations have the same semantics as in the *Match* operator.

The composition engine performs various semantic checks to avoid incidental conflicts, such as attributes overlapping with different types, inheritance cycles, and so on. The composition phase solves also discovered conflicts. The rationale that guides the solution of conflicts in BYADL is to propose a combination of operator's preferred element, if specified, and to propose a default behavior for each operator in the remaining cases. For the sake of brevity, this work does not describe them in detail.

If structural features overlap, merging and feature equivalence detection mechanisms have been implemented, inspired by the UML `PackageMerge` algorithm [10]. Without going into the details, the main principle is to keep consistency while trying to preserve, as much as possible, the semantics of the metamodels to compose.

It is not infrequent that structure and semantics of a MOF metamodel are restricted through OCL constraints. Such constraints must be preserved since, as stated above, the composed metamodel must embrace as much semantics as possible from the initial metamodels. In that sense, the BYADL composition engine copies the OCL constraints into `composedMM` and refines them according to the `composedMM` package structure. By doing that, within the resulting `composedMM`, incidental ambiguities and constraint clashes are reduced to a minimum.

The BYADL composition engine reuses the user interface of AMW. `LeftMM` and `RightMM` are rendered using a tree-based editor and operators are graphically applied via a central panel. We extended the AMW interface so that the semantic links to the A_0 metamodel guide the application of the composition operators. More precisely, once applying an operator, BYADL proposes as target only metaclasses that are semantically compatible with the source metaclass. In this respect, semantics compatibility of metaclasses depends on the type of operator being applied.

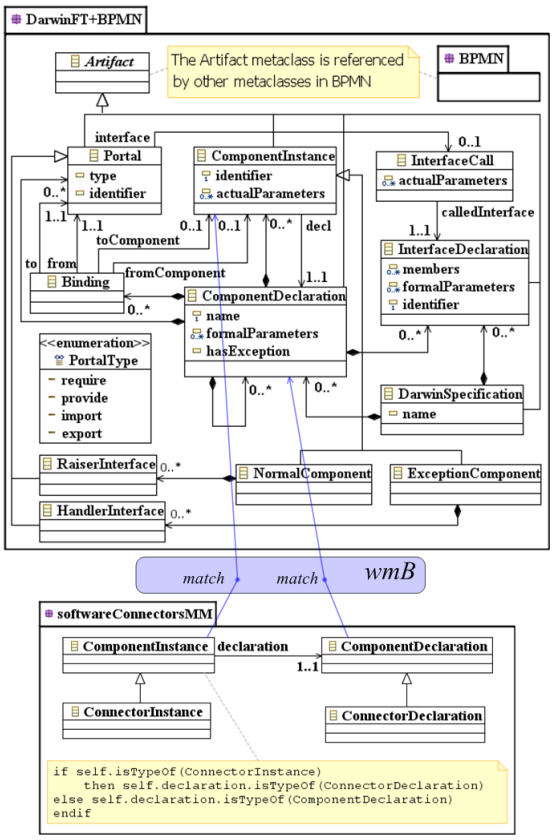


Figure 5: (DarwinFT+BPMN)$_{cc}$: Darwin customization

Focusing on the illustrative example in Section 3.2, in the following we show how the BYADL composition operators are applied in practice. Figure 4 shows a simplified version of the first scenario, i.e., the extension of the Darwin metamodel (introduced in [23]) with the IdealComponent UML profile (proposed in [9]). In this scenario we use the *inherit* and the *expand* operators. Each *inherit* operator has the exclusive property set to true. The *inherit* operator allows the specialization of the `Portal` metaclass of Darwin with `HandlerInterface` and `RaiserInterface` of the idealized component model; they are two interfaces specialized for managing and raising exceptions, respectively. Furthermore, the *inherit* operator is used to specialize a `ComponentInstance` of Darwin in `NormalComponent` and `ExceptionComponent` representing the normal and the exceptional part of a component, respectively. Finally, the *expand* operator is used to add the `hasException` attribute to a `ComponentDeclaration` of Darwin. Such attribute specifies whether the component's behavior is specified through the normal and exception sub-components.

The second scenario deals with the integration between DarwinFT and the Eclipse BPMN metamodel[6]. It makes an extensive use of the *inherit* operator. The main idea is to allow designers to associate Darwin structural elements to BPMN activities (e.g., tasks, sub-processes). The `Artifact` BPMN element has been conceived as an extension point to provide additional information to BPMN processes, so each DarwinFT element extends the `Artifact` metaclass through the *inherit* operator. Due to space limitations and to the simplicity of this composition scenario, we do not show it graphically in this work.

Figure 5 shows the third step, namely the customization of Dar-

[6] `http://www.eclipse.org/bpmn/`

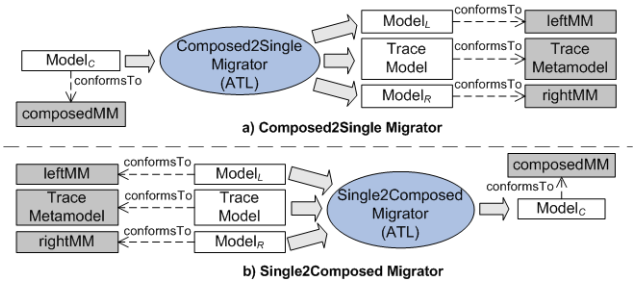


Figure 6: Composed2Single and Single2Composed migrators

winFT+BPMN performed by adding the concept of software connector. This scenario makes use of the *match* operator with the aim to match the `ComponentInstance` metaclass of Darwin with the `ComponentInstance` of the softwareConnectorsMM metamodel that we built ad-hoc. The result is a unique metamodel in which `ConnectorInstance` specializes `ComponentInstance`, and `ConnectorDeclaration` specializes `ComponentDeclaration`. It should be noted that the OCL constraint will also be part of the final composed metamodel; this avoids having connector instances declared as components and vice versa.

4.2 Model migrators

As said in the previous section, BYADL provides software architects with the means to create customized ADLs by extending existing ones. However, in this scenario it is important that tools supporting original ADLs can still be used to manipulate and analyze models which have been created by means of the new composed ADLs. In order to cope with such situations, BYADL provides two HOTs which are able to generate specific model migrators. In particular, the *Composed2SingleMigrator Generator* reported in Figure 2.b generates a *Composed2SingleMigrator* (see Figure 6.a) starting from the metamodels being composed and the weaving model specifying their composition. This generated transformation takes as input a model $Model_C$ conforming to the metamodel `composedMM` and generates three different models: $Model_L$, $Model_R$, and `Trace`. $Model_L$ and $Model_R$ contain the elements in $Model_C$ which can be represented by means of the metamodels `leftMM` and `rightMM`, respectively. The `Trace` model contains the information which is required to compose models by means of the generated transformation *Single2ComposedMigrator* described in the rest of this section.

Listing 1: Excerpt of the Composed2SingleMigrator Generator

```
1 ... helper context ECORE!EClass def: toRuleString():
      String =
2  if not self."abstract" then
3    if self.inherits() then
4      self.inheritManagementRule()
5    else
6      if self.comesFromMatch() then
7        self.matchManagementRule()
8  ...
```

Listing 1 reports a small fragment of the *Composed2SingleMigrator Generator*. Essentially, each metaclass in the composed metamodel is taken into account to determine how it has been obtained by the composition process. Depending on the operator which has been used, a corresponding transformation rule is generated. For instance in case of classes obtained from a match operation, the `matchManagementRule()` (see lines 3-4) is executed to generate a transformation rule which is able to decompose the instances of the considered matched class.

Figure 8.a reports a sample model conforming to the metamodel (DarwinFT+BPMN)$_{cc}$. The model consists of a BPMN diagram specification containing a Darwin description augmented with modeling elements provided by the softwareConnectorsMM metamodel. For example, the element `NetworkType` is a connector declaration which can be given because of the match between the metaclasses `ComponentDeclaration` in Figure 5.

The generated rule `ConnectorTOComponent_Connector` reported in Listing 2 is able to decompose all the source connector declarations (like the element `NetworkType` above). It generates a component declaration in the target DarwinFT+BPMN model, and a connector declaration in the target softwareConnectorsMM model. Additionally, the migrator generates a trace model in order to store the elements which have to be considered together when the separated models have to be recomposed.

Listing 2: Excerpt of the generated Composed2SingleMigrator

```
1 create OUT_M1 :MM1, OUT_M2 : MM2, OUT_TRACE : TRACE from
      IN : MM12;
2 ... rule ConnectorTOComponent_Connector {
3  from s : MM12!ConnectorDeclaration
4  to t: MM1!ComponentDeclaration (
5      hasException <- s.hasException,
6      ...
7     t2: MM2!ConnectorDeclaration ()
8  do {
9    thisModule.createTraceLink(t,t2, #match);
10 }
11 }
```

Being more precise, a trace model is another weaving model consisting of trace links. Each trace link relates two model elements which are instances of metaclasses that have been composed. Figure 7 reports a fragment of the trace model generated by applying the *Composed2SingleMigrator* on the sample model in Figure 8.a. The last trace link relates the `NetworkType` component declaration contained in the left DarwinFT+BPMN model with a connector declaration in the right softwareConnectorsMM model. Such model elements have been generated by applying the rule reported in Listing 2 on the source `NetworkType` connector declaration.

As previously said, BYADL provides software architects with the *Single2Composed Migrator Generator* depicted in Figure 6.b. This generator produces a *Single2Composed Migrator* able to generate a model $Model_C$ conforming to a composed metamodel `composedMM` starting from $Model_L$ conforming to `leftMM`, $Model_R$ conforming to `rightMM`, and a trace model obtained during a decomposition as explained above. For instance, by applying the generated *Single2Composed Migrator* to the models reported in Figure 7, a (DarwinFT+BPMN)$_{cc}$ model is obtained. The trace links drive this operation since they maintain the elements which contribute to the generation of a same target element. For instance, a target connector declaration named *NetworkType* will be generated because of the last trace link reported in Figure 7. Without this information the migrator is not able to distinguish the model elements

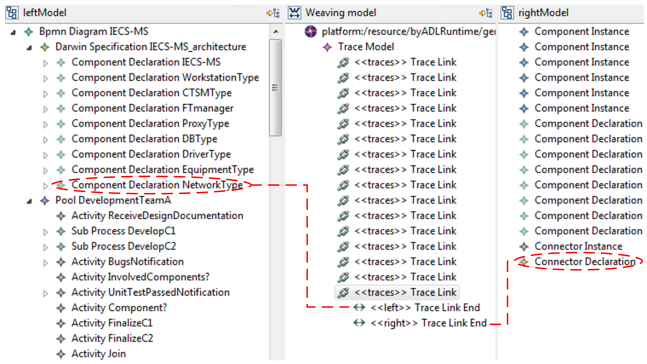


Figure 7: Sample generated Trace model

which have to be composed from those which have to be simply copied to the target model. Due to space limitation, we do not describe the code obtained by applying the *Single2Composed Migrator Generator* on the example, even though the interested reader can download it from the BYADL project Web site[7].

The migrators have been conceived by inheriting the work done in [4, 5]. In general, adapting models with respect to the occurring modifications in the corresponding metamodel is a challenging task. In fact there are many kinds of metamodel changes that have to be taken into account and which can be distinguished into *non-breaking*, and *breaking*. The former are metamodel modifications which do not brake the conformance of the existing models in contrast to the modifications which brake their conformance and that can be in turn classified into *resolvable* and *unresolvable*. An example of a breaking and resolvable modification is the deletion of a metaclass. In this case, existing models can be automatically adapted by eliminating all the instances of the deleted metaclass. However there are modifications (breaking and unresolvable) which require human interventions for their resolution, like the addition of an obligatory metaclass.

According to the semantics of the provided operators, and to the metamodel difference classification available in [4], in our case the migration of models can be automatically performed. This is because i) breaking and unresolvable modifications do not occur and ii) the trace models produced by any *Composed2Single Migrator* contain the necessary information to properly select the model elements which have to be composed by the corresponding *Single2Composed Migrator*.

4.3 Visualization and editor creators

Visualization features and editors are an important and ineluctable aspect of architecture description languages. In fact by means of editors software architects can draw, design, view, and reason about the software architecture of the system they are modeling. By using BYADL three different kinds of editors can be adopted, each characterized by a distinguishing usability level and a certain effort to be implemented. In particular, software architects can use a:

• *tree-based editor*: this graphical editor is obtained for free by EMF. Similarly to the tool support of xADL [7], this editor shows collapsible and hierarchical tree-like structures; it is very simple and useful for large size software architectures, even though it's usability level is quite low. Figure 8.a shows this editor at work;

• *textual editor*: starting from the metamodel of the created ADL, a textual editor conforming to the Human-Usable Textual Notation (HUTN) specification[8] can be automatically generated. HUTN is a generic specification that provides a concrete language for any MOF model. The produced textual editor supports also syntax highlighting and automatic conformance check with respect to the metamodel (i.e., the abstract syntax of the ADL). The usability level of this editor is slightly higher than the tree-based one. In fact, both require a deep knowledge of the metamodel even though in many cases a textual syntax is preferred to the tree based editor. An example of this editor is shown in Figure 8.b.

• *graphical editor*: software architects can also produce a graphical editor with customized graphical elements. This is possible by means of the EuGENia[9] tool. This tool automatically generates the models needed to implement an editor from a single annotated Ecore metamodel. EuGENia is based on the Eclipse Graphical Modeling Framework (GMF)[10] that provides a generative component and runtime infrastructure for developing graphical editors. Suitable annotations allow us to identify the different roles of the elements (such as the root object of the metamodel) and to specify graphical properties (such as *border.color*, *label.icon*, etc.). Let ADL_{new} be the new ADL we are building, let ADL_1 be the ADL we selected as ADL to be extended, and let MM be the metamodel representing the extension. The graphical editor features allow ADL_{new} to inherit graphical elements of ADL_1 and MM, under the assumption that annotated Ecore metamodels were available for ADL_1 and MM. In case of a graphical element for an element of ADL_{new} exists both in ADL_1 and MM, the *preferred* field of the operator we are using for composing can help to disambiguate. Sometimes ADL_1 and MM provide separated views that should be kept separated and sometimes they should be merged. BYADL, by means of its graphical interface, allows the software architect to select the most suitable solution. For instance in Figure 8.c we can see that Darwin, the idealized component model and the connector model have been integrated into a single view (white area in the figure), while BPMN has kept its separated view (gray area in the figure). Links between these two different views are added by hand in Figure 8.c to graphically show existent relationships among elements of these views.

Furthermore, semantic links specified among ADL_1 and A_0 and among the MM and A_0 enable the possibility to inherit the graphical annotations (and hence the graphical representations) of the A_0 linked elements. In other words, A_0 substitutes ADL_1 or MM if they do not have annotated Ecore metamodels. Finally, the obtained GMF editor can be polished and optimized even though this step requires a deep knowledge of GMF. The generation of graphical editors is in its prototypal version and currently it is not able to solve subtle conflicts that the different graphical elements may have. We better describe this point while discussing the limitations of the approach in Section 5.2.

To summarize, there are three different possibilities in BYADL to produce an editor for the ADL being developed. Depending on the effort and the knowledge that the software architect wants to invest, a textual or graphical editors can be conceived each with a different usability level.

5. EXPERIENCE AND EVALUATION

The illustrative example introduced in Section 3.2 and used throughout the paper is summarized in Section 5.1. Section 5.2 discusses advantages and disadvantages of the BYADL approach.

5.1 Summarizing the illustrative example

The illustrative example has been used in the entire paper to describe the main aspects and features of BYADL. In this section we give an overview of a system which has been modeled by using (DarwinFT+BPMN)$_{cc}$, i.e., the new ADL we have constructed by following the three scenarios presented in Section 3.2. The modeled system, called Integrated Environment for Communication on Ship (IECS) [23], is based on a specification coming from a project developed within Selex Communications, a company mainly operating in the naval communication domain. The purpose of this model is just to show how the newly created (DarwinFT+BPMN)$_{cc}$ ADL can be used in practice. Therefore, for space reasons we do not provide further informations on the IECS system.

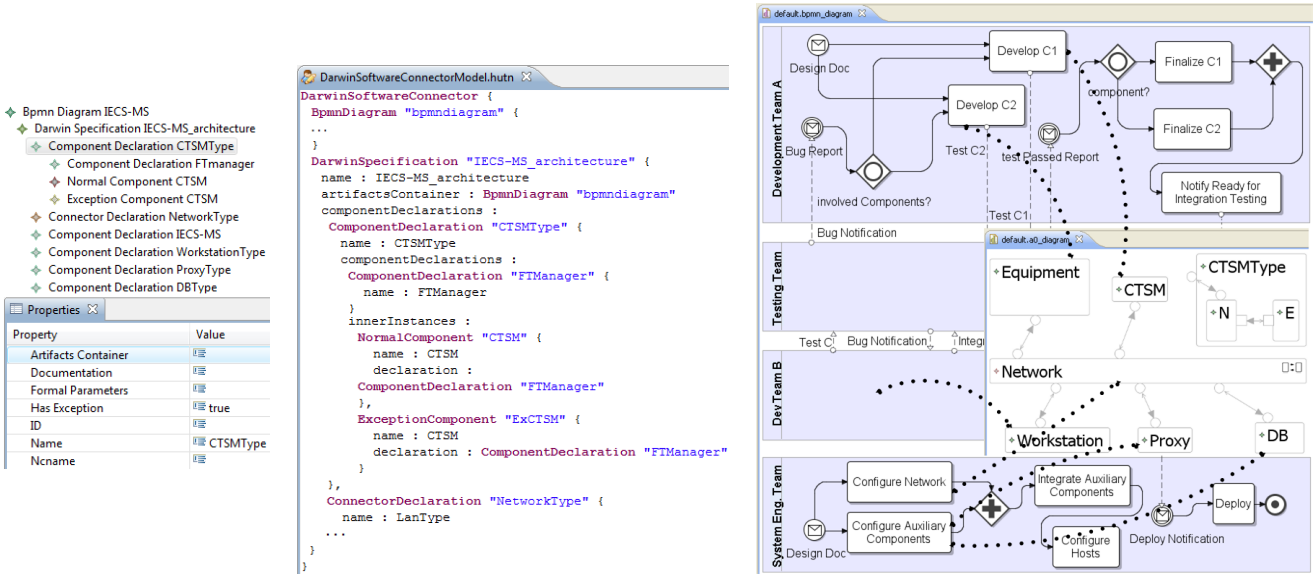
The integration of Darwin with BPMN allows us to model not only the SA of IECS, but also the adopted development strategies (see Figure 8.c). The IECS software architecture is composed of

[7] BYADL Web site: `http://byadl.di.univaq.it/`

[8] HUTN specification: `http://www.omg.org/technology/documents/formal/hutn.htm`.

[9] EuGENia GMF Tutorial: `http://sourceforge.net/apps/mediawiki/epsilonlabs/index.php?title=EuGENia`.

[10] GMF: `http://www.eclipse.org/modeling/gmf`.



(a) Tree-based editor (b) Textual editor (c) Graphical editor

Figure 8: Generated default editors

the Equipment, Workstation, Proxy, DB and CTSM components. The type of the latter one (CTSMType in figure) is defined using the idealized fault tolerant component model integrated in Darwin. Network is a software connector; this is possible thanks to the customization of Darwin we made (third scenario). As can be seen in Figure 8.c, Darwin and BPMN are presented via two different views. (DarwinFT+BPMN)$_{cc}$ enables to assign software components of the designed SA to developer teams. In particular, let us assume to have two developer teams (namely, A and B), a system engineering team and a testing team. The dotted lines graphically show how Darwin components are associated to the BPMN tasks and pools describing the activities assigned to each team (i.e., Proxy, DB, and Network are assigned to the system engineering team, Equipment and CTSM to the development team A, and Workstation to the development team B). Even if these relationships are not graphically rendered, they exist in the IECS model and can be accessed through the Properties panel of both graphical editors. It should be noted that the current IECS model can be manipulated by the original Darwin tool by applying a chain of migrators generated during each step of the composition phase 4.2; a model-to-text transformation (developed in [23]) completes the bridge towards the Darwin tool.

5.2 Advantages and disadvantages

BYADL successfully addresses the main emerging requirements of next generation ADLs. More precisely, the composition mechanism presented in Section 4.1, satisfies requirements **R1**, **R2**, and **R3** since they provide mechanisms to extend the ADL with *Domain specific concerns*, with new *Architectural views*, and with *Analysis notations*, respectively. Furthermore, the proposed operators are used also to define relationships between software architecture and development processes and methodologies, thus addressing requirement **R4**. The requirement **R5** about interoperability is inherited by **DUALLY** [23], and finally the extended (and even extensible) ADL inherits the BYADL tool (with its textual and graphical editors, extensibility mechanisms, interoperability features, and migrators), thus satisfying requirement **R6**.

Extensions in BYADL are defined in an ADL-independent manner and they are collected in libraries, so to be reused for further extensions of (even different) ADLs. It is important to highlight that the approach promoted by BYADL is incremental, so that software architects are able to extend and customize their ADL whenever required. This is particularly important in practice since often the characteristics of "optimal" ADL (for instance as required by the considered non-functional aspects) may change during the architecting phase [27]. However, the proposed approach suffers of the following limitations:

- The semantics of the ADL is defined by means of relationships with A_0. Therefore, a precise semantics cannot be expressed for elements that are not in A_0. For instance A_0 does not explicitly handle hardware devices. Consequently hardware components should be typed as generic components. However, A_0 has been defined in [23] as extendible and **DUALLY** provides also extension mechanisms for it. In this way A_0 can be extended with domain specific concerns, thus enabling a more precise semantics for the considered domain (i.e., A_0 can be extended with hardware aspects and with explicit hardware components). By the way we plan to investigate more powerful and formal ways to provide semantics;
- There is no evidence that the defined operators are enough for extending any existing ADL. New operators can be incrementally added within the BYADL framework in a modularized way. Technically, adding a new operator means to (i) extend the `Operator` metaclass in `Composition Metamodel`, (ii) update the *Composition2Ecore* transformation with the execution logic of the added operator and (iii) update the BYADL HOTs in order to generate migrators reflecting also the logic of the added operator;
- As said before, the generation of graphical editors is in a prototypical stage and it is not able to automatically solve every possible conflict that may arise. This aspect needs further investigation. An important aspect that should be considered is that software architects associate to graphical elements a semantics. When inheriting a graphical element we inherit also this implicit semantics that must conform to the one associated to the element by the tool;
- The current status of BYADL only allows the creation of new ADLs by extending existing ones. The creation of a new ADL from scratch is not properly supported and investigated. In this case A_0 could play an interesting role by providing a minimal and generic ADL that can be extended as needed.

6. CONCLUSION AND FUTURE WORK

In this paper we presented BYADL, a framework for developing a new generation of ADLs. As we have shown throughout the paper, the software architecture world is changing and consequently there is the need of a new generation of architecture modeling approaches. BYADL exploits model-driven techniques and allows a software architect to define its own new generation ADL by starting from an existing ADL, and i) adding domain specificities, new architectural views, or analysis aspects, ii) integrating the ADL with development processes and methodologies, and iii) customizing the ADL by fine tuning it. The incremental extension and customization of a real ADL shows the use in practice of BYADL for defining a new ADL starting from an existing one.

As next steps, we aim at addressing the limitations highlighted in Section 5.2. We plan also to further investigate the role of the migrators. More precisely, the challenge is to investigate how to provide means to identify the parts of the software architecture that are affected by changes made by native ADL tools. In fact this enables software architects to understand if results of performed analysis remain valid even after changes or if the analysis must be re-performed. Moreover, we plan to work on quantifying the effectiveness of our approach by applying it on real-sized case studies.

Acknowledgments

This work is partly supported by the Italian PRIN d-ASAP and FIRB ArtDeco projects. The authors would like to thank Rich Hilliard for his valuable support and comments. We would like to thank also Antonia Bertolino, ISTI CNR and Scuola Superiore Sant'Anna for having hosted us after the earthquake we had in L'Aquila; their hospitality has been crucial for realizing this work.

7. REFERENCES

[1] J. H. Bae and H. S. Chae. UMLSlicer: A tool for modularizing the UML metamodel using slicing. In *CIT 2008.*, 2008.

[2] J. Bézivin, S. Bouzitouna, M. Del Fabro, M. P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige. A canonical scheme for model composition. In *Model Driven Architecture - Foundations and Applications*, 2006.

[3] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *LNCS, Vol. 3599*, 2005.

[4] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC 2008*, 2008.

[5] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In *ICMT'09*, 2009.

[6] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3), 2006.

[7] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *TOSEM*, 14(2), 2005.

[8] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. A highly-extensible, xml-based architecture description language. In *Proceedings of WICSA '01*, 2001.

[9] D. Di Ruscio, H. Muccini, A. Pierantonio, and P. Pelliccione. Towards weaving software architecture models. In *MBD-MOMPES '06*, 2006.

[10] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *SoSyM*, 7(4), 2008.

[11] G. Edwards and N. Medvidovic. A methodology and framework for creating domain-specific development infrastructures. In *ASE'08*, 2008.

[12] P. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis and Design Language (AADL) standard. In *RTAS*, 2003.

[13] G. Ferreira, C. M. Rubira, and R. de Lemos. Explicit Representation of Exception Handling in the Development of Dependable Componet-based Shystems. In *HASE01*, 2001.

[14] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON'97*, 1997.

[15] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Soft. Eng. and Know. Eng.*, volume 2, 1994.

[16] ISO. Fourth working draft of Systems and Software Engineering – Architectural Description (ISO/IECWD4 42010). Working doc.: ISO/IEC JTC 1/SC 7 N 000, 2009.

[17] C. Jeanneret, R. France, and B. Baudry. A reference process for model composition. In *AOM '08*, 2008.

[18] S. John. Architecture Description Markup Language (ADML): Creating an Open Market for IT Architecture Tools. Technical report, The Open Group, 2008.

[19] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proceedings of the MTP, Workshop at MoDELS 2005*, 2006.

[20] P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. *QoSA*, 2006.

[21] A. Lédeczi, G. Nordstrom, G. Karsai, P. Völgyesi, and M. Maróti. On metamodel composition. In *CCA '01*, 2001.

[22] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6), 1996.

[23] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. *IEEE TSE*, 36(1), 2010.

[24] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. Moving architectural description from under the technology lamppost. *Infor. and Software Technology*, 49, 2007.

[25] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1), 2000.

[26] Object Management Group. OMG/Unified Modelling Language(UML) V2.0, 2004.

[27] P. Pelliccione, P. Inverardi, and H. Muccini. CHARMY: A framework for designing and verifying architectural specifications. *IEEE TSE*, 35(3), 2009.

[28] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. In *SIGSOFT Software Engineering Notes*, volume 17, 1992.

[29] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In *GPCE'03*, 2003.

[30] R. A. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *VLDB '2003*, 2003.

[31] J. Sanchez Cuadrado and J. G. Molina. A model-based approach to families of embedded domain-specific languages. *IEEE Trans. Softw. Eng.*, 35(6), 2009.

[32] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[33] T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger. Model integration through mega operations. In *MDWE*, 2005.

[34] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2009.