Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies

Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, Damien A. Tamburri

Technical Report TRCS 001/2009
Providing Architectural Languages and Tools
Interoperability through Model Transformation Technologies

Ivano Malavolta, Henry Muccini,
Patrizio Pelliccione, Damien A. Tamburri
Dipartimento di Informatica, Università dell’Aquila, Via Vetoio, 67100 L’Aquila,
{ivano.malavolta,muccini,pellicci,damien.tamburri}@di.univaq.it

Abstract

Many architectural languages have been proposed in the last fifteen years, each one with the chief aim of becoming
the ideal language for specifying software architectures. What is evident nowadays, instead, is that architectural
languages are defined by stakeholder concerns. Capturing all such concerns within a single, narrowly focused notation
is impossible. At the same time it is also impractical to define and use a “universal” notation, such as UML. As a
result, many domain specific notations for architectural modeling have been proposed, each one focussing on a specific
application domain, analysis type, or modeling environment. As a drawback, a proliferation of languages exists, each
one with its own specific notation, tools, and domain specificity. No effective interoperability is possible to date.
Therefore, if a software architect has to model a concern not supported by his own language/tool, he has to manually
transform (and eventually keep aligned) the available architectural specification into the required language/tool.

This paper presents DUALLY, an automated framework that allows architectural languages and tools inter-
operability. Given any number of architectural languages and tools, they can all interoperate thanks to automated
model transformation techniques. DUALLY is implemented as an Eclipse plugin. Putting it in practice, we apply the
DUALLY approach to the Darwin/FSP ADL and to a UML2.0 profile for software architectures. By making use of
an industrial complex system, we transform a UML software architecture specification in Darwin/FSP, we make some
verifications by using LTSA, and we reflect changes required by the verifications back to the UML specification.

Index Terms

D.2.11 Software Architectures, D.2.12 Interoperability, D.2.11.b Domain-specific architectures, D.2.2 Design Tools
and Techniques, Model Transformations.

I. INTRODUCTION

Up to the present day, many languages for specifying and analyzing software architectures have been proposed
(e.g., [38], [20], [41]) and historically classified into two generations [36]. A “first generation” of Architecture
Description Languages (ADLs) going from 1990 to 2000, had the main purpose to design an ideal ADL [21] whose
chief aim was to enable support of components and connectors specification and their overall interconnection [40], [21], as well as composition, abstraction, reusability, configuration, heterogeneity, and analysis [43]. Later on, during the “second generation”, going from 2000 up to today, new requirements emerged, and new ADLs have been proposed to deal with more specific features [10], [41], such as configuration management, distribution, and product line modeling support.

As a result, a proliferation of architectural languages can be noticed today\(^1\), each characterized by slightly different conceptual architectural elements, different syntax or semantics, focusing on a specific operational domain, or only suitable for different analysis techniques. As noticed in [36], one of the main reasons for such a long list of architectural languages is related to stakeholder concerns: a notation has to adequately capture design decisions judged fundamental by the system’s stakeholders, thus the notion of software architecture has been expanded and notations as well as approaches for modeling software architectures have themselves continued to evolve. Stakeholder concerns are various, ever evolving, and adapting to new environment requirements; hence it is impossible to capture all such concerns with a single, narrowly focused notation. Instead of a unique language for specifying software architectures, we must therefore accept the existence of domain specific languages for Software Architecture (SA) modeling, therefore considering each ADL aimed at solving specific stakeholder concerns (e.g., architectural styles, real-time constraints, data-flow-architectures, code generation, etc.). Even the adoption of UML for modeling architectures (e.g., [37], [18]) is biased by different concerns: a number of UML profiles and extensions have been proposed for modeling different concerns, thus increasing even more the proliferation of architectural languages. These extensions cannot fully represent all aspects/features of every ADL and on the other side, as already claimed in [11], [18], [38], is impractical to have a “universal” notation.

Furthermore, when architecting a software system several significant decisions must be taken. Some of them can not be taken at the beginning of the architecting phase and are made iteratively by analyzing and studying an architectural prototype [39] often sketched by the architect starting from the nebulous dark set of constraints, requirements and ideas. Moreover, typically it is not immediately evident what kind of non-functional aspect must be considered in a system. This implies that the choice of the best ADL (as required by the considered non-functional aspect) may change during the architecting phase. Whether the architecture has been already modeled in a specific ADL, an interchange language is required or the architecture must be rewritten in the most suitable ADL.

Very limited interoperability possibilities among tools and notations exist in practice. In Section V, among the existing ADLs we will extract and analyze those whose goal we consider close to ours in terms of effort invested in extensibility or interoperability mechanisms. The Acme initiative [20] is famed for being one of the very first technologies to tackle the problem of architectural data interchange. A number of reasons drifted Acme somewhat away from its initial goal, the main of which is that Acme obliges to explicitly write end-to-end transformations among each pair of ADLs; see Section V for a detailed discussion on Acme as an interchange language. This brings no evident advantage in having an intermediate language. Therefore, assuming an SA specified in, e.g., a UML

\(^1\)In an ongoing study we are conducting, we counted more than 50 ADLs proposed both from academia and industry.
profile, needs to be model checked, a different architectural specification needs to be produced, e.g., in Darwin/FSP, since UML tools does not support model checking. As soon as the model-checking analysis in Darwin/FSP [33], [34] requires a modification in the SA, consistency between the new Darwin specification and its UML counterpart has to be manually maintained.

These considerations led us to propose DUALLY, a framework to create interoperability among ADLs themselves as well as UML. As conceptually shown in Figure 1, DUALLY permits transformation of concepts of an architectural model M1 into semantically equivalent concepts in the architectural model M2. Each Mi conforms to its MMi that is a meta-model or a UML profile. Therefore, DUALLY works at two abstraction levels: meta-modeling (upper part of Figure 1), and modeling (lower part of Figure 1).

At the meta-modeling level, model driven engineers provide a specification of the architectural language in terms of its meta-model or UML profile. They then define a set of mappings so as to semantically relate architectural concepts in MM1 with the equivalent elements in MM2.

At the modeling level, software architects will specify the SA using their preferred ADL or UML-based notation. DUALLY allows the automatic generation of model-to-model transformations which enable the software architect to automatically translate the M1 specification (written according to MM1) into the corresponding M2 model (conforming to MM2) and vice versa.

As it can be noticed in Figure 1, the semantic mappings (and its corresponding generated transformation) relates MM1 to MM2 (as well as M1 to M2) passing through what we refer to as $A_0$. $A_0$ represents a semantic core set of modeling elements, 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. The why and how of $A_0$ will be discussed in Section II.

DUALLY is implemented as an Eclipse plugin. MMi meta-models are expressed in Ecore. MMi profiles can

---

2This figure will be refined in the following of this paper.
be realized using any UML tool which exports in UML2\(^3\), the EMF-based implementation of the UML 2.0 meta-model for the Eclipse platform. \(A_0\) is represented as a meta-model. Transformations among meta-models/profiles and model-to-model transformations are implemented in the context of the AMMA platform [4]. Model-to-model transformations are then automatically instantiated, 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 \textsc{Dually}.

We argue that \textsc{Dually}, taking into account the evolutions in the context of software engineering (e.g., the success of model-driven technologies), can succeed where previous works failed for various reasons both conceptual and technological.

Conceptual side:

- The dominance of UML and the proliferation of UML-based ADLs, which undoubtedly share more in common than early ADLs (particularly the shared meta-model in MOF) induced \textsc{Dually} to enable the transformation among formal ADLs and/or UML model-based notations.
- The evolved notions of software architecture (e.g., [36]) demonstrate that the core set of architectural elements defined by previous approaches is somehow obsolete. Moreover, recently there is a proliferation of ADLs with domain specific features. \textsc{Dually} considers both a state-of-the art core set of architectural elements and extensibility mechanisms to augment the core with domain specific and new concepts.
- The choice of the best ADL (as required by the considered non-functional aspect) may change during the architecting phase. In fact, software architects starting from the nebulous dark set of constraints, requirements and ideas typically manually sketch an architecture prototype making often suboptimal solutions. This calls for architectural design iterations [39]. \textsc{Dually} avoids, whether the architecture has been already modeled in an ADL, to redesign the software architecture in the different and required analysis-specific ADL.

Technological side:

- \textsc{Dually} exploits technologies for model-driven approaches that are the result of the recent explosion of interest in model-driven engineering. Model transformation languages, tools, and techniques are more powerful with respect to techniques used in the past, such as normal programming languages or XSLT\(^4\) that may suffer from code maintainability and scalability issues [9]. Further on, checking mathematical properties like correctness or completeness of transformations based on common programming languages is very difficult since they lack a strong mathematical basis.
- \textsc{Dually} works at two abstraction levels, providing a clear separation between the role of model driven experts (i.e., the technical stakeholder) and that of software architects (i.e., the final users). The model transformation engine is completely hidden to a software architect, making \textsc{Dually} extremely easy to use.
- Software architects can continue using familiar architectural notations and tools even reusing legacy ar-

\(^3\)For a list of UML2-compatible UML Tools, please refer to http://wiki.eclipse.org/MDT-UML2-Tool-Compatibility.

\(^4\)XSL Transformations (XSLT) Version 2.0, W3C Recommendation: http://www.w3.org/TR/xslt20/.
chitectural models. The DUALLYzation of legacy, textual specifications is also possible via a preliminary transformation step producing models from textual specifications; such a preliminary step can be performed automatically [6] with DUALLY providing the means to integrate such a technology (see Section II).

- DUALLY enables both languages and tools interoperability.
- The semantic links among two architectural notations are defined once, and reused for each possible model.

The rest of the paper is structured as follows: Section II presents DUALLY and its main principles. Section III presents the technology used to engineer DUALLY and discusses its implementation. Section IV shows a complex example of the integration of a UML profile and Darwin/FSP within DUALLY and our experience in translating a UML model into a Darwin/FSP specification. Section V describes related work, while Section VI evaluates DUALLY and provides consideration on its implementation and usage. Section VII concludes the paper and envisions future research directions. Finally, a list of acronyms is provided in Appendix.

II. DUALLY: THE FRAMEWORK IN CONCEPTS

Figure 2 shows a high level conceptual view of DUALLY. This picture also puts in evidence the main purpose of DUALLY, that of allowing different formal ADLs and/or UML-based languages and tools for software architecture modeling to interoperate. Deeper down to the conceptual details, Figure 1 clearly represents a full instance of two branches from Figure 2’s more general view.

DUALLY aims at support both languages and tools interoperability.

As far as concerns tool interoperability it is important to note that the Eclipse platform upon which we act is fully XMI compatible and import/export capable. Stemming from this key aspect of the technology we base around, it can be argued that all the tools capable to import/export from/to the Eclipse EMF in an XMI compatible way are automatically integrated with DUALLY. Many ADLs exist which support such a feature, e.g., AADL and xADL that store their resident format in XMI, ACME which retains a full UML counterpart, and all UML-based notations.

Tools that are not capable to import/export from/to the Eclipse EMF in an XMI compatible way (e.g., textual notations like Darwin) need a preliminary step to translate their resident format into a model driven compatible one. DUALLY supports this preliminary step by providing an extension point for additional transformations importing/exporting from/to these technologies. Such transformations may be designed via JET (Java Emitter...
Templates\textsuperscript{5}\textsuperscript{5} templates, Java classes or ATL modules. To provide a solution for the cases above, DUALLY’s extension point provides the following facilities:

- a custom DUALLY import/export menu entry;
- an import (export) dialog wizard ending up with the automatic execution of the proper transformation.

A number of approaches are emerging that may be exploited to automate the generation of both the meta-model and the import/export transformations starting from the language’s BNF (Backus-Naur Form) grammar or DTD (Document Type Definition). For instance the work in [6] presents an effective methodology to map models developed in “legacy” DSLs (which ADLs can be seen as a particular form of) to MDA compliant models. This methodology is provided with a full-fledged supporting framework.

In the following of this section we focus on languages interoperability that is more challenging and interesting, assuming that these preliminary steps (if needed) have already been taken care of.

A number of key elements can be distinguished in both Figure 1 and Figure 2: meta-models, UML profiles, semantic links, $A_0$, architectural models and model transformations. The two main users intended for DUALLY are meta-modeling experts, who will act mainly on the first three elements, and software architects, acting mainly on the last two elements. While being the basic reference for new projects, $A_0$ also 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 DUALLYzation. As clearly shown in Figure 2, for DUALLY’s realization we chose a “star” architecture. $A_0$ is placed in the center of the star, while DUALLY’s transformation engine is in charge of maintaining the transformation network. The depicted star architecture and the devised branches will enable multiple technology intercommunication. In Section II-A we provide details on $A_0$, its structure and meaning as well as a thorough explanation of its role within our transformation engine. The DUALLY transformation engine is explained in Section II-B.

A. DUALLY Core Set: $A_0$

DUALLY inherits from both xArch [10] and Acme [20] the idea of identifying a core set of architectural concepts, hereafter called $A_0$ (see Figure 1). The main purpose of $A_0$ is to provide a centralized set of semantic elements with respect to which relations must be defined. Considering that many languages need to be related together (see Figure 2), instead of creating a point-to-point relationship among all languages, a linear relationship between the selected language and $A_0$ is created, thus reducing the number of connections needed. What may indeed happen is that two ADLs, say $ADL_i$ and $ADL_j$, share some domain specific concepts and these are not contemplated in $A_0$. In this case, $A_0$ should be extended by means of the DUALLY extension mechanisms explained in the following, in order to include domain specific concepts.

\textsuperscript{5}JET home page: http://www.eclipse.org/modeling/m2t/?project=jet.
The selection of the elements within $A_0$ was mainly guided by the principle of maintaining our base notation 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). The selection phase has been performed by studying architectural languages with purposes similar to DUALLY (e.g., xArch, xADL, Acme), relevant papers (e.g., [36], [14]) and UML. We exploited and inherited features we judged satisfactory, overcoming identified limitations (e.g., xArch is extensible but makes use of XML, UML is very expressive but is ambiguous, etc.).

$A_0$ acts as a central pillar of the model transformation network. It is the base language that every technology may use to keep “aligned” with any other integrated one. All the passages between technologies are referenced through $A_0$ and this makes sure that architectural information is maintained aligned at every step of the cycle: modeling $\rightarrow$ transformation $\rightarrow$ modeling and so on.

Fig. 3. $A_0$ meta-model

Here in the following we present in further detail the $A_0$ meta-model (shown in Figure 3), by providing a
description of its main meta-classes:

**Architecture**: a collection of components and connectors instantiated in a configuration. It may contain also a *TypesSpecification* defining a set of architectural types, a *Behavior* that specifies a high-level description of dynamic aspects of the system and *SAinterfaces* representing points of interaction between the external environment and the architecture being modeled.

**SoftwareArchitecture**: a specialization of *Architecture* representing exclusively the software part of the system being designed.

**SAcomponent**: a unit of computation with internal state and well-defined interface. A component can contain a sub-architecture consisting of lower-level components, connectors and their configuration. It can contain a behavioral model. Components interact with other architectural elements either directly or through *SAinterfaces*.

**SAconnector**: represents a software connector containing communication and coordination facilities. A connector can contain a behavioral description, a set of interfaces defining its points of interaction with other architectural elements and a sub-architecture in the same way of *SAcomponents*.

**SAinterface**: specifies the interaction point between an *SAcomponent* or an *SAconnector* and its environment. It is semantically close to both UML concepts of interface and port, it may contain a set of *Properties* and it can play the role of a link end, i.e., architectural channels can be attached to it, further it can have either input, output or input/output direction.

**SArelationship**: its purpose is that of delineating general relations between $A_0$ architectural elements; it can be either bidirectional or unidirectional. It is connected to any *ConnectableElement*, making the configuration of $A_0$ models as generic as possible.

**SAchannel**: a specialization of an *SArelationship* representing a generic communication mean; it supports both unidirectional and bi-directional communication, both information and control can be exchanged. Because inconsistencies in the design too often cause design flaws, the *SAchannel* is aptly left general enough so that further specification is categorically needed at a later design time.

**SAbinding**: relates an *SAinterface* of a component to an *SAinterface* of one of its inner components. We do not define intentionally the nature of the relationship between the two *SAinterfaces*, it would be interpreted as an equivalence relationship (i.e., the two interfaces are precisely the same), a message passing or a service call. Semantics can be either inferred from the direction of the *SAinterfaces* or defined through future extensions of the meta-model. *SAbinding* is semantically very similar to the UML Delegation Connector.

**SAtype**: type specification can be as important as any kind of topological or architectural specification; *SAtypes* define architectural types, so any architectural element can potentially be an instance of a particular *SAtype*. Each *SAtype* can contain a set of properties and the behavior of its instances, its internal structure is not specified. *SAtype* is a specialization of *Type*.

**SAstructuredType**: each *SAstructuredType* is a specialized *SAtype* that can contain also a definition of its sub-architecture, i.e., a set of architectural elements defining its internal structure in terms of components,
connectors and their interconnections. Internal elements are interpreted as default elements contained into every instance of the corresponding SStructuredType.

**Behavior**: represents the behavior of a component, a connector, an interface or a system architecture. It is an abstract meta-class that plays the role of a “stub” for possible extensions of the meta-model representing dynamic aspects of the system (e.g., finite state processes, labeled transition systems, state diagram, UML-like sequence diagrams).

**Development**: represents the direct relation between the architectural and the technological aspects, such as the process that will be used to develop the system, the distribution of tasks within the developing teams, the programming languages used to develop a certain component, etc. Since the Development meta-class is aggregated to Element, development details can be associated to every element of the A0 model. Development is an abstract meta-class and its realization is left to future extensions of the A0 meta-model.

**Business**: an abstract meta-class to be specialized via future extensions of A0 and represents the link to business contexts within which software systems and development organizations exist. Its possible realizations may include a system marketing strategy, costs, product-line issues (e.g., variants and options) or generic issues regarding how the developing organization is related to the system. In the same way as Development, it is aggregated to Element in order to provide business information to every element of A0 models.

**Type**: its obvious purpose is that of specifying any aspect concerning a data type definition. Types can be associated to any other architectural element of A0. This is particularly useful for defining semantics within specific contexts. Types can be related through a “supertype-subtype” relationship in order to give the possibility to create hierarchical type systems within the DALLY framework.

**TypesSpecification**: type specification diagrams contain the specification of types to correctly and orderly put them in relation or simply to keep them on an architecturally separate level. Defining a TypesSpecification element within a software architecture means that such architecture may include instances of the types defined in the TypesSpecification.

**Property**: a generic feature of a PropertyHolder, i.e., every element that is allowed to contain properties. Each Property is defined by a name, a type (PropertyType) and a value (PropertyValue). Since the Property meta-class is aggregated to Element, properties can be associated to every element of the A0 model.

**PropertyType**: an enumeration containing (i) common primitive datatypes, (ii) a “reference” type that represents a reference to another element of the model and (iii) a generic type called “any” inspired by the OclAny type in the OCL language, which is considered the supertype of every PropertyType.

**PropertyValue**: a meta-class defining the value of a Property. For each PropertyType it contains an attribute conforming to the equivalent primitive type. Therefore, each PropertyValue is bound to its corresponding real value; this overcomes one of the weaknesses of Acme in which the value of a property consists only in a string that exposes models to ambiguity and expressivity problems. Moreover, thanks to the extensible nature

---

of the $A_0$ meta-model, the PropertyValue meta-class can be extended in order to contain OCL expressions providing more formal (and analyzable) means to define architectural properties.

**Group**: a logical grouping of any element of $A_0$, and it can contain architectural elements, properties, other groups and so on. The semantics of group within the $A_0$ meta-model has been strongly inspired by the xArch concept of group. It differs in the relationship with the group members only. xArch defines a group as a containment, while in $A_0$ we specify it as a plain association relationship. We believe that this provides more generality to the modeling framework since $A_0$ groups can be considered as pure logical groupings, avoiding the constraint that each member of a group must be contained into it. This also avoids the problem that would arise if a model element belongs to more than one group.

During the development of the $A_0$ meta-model, the issue of future extensions has been taken into account. With the term “extension” we mean annotation or specialization of each element of $A_0$, addition of new elements (for instance a new diagram used for specifying the behavior or new elements in a diagram), as well as composition and association relations between elements of $A_0$.

We intentionally defined many minor abstract meta-classes with very generic semantics. The rationale behind this choice is that of providing different granularity of “placeholders” to which meta-classes of future extensions can “hook-up” to.

**Structural abstract meta-classes**

**Element**: the root of the meta-model, i.e., every meta-class specializes Element either directly or indirectly. Every Element has a name and a description field.

**TypedElement**: represents every modeling element that may have a type; so it has a reference to the Type meta-class in order to relate the TypedElement to the Type it is instance of.

**PropertyHolderElement**: represents every $A_0$ model class that can be associated with a set of properties.

**ConnectableElement**: represents all the elements that can play the role of link ends; they are the elements to which a link can be attached to.

**ArchitecturalElement**: the superclass for all $A_0$ elements that can be contained into a software architecture.

**Component**: defines computational elements and data store elements offering services through a predefined set of InteractionPoints. Components communicate with other system elements via Links.

**InteractionPoint**: is contained by Components and defines a generic point of interaction between a system element and its environment.

**Link**: an abstract connection between model elements; semantics can be added to it through specialization, e.g., to represent communication channels, service requests, and control/data flow.

$A_0$ is designed to be a generic, fully-expressive meta-model in order to ease the mapping of ADL-specific concepts to $A_0$ elements; the well-formedness of the $A_0$ meta-model is ensured by a set of constraints defined
in OCL. For example, constraints ensure that SAbindings relate SAinterfaces with same direction, that an SAchannel cannot have a sub-architecture, and so on. For the sake of readability we omitted the description of such constraints. If the need for a more “disciplined” $A_0$ arises (for instance in order to define architectures with specific styles or configuration rules), additional OCL constraints can be added to every entity of the meta-model. Additional constraints can be defined straightforwardly exploiting the extensible structure of the meta-model. In the following we describe the extensibility mechanisms of $A_0$.

1) Extensibility mechanisms of $A_0$: The current release of $A_0$ envisions a large number of extension points. Business, behavioral aspects, and domain modeling extensions are natively supported, while any other classes can be extended through the proposed mechanisms. If we need to define an extension that cannot be identified as an extension of existing meta-classes, it is always possible to extend $A_0$ by extending its root meta-class, i.e., Element (please refer to Figure 3). The core itself is now frozen and allows for extensions only in certain ways so that the essential elements in the kernel remain untouchable: in this way, backward compatibility is ensured.

The different kinds of extensions supported by DALLY are:

- Annotation of each element of $A_0$ or its previously defined extensions: for instance probability measures associated to elements in order to perform performance evaluations.
- Specialization of each element of $A_0$ or its extensions: for instance if we are interested in adding hardware components we can extend the Component meta-class.
- Addition of new elements: for instance a new diagram used for specifying the behavior or new elements in a diagram.
- Addition of composition and association relations between elements: it is possible to add composition and association relations among elements of $A_0$ or its previously defined extensions.

These kinds of extensions are realized by means of the inheritance mechanism. Each element of the meta-model can be extended. For this reason $A_0$ contains abstract meta-classes for behavior, business, etc., that define the “type” of the extension. Composition and association relationships can be added among each element of the extension and each element of $A_0$ in order to integrate the parts. An extension of $A_0$ with the aim to add the behavior can extend the Behavior meta-class. In the following we show how to realize that in practice by presenting an example of extension that gives the possibility to associate behavioral state-based descriptions to architectural elements.

This is done by a two-step process: (i) definition of StateDiagramMM, a basic state diagram meta-model and (ii) relating elements of StateDiagramMM to abstract meta-classes of $A_0$. For the sake of simplicity we defined a very simple and generic state diagram meta-model; its semantics is taken from [23]. The lower part of Figure 4 shows the StateDiagramMM, which contains a main class called StateDiagram composed of states and transitions. Each state can either be generic, initial or final and may be hierarchically structured, i.e., it contains the definition of a sub state-diagram. Each transition may be labelled and a guard may check whether it can occur or not.

It should be noted that we defined a simple meta-model since the aim of this extension is to give an example
on how to extend $A_0$ without the aim to provide a complete behavioral notation for software architectures. The StateDiagramMM can be completed with a variety of state-based constructs, or additional extensions may specialize $A_0$’s Behavior meta-class with other behavioral notations like activity diagrams, sequence diagrams, Finite State Process (FSP) specifications [29], etc.

The upper part of Figure 4 shows how we related the StateDiagramMM meta-model to $A_0$ (note that for presentation purposes we only show parts of the $A_0$ meta-model that are relevant for the extension). As expected the StateDiagram meta-class extends $A_0$’s Behavior, thus it has also a name, a description and can be associated to all the architectural elements that can have a behavioral description, e.g., SACOMPONENT and Sainterface. Moreover, State and Transition extend the PropertyHolderElement meta-class so that software architects are allowed to associate properties to them. The case study presented in Section IV will make use of this extension of $A_0$.

In the next section we will explain how the $A_0$ meta-model interacts within our technology to achieve DUALLY’s goal. The previously identified elements from Figure 1 will be analyzed and their interaction points within DUALLY will be defined.
B. **DUALLY Model Transformations**

To enable the possibilities exposed in the previous sections, precise and powerful transformations must be devised at each level, to “fill” the interconnections existing in Figure 1: model transformations as well as meta-model to meta-model links must be somehow provided. The star architecture permits an agile and easy integration of more and more technologies as the need rises. As previously stated, the transformation system we deliver is made of a series of low level model-to-model transformations that enable information migration among model instances. These model-to-model transformations are constructed automatically executing higher order transformations (i.e. transformations taking other transformations as input or producing other transformations as output). Such transformations utilize information provided by the meta-model to meta-model links that “physically” integrate the DUALLYzed technology, in the process of creating the lower level model-to-model-transformations.

In the following we describe meta-model to meta-model links and higher order transformations to create communication among models. It may be noted that only properly skilled meta-modelers take part in the specification phase (only involving the software architects for semantically define the links among the ADL to be DUALLYzed and A₀), while software architects work at the modeling level.

**Meta-model to meta-model Links: Bringing meta-models together**

What we earlier called meta-model to meta-model links are the main branches that stem out of A₀ and they constitute a mechanism to provide references and semantic relations among elements at a meta-model level and A₀; using a terminology more specific to our technology, these semantical links can be called weaving models. As previously stated, weaving models define these links so as to later use them to generate lower level transformations. These links are defined once, during the process that integrates a certain technology within DUALLY; their role is in essence that of constituting the bridge between A₀ and the DUALLYzed technology.

A number of methods to specify and construct weaving models are currently being explored and exploited, but conceptually, weaving models conform to a given weaving meta-model, and can be defined either manually or by utilizing ad-hoc scripting languages. DUALLY defines its own weaving meta-model and therefore enables the definition of its own weaving models, which will constitute the main step to be carried out in order to integrate a certain technology.

**Higher order transformations: Creating Lower Level Communication**

A model transformations can be considered a model. Recent research efforts enhanced the concept of model transformation, promoting it into that of transformation model [2]. Just as a model can be created, modified, and augmented through a transformation, a transformation can be regarded as a model and therefore, it can itself be instanced, modified and so on. Aptly constructed transformations can carry out this particular task: thanks to higher order transformations specified and developed upon DUALLYzation, model level transformations are automatically obtained. These transformations can be executed directly. Uniting all these transformations a network is obtained; this will enable full, level-wide communication.

1) **Correctness of the transformations:** Our goal is to guarantee the correctness of the transformations. The simplest notion of correctness is syntactic correctness: given a well-formed source model, can we guarantee that
the target model produced by the transformation is well-formed? We can guarantee this syntactic correctness since DUALLLy contains mechanisms to check if a model conforms to its meta-model.

A significantly more complex notion is semantic correctness: does the produced target model have the expected semantic properties? Each DUALLyzed ADL has its own transformation rules. For this reason we cannot define the properties that the model transformations satisfy. What we can do is to exactly define what kind of properties should be satisfied.

When dealing with different models, making changes to one model may require to propagate the changes to other models. To avoid inconsistencies, changes to the target model have to be reflected back to the source model. This is the process known as Round-Trip Engineering (RTE) [24]. The main difficulty of RTE is that in general the transformations are neither total nor injective, i.e., some elements of the source model do not have a correspondence in the target model, and vice versa. Referring to [24] we report some definitions with the aim to define properties that hold also for transformations that are neither total nor injective. Before reporting the most interesting definitions we provide auxiliary definitions: i.e., model transformation in Definition 1, injectivity in Definition 2, surjectivity in Definition 3, and bijectivity in Definition 4.

**Definition 1.** A model transformation \(\text{trans}: \mathcal{M}_S \rightarrow \mathcal{M}_T\) is a mapping of elements of \(\mathcal{M}_S\), which is the source meta-model, to elements of \(\mathcal{M}_T\), which is the target meta-model.

**Definition 2.** Let \(\text{trans}: \mathcal{M}_S \rightarrow \mathcal{M}_T\) be a model transformation. \(\text{trans}\) is injective if \(\forall S, S' \in \mathcal{M}_S\), if \(\text{trans}(S) = \text{trans}(S')\) then \(S = S'\).

**Definition 3.** Let \(\text{trans}: \mathcal{M}_S \rightarrow \mathcal{T}_T\) be a model transformation. \(\text{trans}\) is surjective if and only if \(\forall S \in \mathcal{M}_S\), \(\exists T \in \mathcal{M}_T\) such that \(\text{trans}(S) = T\).

**Definition 4.** Let \(\text{trans}: \mathcal{M}_S \rightarrow \mathcal{M}_T\) be a model transformation. \(\text{trans}\) is bijective if \(\text{trans}\) is injective and surjective.

In order to deal with transformations that are neither total nor injective, [24] introduces the definitions of *relevant source models* and *relevant target models*, i.e., the parts of the source and target models that are involved by the considered transformation and the function \(\text{strip}\) that maps elements to their stripped-down relevant source and target models. Once defined the relevant source and target model it is possible to define synchronization between two models. The following definition is taken from [24]:

**Definition 5.** Two models \(S\) and \(T\), instances of their respective meta-models \(\mathcal{M}_S\) and \(\mathcal{M}_T\) are synchronized with respect to a transformation \(\text{trans}: \mathcal{M}_S \rightarrow \mathcal{M}_T\) if \(\text{trans}(S) = \text{strip}(T)\).

Informally, two models are synchronized if the relevant part of the target model can be created by applying the transformation to the source model.

When dealing with total and bijective model transformations the mathematical inverse of \(\text{trans}, \text{trans}^{-1}\), can be used to reobtain the source model starting from the target model, i.e., \(\text{trans}^{-1}(T) = S\). Since typical model transformations are neither total nor bijective, we need a relaxed version that requires the inverse to be defined on the domain and range of the source and target model: \(\text{trans}^{-1}_\text{rev}(T) = \text{strip}(S)\).

Actually, round-trip engineering does not aim at recovering lost and unavailable source models, but aims at producing a new source model that when transformed produces the target model with the changes made on it. More
formally:

**Definition 6.** A function \( \text{trans}^{RT}: \mathcal{M}_S \times \mathcal{M}_T \times (\mathcal{M}_T \rightarrow \mathcal{M}_T) \rightarrow \mathcal{M}_S \) is a round-trip transformation if it maps to a new source model \( S' \) the source model \( S \), the target model \( T \) and a target model change \( \Delta_T: (\mathcal{M}_T \rightarrow \mathcal{M}_T) \) such that: \( \text{trans}^{RT}(S, T, \Delta_T) = S' \), where \( S' \) and \( \Delta_T(T) \) are synchronized.

**DUALLY** provides the mechanisms that can be used to define model transformations that satisfy this property.

When translating from \( A \rightarrow B \rightarrow A' \), these mechanisms preserve features that appear in \( A \) and cannot be rendered in \( B \). These features are retrieved when round-tripping to \( A' \). In this way “lost in translation” is avoided. Mechanisms for managing “lost in translation” are explained in Section II-C.

2) **Quality of the implemented model transformations:** Once the model transformations have been defined, they must be implemented to work in the **DUALLY** framework. A transformation process is composed of three parts: source model, target model and transformation itself. Assuming the correctness of the source models, when discussing about quality of the generated transformations we deal with quality of the obtained target models. In **DUALLY**, the target model natively conforms to its meta-model (i.e., the quality of the transformation creating the target model does not count as the model will be, by construction, an instance of its reference meta-model). Obviously, if the meta-model is not accurate and completely correct, this may hinder the transformations quality. Dealing with transformation quality requires to ensure that certain “properties” are preserved by transformations; more precisely, if the source model satisfies a given set of architectural properties then the target model has to satisfy those properties as well. Since we are considering properties that are intrinsically intermingled with a specific system we can only provide methodologies and guidelines to ease the problem’s solution.

Based on [3] there are two main approaches to tackle the problem: a **Generic approach** and a **Transformation-based approach**.

**Generic approach.** As represented in Figure 5.a, two are the input models: the model to test and the model containing the specification of the properties to verify. The **Verifier** component may be implemented as a generic ATL transformation, so we can reuse the AMMA platform set up for our framework. **Diagnostic** is a model containing the outcome of the verification process; it will conform to a self-described meta-model which is represented in Figure 5.b.

The benefits of this approach are: (i) the **Verifier** component is generic (i.e., it is always the same for every property to check) and thus it is maintainable. The properties are expressed through ad-hoc models and they are...
“engineerable”, i.e., many software engineering techniques (e.g., automatic generation of the properties to verify) may be applied to the properties in an MDE fashion. The most relevant drawback of this approach is that a meta-model for the properties’ models must be designed. This is not straightforward because it has to cope with different kinds of properties and because it must be complete in terms of power of expression (i.e., it is not acceptable that a specific property cannot be checked only because the meta-model does not contain the constructs to express it).

Transformation-based approach (Figure 6). This approach was presented and developed in [3].

Fig. 6. Transformation-based approach for maintaining models’ consistency

The input model is only one, i.e., the model being validated. The properties being verified are embedded into conditional OCL-based expressions that guard the rules of an ATL transformation. The Diagnostic is expressed through the same meta-model described in the previous approach. This approach does not “constrain” the properties into models, thus we do not need to develop a meta-model for properties. Possible drawbacks are that for each set of properties (or even for each single property) the architect has to develop an ATL transformation that embeds them. Moreover modifying a property implies the corresponding modification to the transformation that verifies it: the architect is forced to know the ATL language and tracing problems between the specification of the properties and the Verifier component may rise. Future advancements of DUALLY may include such a mechanism in order to verify the stated quality needs.

Finally, a set of model transformation testing approaches such as [32], [31] or [7] may be used to check the generated transformations verifying on their effective correctness. These approaches provide frameworks and algorithms to automatically generate test-models for model transformations, to run a path coverage of the execution of a model transformation, to apply model differencing techniques to model transformation testing and so on. All of these techniques would provide additional confidence on the quality of the generated transformations.

C. Managing the lost in translation

Within the DUALLY framework one of the most important properties to preserve is the synchronization between DUALLYzed notations, i.e., changes made on a specific (generated) model must be propagated back to the others. This is one of the key issues in languages interoperability. Reasoning by example, in Section IV we present a DUALLYzation of a UML-based notation and Darwin/FSP. In this case study we automatically obtain a Darwin/FSP specification from a UML model and perform a deadlock-freedom analysis on the Darwin/FSP specification. Therefore, once the Darwin/FSP model has been refined, as required by the performed analysis, it is crucial to propagate back those changes to the UML model in order to keep the models aligned and consistent.

Various approaches have been recently proposed in order to tackle this problem. In [24] the authors provide a framework to compare current model synchronization approaches, classifying them by the nature of the involved
transformations (i.e., whether they are total or partial, bijective or injective and if the reverse transformations are given or not). Since the DUALLY framework is built upon the MDA infrastructure, all of these approaches can be exploited depending on the assumptions made on the transformations generated from the weaving models. For example, if designers prove (or assume) that the generated transformations are total and bijective, then the corresponding approach may be used. This implies that designers have to analyze the generated transformations and make assumptions on them. Obviously, this is not always possible (e.g., transformations with many manual ad-hoc refinements are hard to classify) and for those cases we devised a basic and generic mechanism to keep models within the DUALLY framework consistent. It is important to note that this mechanism is not intended to be the unique synchronization method in DUALLY. On the contrary, this method represents the default solution to be used when other approaches cannot be applied since no assumptions can be made on the generated transformations. Figure 7 shows how the mechanism works.

Let us suppose that a meta-modeling expert DUALLYzed a generic source notation MMsource and that a model Ms conforming to its meta-model MMsource has been developed; then the software architect executes the generated transformation Msource2Mtarget producing a model Mt. As it can be noticed in Figure 7, only some of the Ms elements have been translated into Mt (i.e., the gray part, technically called the strip \((Ms,Msource2Mtarget)\) [24]) while some other concepts have not been translated.

At this point changes would be made on the Mt model obtaining the model Mt’. Now the need to synchronize Mt’ with the initial Ms model arises. Applying Mtarget2Msource, the software architect obtains only the gray part of the model Ms’, i.e., the part that is aligned with the part matched by the initial transformation (i.e., Msource2Mtarget). This part of the Ms’ model does not contain the unmatched elements corresponding to the white part of Ms in the figure which represents a loss of information. Our mechanism provides the means to automatically store and re-add those elements when closing the round-trip journey, thus obtaining a full-featured
source specification, synchronized with the $M_{t'}$ model.

More precisely, executing the $M_{source2M_{target}}$ transformation the part of $M_{s}$ that is not involved by the transformation is stored in the $MLost$ model. When executing the $M_{target2M_{source}}$ transformation on $M_{t'}$, we obtain a model containing only the elements corresponding to $strip(M_{s}, M_{source2M_{target}})$. Our mechanism allows us to produce a full $M_{s'}$ model that contains also the elements that were in $M_{s} - strip(M_{s}, M_{source2M_{target}})$. We call such set of elements *lost-in-translation* because they are lost during the execution of $M_{source2M_{target}}$.

$MLost$ conforms to a single, generic meta-model called $MMLost$ and represented in Figure 8.

![Lost-in-translation meta-model](image)

Each model conforming to $MMLost$ is composed of a root element containing the URI of the model ($M_{s}$ in Figure 7) and the transformation it has been generated from ($instructed M_{source2M_{target}}$ in Figure 7). The root element is composed of a set of lost-in-translation elements that may be either (i) model elements containing a reference to the corresponding model element or (ii) structural features (i.e., attributes or references to other elements). Each $LostStructuralFeature$ contains a reference to the original model element that owns it. $Ref$ represents a reference to the element lost during the transformation; its attribute $ref$ contains the XMI-ID of the corresponding element. The $Ref$ meta-class can be extended to support different identification mechanisms. For example, the name attribute of the corresponding element could be used as a reference key if XMI-IDs are not preserved during the execution of the transformations. Within the development of the $DUALLy$ framework we extended the ATL transformation engine in order to preserve the XMI-IDs while executing the transformations. Then, the XMI-ID attribute is enough to uniquely identify model elements. This provides also a good level of scalability since software architects do not need to trace models while round-tripping $DUALLy$zed models. The correspondence between model elements is identified by directly referring to their XMI-ID attribute.

From an implementation point of view, we extended the higher-order transformations of $DUALLy$ so that:
• the generated transformation is “instructed” so that it returns as output a target model and an additional model containing the lost-in-translation elements;
• the generated transformation takes as input a source model and a previously created lost-in-translation model and re-adds its elements to the target model.

When executing a higher-order transformation a window dialog appears asking the user if the transformation that will be generated must be instructed or not. In this way the user is allowed to produce three kinds of transformations:
1) **Not instructed**: the model transformation does not take into consideration the lost-in-translation mechanism of DULLY;
2) **Instructed Ms2Mt**: the transformation creates the additional lost-in-translation model;
3) **Instructed Mt2Ms**: the transformation takes as input the additional lost-in-translation model and adds its elements to the target model.

In the following we expose how instructed transformations manage lost-in-translation models within the DULLY framework. Both Ms2Mt and Mt2Ms are instructed by adding an endpoint rule to them, i.e., a rule that is executed at the end of the execution of the model transformation. This gives the possibility to check also elements that belong to the target model and make assumptions on them. Ms2Mt, the transformation producing also a lost-in-translation model, is instructed so that it generates a LostModelElement for each model element that is not matched by any of its rules and sets the corresponding Ref to the XMI-ID of the current not-matched element. Moreover, it creates a LostStructuralFeature for each attribute or reference that is not matched and sets a Ref element containing the XMI-ID of the model entity that owns that feature. Mt2Ms, the transformation that takes as input the auxiliary lost-in-translation model, performs a three-steps task:

1) for each LostModelElement it checks if the parent entity of the lost element in Ms still exist in the target model Ms’. If this is the case a fresh new model element is created looking up the element pointed by the corresponding Ref, otherwise the current LostModelElement is ignored;
2) for each LostStructuralFeature the corresponding attribute or reference is set by copying the value pointed by the corresponding Ref element. If the owner of this structural feature does not exist in the target model, the current LostStructuralFeature is ignored;
3) for each LostModelElement successfully re-applied in step 1, Mt2Ms restores all the values of attributes and references that do not have a default value in the original source model Ms.

## III. IMPLEMENTING DULLY

We developed the current version of DULLY in the context of the ATLAS Model Management Architecture (AMMA) [4]. More specifically, our tool is available as a plugin of the Eclipse platform that extends the ATLAS

---

7The home page of the DULLY project is http://www.di.univaq.it/dually while the source code can be found in http://sourceforge.net/projects/dually, released under the GNU General Public License (GPL).
Model Weaver (AMW) [13]. Eclipse\(^8\) is an open source development platform comprised of extensible frameworks and tools for building, deploying and managing software across the lifecycle. The Eclipse open source community has over 60 open source projects. One of these projects is Eclipse Modeling Framework (EMF)\(^9\) that is a modeling framework and code generation facility for building tools and other applications based on a structured data model. AMMA is built on top of Eclipse and then models and meta-models are integrated into the same platform with several modeling technologies, such as Ecore (which is the meta model of the EMF framework for describing models and runtime support for the models) and UML2 (which is is an EMF-based implementation of the UML 2.x OMG meta-model for the Eclipse platform). We selected AMMA since this model management architecture is extensible and supports the concepts of weaving model and meta-model independence. A high-level overview of the technologies we used is represented in Figure 9. Weaving models are particular kinds of models that may be used to define semantic links among meta-models or UML profiles. Both meta-models and models (also weaving models via AMW’s specific editor) are expressed via XMI, while the transformation engine is based on ATL transformations [28]. Implementation details will be described in the following sections.

A. Technologies overview

AMMA is a model management platform designed and developed by the ATLAS Team at INRIA Institute. Its core elements are:

- Atlas Transformation Language (ATL) [28] which is a QVT (Query/View/Transformation)-like\(^{10}\) model transformation language, with its own abstract syntax and environment (i.e., an execution virtual machine and IDE). It allows us to transform a source model into a target model. ATL is a declarative and imperative hybrid language. This is one of the key features of ATL: it takes advantage of the simplicity of declarative constructs, while complex tasks (e.g., manipulating UML profiles) can be performed in an imperative fashion. The transformation is itself a model conforming to a specific meta-model. This permits the creation of higher order transformations, i.e., transformations that produce ATL transformations. This feature allows DUALLY to automatically generate ATL transformations as needed.

- Atlas Model Weaver (AMW) [13] is the platform that manages weaving models. It allows the definition of correspondences among models (or meta-models) and to establish semantic links among model elements. The links are saved in a weaving model, which conforms to an extensible weaving meta-model. The weaving meta-model input format is KM3 (Kernel Meta-Meta Model) [27], a language that provides a textual concrete syntax for the coding of meta-models in a Java fashion. The weaving and woven models are defined by the XMI (XML Meta Data Interchange) specification, the OMG format for model and meta-models interchange. Besides AMW is featured with a set of extension points that enable us to add specific semantics to the weaving mechanisms.

---

\(^8\)Eclipse project Web site: [www.eclipse.org](http://www.eclipse.org).


• AM3 (ATLAS MegaModel Management) is a global resource manager that operates in a model-engineering environment. This component provides support for modeling in the large i.e., dealing with models, meta-models, tools, services and their relations as a whole, while ignoring internal details. Within one platform (local or global), a megamodel [5] records all available resources and acts as an MDE repository. From a practical point of view AM3 manages megamodel elements (for example ATL transformations, tools, UML models, meta-models) and provides user interfaces to manipulate them.

• Other peripheral tools are grouped into the ATP (ATLAS Technical Projectors) component. These tools perform model transformation tasks. A subset of ATP tools consists of injectors and extractors to/from other technical spaces (e.g., UML, ATL, XML, SQL etc...) with little or no loss in information.

All AMMA projects are built on top of EMF. AMMA is a modeling framework that provides a MOF-like core meta-model (i.e., Ecore) to define both models and meta-models, tools for importing models and generating code, runtime model support (i.e., reflection, notification and dynamic definition), persistence layer (XML/XMI resource implementations), validation of models and UI-independent viewing and editing support.

B. Realizing the DULLY framework

DULLY is engineered as an extension of AMW. This extension consists of (i) a customized editor for the management of weaving models, (ii) a weaving meta-model that defines the types of link that the user can establish between meta-model elements, (iii) a set of higher order transformations to automatically generate ATL transformations at the model level.

Figure 9 points out how we use the technologies mentioned above. More precisely, both transformations between models and higher order transformations are expressed through ATL. According to the definition of their extension
point, weaving meta-models are defined in the KM3 language. The meta-models (or profiles) are expressed in XMI, demanding their import and export to the underlying Eclipse platform. The meta-model level of Figure 9 shows the intended passageway to integration through the $A_0$ meta-model.

The weaving model contains the links between elements of $MM_x$ meta-model/profile and elements of the $A_0$ profile. It holds the logic that will guide the automatic generation of the ATL transformations. Each weaving link is used by the higher order transformations (called HOTs in the context of AMMA) to generate either rules or bindings of the ATL transformation. The types of link are specified in the weaving meta-model.

As previously stated, DUALLY operates both on ADL meta-models and UML profiles. The meta-modeler uses the Ecore formalism to define ADL meta-models. These meta-models and their models can be expressed using either the tree-like editor or the graphical editor of EMF. Moreover, many tools exist that import/export Ecore models into/from the Eclipse platform. On the other hand UML profiles are defined using UML2, the implementation of the UML meta-model for the Eclipse platform\textsuperscript{11}. The main advantages that DUALLY gains from using UML2 are: compliance with OMG standards (specifically UML 2.0 and MDA) and interoperability with other UML2-based tools. This allows the user to graphically design UML profiles with any UML2-based tool and directly import them into DUALLY. The same mechanisms guide the import/export of UML2 models. By this means our technology achieves independence from tools used for modeling SAs.

As already mentioned in Section II, there are two main users of DUALLY: software architects and meta-modelers. A typical software architect usage scenario is:

1) modeling the software architecture;
2) applying the DUALLY model transformations to the model in order to obtain the software architecture in the target architectural language;
3) working on the software architecture in the target architectural language.

On the other hand, for each meta-model to weave $MM_x$ (for example MM1 or MM2 in Figure 9), a meta-modeler usage scenario of DUALLY is:

1) creating (or importing) the meta-model/profile $MM_x$ into Eclipse;
2) graphically developing the weaving model between $MM_x$ and $A_0$ through the DUALLY weaving models editor (represented in Figure 10);
3) applying the DUALLY higher order transformations to the weaving model in order to automatically generate transformations at the model level.

In this section we focus and provide details on the meta-modeler’s scenario while the software architect scenario will be explained thoroughly in the case-study section. The first activity of the meta-modeler’s usage scenario is based on the import/export mechanisms of the EMF framework; in the following subsections we explain how our technology supports the remaining activities. Section III-B1 presents how DUALLY extends the graphical editor of AMW (mainly by adding the buttons to launch the HOTs and to overcome some presentation issues) and

\textsuperscript{11}UML2 project Web site: http://www.eclipse.org/uml2/.
Section III-B2 describes DUALLY’s weaving meta-model. Section III-B3 explains the higher order transformations of DUALLY and Section III-B4 explains the generated first-order transformations.

1) DUALLY weaving models editor: the weaving models editor is the DUALLY’s graphical front-end. It manages the interaction between users and the model transformation engine. It is composed of three main panels: weaving panel, left woven meta-model panel and right woven meta-model panel.

In Figure 10 we illustrate how the weaving model and woven meta-models are represented within the DUALLY’s editor. The weaving panel (part B in Figure 10) is a tree-based editor to create a weaving model, i.e., the mappings between elements of two meta-models (left and right). It is built on the base weaving panel provided by AMW, taking advantage of the reflective EMF capabilities; this avoided us the effort of writing a specific editor for DUALLY and enabled us to rely on a tested editor also commonly used in research.

The elements of the editor reflect those of the weaving meta-model (later described in Section III-B2); each element is featured with its own icon and is created through a contextual menu. These choices speed up and simplify the development of weaving models: if the user clicks on a given element, the contextual menu shows only the child and sibling elements that may be added to such element, in respect to the rules imposed by the meta-model.

The basic toolbar of the weaving panel of AMW was extended with two buttons (see part E in Figure 10)

Fig. 10. Graphical interface of DUALLY
that allow the user to execute the HOT directly from the weaving panel; DUALLY automatically retrieves the information needed to launch the transformations.

The left and right woven meta-model panels (parts A and C in Figure 10) graphically represent left and right meta-models, respectively. These panels extend the reflective editor provided by EMF with the aim of representing the meta-models/profiles as hierarchical trees. The extension solves a problem presenting itself when dealing with hierarchical meta-models: let us suppose that the user is creating a weaving model between a UML profiles and a meta-model, namely leftProfile and rightMM; a correspondence between the Port element of leftProfile and the Interface element of rightMM must be created. At this point the user has to specify also the binding between structural features, say the “name” attributes at both sides. If DUALLY uses the standard EMF tree editor, the Port and Interface elements contain only their own features (leaving out the inherited ones, e.g., “name”), so the user has to navigate the tree searching the feature “name” in the parent elements of Port and Interface. This makes the creation of weaving links complex and error-prone. We extended the EMF tree editor so that each element displays all its structural features (inherited and local).

The decision to render the meta-models as trees was taken according to three main reasons: (i) the meta-models that formalize ADLs usually contain a large number of elements (e.g., UML profiles) and a graphical viewer could make DUALLY too complex to use, (ii) meta-models usually contain many hierarchical relations and the tree-view best shows this feature, (iii) to render models as trees is a common modeling practice, hence requiring less training from DUALLY users.

Woven meta-model panels have a toolbar (part D in Figure 10) that allows the user to render the meta-model (profile) as an Ecore diagram (UML profile diagram) taking advantage of EMF and UML2 Eclipse plugins.

As mentioned earlier, the meta-models to weave form an A0-centered star topology, therefore either left or right woven meta-model panel should contain A0.

Particular attention has been paid to the management of UML profiles. The procedure that DUALLY automatically applies while loading a UML profile is composed of two main steps:

1) transform the profile into an Ecore meta-model. In so doing, we avoid having to develop specific HOTs from Ecore meta-models to UML profiles and vice versa. More precisely, we develop a single type of HOT taking as input Ecore meta-models and the corresponding weaving model.
2) add the UML meta-model package to the profile. This step is unavoidable because the XMI file of a UML2 profile contains only the extensions defined by the profile. Conversely, a profile is by definition an extension of the UML meta-model, and so it must contain both its elements and the elements of the UML meta-model. DUALLY programatically adds the UML meta-model to the profile to fill this gap. We remark that the UML meta-model is not hard-coded in DUALLY, but rather it is dynamically retrieved from the UML2 Eclipse plugin. Therefore, DUALLY is auto-upgrading with respect to future UML versions.

In so doing, DUALLY becomes an integrated environment to map ADL constructs, to graphically develop the corresponding meta-models (profiles), and to perform model transformations of all possibly instantiable models. Furthermore, this environment is open to other modeling tools thanks to the importing mechanisms provided by
the EMF Eclipse platform.

2) **DUALLY weaving meta-model**: The AMW Eclipse plugin is based on a small weaving meta-model, defining the abstract notions of weaving links (what we earlier referred to as meta-model to meta-model links) between model elements.

This meta-model is abstract and is thought to be as generic as possible. However, it provides an extension mechanism to define domain-specific types of link. We developed **DUALLY**'s own weaving meta-model by extending the AMW basic one. Our extended meta-model adds ADL-specific constructs (e.g., specific links to map structural features) and is oriented to the automatic generation of ATL transformations (e.g., there is the distinction between source and target woven elements).

![DUALLY weaving meta-model](image)

**Fig. 11.** **DUALLY** weaving meta-model

Figure 11 illustrates the **DUALLY** weaving meta-model through its Ecore diagram (a graphical formalism to define Ecore-based models).

The **DUALLY** weaving meta-model elements are:

- **Dually** extends WModel of AMW and is the root element of each weaving model. It is composed of the references to **left** and **right** meta-models and a set of correspondences.
- **Correspondence**: represents a generic mapping between elements of the woven meta-models. It extends
the WLink element of AMW augmenting it with the condition property. Condition specifies the ATL guard of the matched rule to generate and is automatically injected into the generated ATL transformation. A Correspondence can be of three types: Left2RightCorrespondence, Right2LeftCorrespondence and EquivalenceCorrespondence. This is a peculiarity of DUALLY: it adds navigability semantics to weaving links. This gives us many advantages: (i) easier automatic generation of different ATL transformations from the same weaving model, (ii) creation of more complex and structured weaving models, (iii) strict relation between each weaving mapping and the corresponding generated ATL rule.

- **Left2RightCorrespondence**: this element represents a unidirectional correspondence, specifically a correspondence from an element of the left meta-model to one or many elements of the right meta-model. It contains a reference to one and only one element of the left meta-model and a reference to one or many elements of the right meta-model.

- **Right2LeftCorrespondence**: this is the opposite version of the previous element. So it contains a reference to one and only one element of the right meta-model and a reference to one or many elements of the left meta-model.

- **EquivalenceCorrespondence**: this element represents a correspondence with bidirectional navigability. The woven elements of this correspondence are exactly two (one from the left meta-model and one from the right meta-model), they are equivalent from the weaving point of view. This element contains a set of FeatureEquivalences that represents the bindings between the structural features (i.e., attributes and references) of the woven elements. It contains also a boolean attribute called MatchAllFeatures that will be explained in the next section.

- **WovenElement**: it extends AMW WLinkEnd and it is the abstract element that indicates the extremity of a correspondence. The feature variableName represents the name of the variable assigned to the element in the generated ATL transformation. Similarly to Correspondence there are three types of WovenElement: SourceElement, TargetElement, and EquivalenceElement.

- **SourceElement**: it is the source element of a directed correspondence. For example a Left2Right-Correspondence must contain a SourceElement belonging to the left meta-model.

- **TargetElement**: it is the target element of a directed correspondence. Similarly to EquivalenceCorrespondence it contains a set of FeatureEquivalences and the MatchAllFeatures attribute.

- **EquivalenceElement**: it represents an element of an EquivalenceCorrespondence.

- **FeatureEquivalence**: it extends AMW WLink and defines a mapping between two features.

- **Feature**: it extends AMW WLinkEnd and represents a structural feature of a woven element.

- **WovenElementLink**: it extends AMW WLink and specifies a correspondence between a WovenElement (or one of its features) and a feature of another woven element.

3) **DUALLY higher-order transformations**: DUALLY HOTs act as a bridge between the meta-modeling and modeling levels. Figure 12 illustrates the configuration of DUALLY while generating ATL transformations between MM1 and A0. The other symmetric transformations are generated with the opposite configuration.

The input of a HOT is composed of three models: (i) the weaving model, (ii) the left meta-model and (iii) the right meta-model (referring to Figure 12, WM1, MM1, and A0, respectively). The output is an ATL transformation
generated on the basis of the mappings defined into the weaving model (referring to Figure 12, Mm1_2_A0 and A0_2_Mm1); at a high level of abstraction, the current version of DUALLY translates each Correspondence into an ATL matched rule and each FeatureEquivalence into an ATL binding.

Furthermore, bindings can also be generated implicitly by the MatchAllFeatures attribute in a TargetElement. If this attribute evaluates to true the higher-order transformation calculates the union of the structural features of the woven elements, determines the subset of features with the same name and automatically generates the ATL bindings among the features of the subset. This avoids the manual definition of numerous weaving links when all the features of two elements must be bound.

If the user sets MatchAllFeatures to true and defines other feature equivalences, these will override the corresponding implicit ones, making the editor very flexible. For example, let us suppose the existence of elA and elB, in the left and right meta-models, respectively. elA and elB contain ten attributes with the same name. Figure 13 illustrates two versions of a weaving model that binds all the features of the elements. The weaving model (a) contains a feature equivalence for each feature to map and (b) implicitly binds such features. The weaving model (a) has complete control on the bindings, but the weaving model (b) is more readable and easy to design.

There are two HOTs in DUALLY: Left2Right and Right2Left. The former generates a transformation from models conforming to the left meta-model to models conforming to the right meta-model while the latter generates the inverse transformation.

4) DUALLY generated transformations: DUALLY automatically generates ATL transformations at the model level; we call them Basic Transformations (BT). For example these transformations are represented in Figure 9 by the arrow between M1 and M(A0).
The main difference between HOTs and BTs is that the former ones operate at the meta-model level while the latter ones operate at the model level. Therefore, a HOT is usually executed only once for each meta-model $MM_x$ during the generation phase; on the contrary BTs are executed on each model conforming to $MM_x$.

The HOTs dynamically verify if one of the input meta-models is a UML profile and reflects it to the generated BTs. In fact there are three types of generated transformations, depending on the type of meta-models they are generated from (Table I). The element $Ecore$ represents a MOF-compliant ADL meta-model, $UML$ profile represents
a UML profile, UML represents the UML meta-model and PRO represents the MOF definition of a UML profile. The last element is necessary because the ATL transformation must retrieve the definition of the stereotypes and tagged values that may have been set in the target model.

<table>
<thead>
<tr>
<th>Source meta-model</th>
<th>Target met.</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecore</td>
<td>Ecore</td>
<td>Ecore</td>
<td>Ecore</td>
</tr>
<tr>
<td>Ecore</td>
<td>UML profile</td>
<td>Ecore</td>
<td>PRO UML</td>
</tr>
<tr>
<td>UML profile</td>
<td>Ecore</td>
<td>Ecore</td>
<td>UML</td>
</tr>
</tbody>
</table>

**TABLE I**

**INPUT/OUTPUT OF THE GENERATED TRANSFORMATIONS**

We faced other technical issues while generating transformations that handle the UML2 Eclipse meta-model. This is caused by the specification of UML models provided by the Eclipse UML2 platform. UML2 is a MOF-compliant model, but the profile (stereotype) applications are only annotated to the elements. This means that the applied stereotypes of an element are not directly visible, but must be accessed through UML2 specific method calls.

The source element of a rule is matched against the meta-class of a stereotyped element, while the condition of the rule verifies if a specific stereotype is applied. The same mechanism is applied to the target element of the rule. Similarly an ATL rule cannot bind the tagged values of a stereotype, but must get/set their values through UML2 specific method calls.

**DUALLY** includes a sub-system that manages the name of variables in the BTs. It prevents the creation of transformations with conflicting variable names; if the user does not specify the attribute `variableName` for a woven element, **DUALLY** also generates a unique identifier for such an element.

**IV. IECS CASE STUDY**

In this section we show the application of **DUALLY** to a real, industrial, case study. The objective of the case study is to show:

- the **DUALLY**-zation mechanisms;
- how the extension mechanisms work in practice (actually, a behavioral extension to $A_0$ has been already presented in Section II-A1, while this section will show its use);
- the management of the lost-in-translation problem (i.e., how to avoid information loss during round-trip);
- the usefulness of **DUALLY**.

The software architecture of our case study comes from a project developed within Selex Communications, a company mainly operating in the naval communication domain. The final objective of this collaboration was to model check the software architecture of this system (called IECS - Integrated Environment for Communication

---

on Ship) and for this purpose, in [8] we used the CHARMY [39] modeling and analysis framework. The IECS architecture was already modeled in Selex Communications in terms of UML Component and State diagrams; however, in order to use CHARMY we were obliged to remodel it in terms of a CHARMY specification (since CHARMY is unable to import UML specifications).

Fig. 14. **DUALLY** Superstructure

In the same line of action, this section shows how **DUALLY** can be used to automatically transform a UML-based specification of an SA into an ADL suitable for model checking. More precisely, we model the IECS software architecture in a UML2.0 profile [25] and we make use of Darwin/FSP [33], [34] and its supporting tools for model checking purposes. We also describe how changes in the Darwin/FSP generated model can be propagated back to the (original) UML model. This round-trip journey is also used in order to show in practice the management of the lost-in-translation problem.

Figure 14 depicts the modeling technologies we used to develop the case study.

The whole process of the case study can be divided into two main phases, operating at the meta-model and model level, respectively:

1) the MDE expert, exploiting the software architects’ know-how for conceptually defining the semantic links, develops the weaving models (points 1 and 2 in Figure 14) and **DUALLY** automatically produces the following ATL transformations: \textit{UMLCC2A0} (point 3 in Figure 14) that produces models conforming to the \textit{A0} meta-model from UML models, and \textit{A02Darwin} (point 4 in Figure 14) that takes as input the \textit{A0} model (generated by \textit{UMLCC2A0}) and produces the corresponding Darwin/FSP specification;

2) software architects execute the generated ATL transformations obtaining Darwin/FSP specifications (7) from UML models (5) and vice versa. \textit{A0} models (6) are the means by which the two architectural notations interoperate.

The next sections present the details of the case study.
A. The UMLCC profile and Darwin/FSP meta-model

UMLCC is a UML profile for component-based architectures containing mechanisms to specify systems via components, connectors and their behavior; we designed this UML profile having in mind the guidelines provided in [25] and extending them in order to describe also connector roles, behavior and hierarchically-structured systems. The UMLCC profile is presented in Figure 15.

Fig. 15. UMLCC profile

According to the UMLCC profile, a software architecture is described by a component diagram containing the main components, connectors and their configuration; an optional auxiliary component diagram specifies the type system of the architecture being modelled. A behavioral description of each component and connector can be also defined.

Run-time architectural components are expressed using stereotyped UML Components called CcComponent Instances. We decided to extend UML Components so that ports can be associated to them and, since in the UML 2 meta-model Component is a subtype of StructuredClassifier, the internal structure of each component through its inner components, connector and attachments can be defined. Moreover, using UML Components allows designers to associate behavioral descriptions to each architectural component; behavior is expressed in terms of state machines (CcBehavior). Designers may specify different behavioral descriptions e.g., a use case to model the usage of a component, a set of state-machines to describe the internal policies or a scenario to specify the interaction with other components of the system. CcComponentInstances interact
with their external environment through a set of **CcPorts**; they extend the UML port meta-class and represent interaction points between a component and its external environment. **CcPorts** may act as endpoints of both **CcAttachments** and **CcDelegations**. **CcConnectorInstance** represents an architectural connector, an entity providing communication and coordination facilities. The structure of **CcConnectorInstance** is very similar to the one of **CcComponentInstance**, the only difference is that it interacts with external entities via **CcRoles** instead of **CcPorts**. Components and connectors communicate through stereotyped UML Dependencies called **CcAttachments**, which represent communication channels (e.g., message exchanges, service calls). A **CcAttachment** is a directed relationship, i.e., it always has a one-way direction; bidirectional channels can be expressed by a pair of **CcAttachments** with the same name and defined between the same ports. In the case of hierarchically-structured components, **CcDelegations** are used to link ports of the external component to the corresponding ports of inner components. The direction of the **CcDelegation** indicates whether it is the outer component that forwards messages to its internal elements or the contrary. The same mechanism is valid when connecting hierarchically-structured connectors through **CcRoles**. A set of OCL constraints have been defined to assure that **CcPorts** delegate only to **CcPorts** and that **CcRoles** delegate only to **CcRoles**. The **UMLCC** profile provides also a mechanism to specify the type system of a software architecture, so types of components and connectors can be expressed via **CcComponentTypes** and **CcConnectorTypes**, respectively; the various relationships between architectural types are captured by a different, re-usable component diagram. Figure 19 shows a high-level IECS UML model conforming to the **UMLCC** profile. It will be described in Section IV-D.

Darwin and FSP (Finite State Process notation) [29] are two strictly related efforts aimed at fully describing the software architecture from a structural and behavioral viewpoint. Darwin is a structural ADL capable of describing a system in terms of components, interfaces and interconnections among components.

The FSP specification needs to be attached to the Darwin specification. This means that in turn, every component will need its FSP process counterpart. FSP is a language that enables modeling behavioral aspects of a software system in terms of concurrent processes. Each automaton produced via an FSP is an LTS (Labeled Transition System) and can be analyzed via LTSA (LTS Analyzer)\(^\text{13}\).

Two fundamental elements are present in an FSP specification: actions (which make up processes) and states. Actions are either input actions that come from the process’s environment, or output actions that originate from the process. Processes can be described recursively. By convention, action names (in the alphabet of a process) are lower-case, and process names are upper-case. The Darwin/FSP meta-model is presented in Figure 16.

We constructed the joint Darwin/FSP meta-model directly encoding the two official BNFs\(^\text{14}\). In Figure 16 the joint meta-model is realized through the linking point being the association of **ComponentSpecification** of the Darwin part to **ProcessSpecification** of the FSP part. This relation ensures that each component has

---

\(^{13}\)LTSA tool: [http://www.doc.ic.ac.uk/ltsa/](http://www.doc.ic.ac.uk/ltsa/).

its behavioral specification. This relation is therefore the sole relation bringing together the two meta-models. It should be noted however that constraints defined in the comprehensive meta-model ensure that FSPs associated to components have the same alphabet of the component Portals [35].

Each model conforming to this meta-model is composed of two main parts: a DarwinSpecification and its relative FspSpecification. The Darwin part is made up of ComponentInstances, Portals and the relative Bindings; in turn, Portals refer to InterfaceDeclarations in order to specify the interfaces they expose and their polarity; a Darwin specification may also contain ConstantDeclarations which are valid throughout the entire specification, as well as AssertDeclarations which are used to maintain key properties
on the specification itself. This Darwin part must then be matched with the relative behavior specified through *ProcessDeclarations*, which are matched with the components they refer to. Each *ProcessDeclaration* is made up of *PrefixableElements* which can in turn be: (i) simple *Actions*; (ii) *GuardedActions* which carry an activation guard; (iii) *Choices* which carry a choice statement and (iv) *ProcessCalls* which embody a call to another *ProcessDeclaration*. FSP semantics also imposes the existence of a relabeling mechanism represented by the *Relabel* element, used only with relabeling operations (i.e., operations which deliberately change the label within the specification). Just like its Darwin counterpart, FSP provides the possibility of defining *Constants* to be valid throughout the specification. It must be noted that the presence of the three special *Processes*, namely STOP, ERROR, END, represent three special end-points, which must be present in the FSP specification by default.

All the elements captured in the meta-model have a counterpart in the textual notation. Once defined the transformations for importing and exporting Darwin/FSP textual notations a proper extension of *DUALLY* has been defined to take care of such features as shown in Section II. Since those transformations encode plain one-to-one mappings in the following we will not provide further details on the import/export mechanism.

**B. Weaving Models: UMLCC\_A\_0 and DarwinFSP\_A\_0**

In this section, by acting as the MDE expert, we show how weaving models can be created by any MDE expert interested in *DUALLY*zing a new notation.

The weaving model is the means by which the meta-modeling expert establishes semantic bindings between two notations. Therefore, in our case study we developed two weaving models:

1) *UMLCC\_A\_0* contains the bindings between the *UMLCC* profile and the *A\_0* meta-model;
2) *DarwinFSP\_A\_0* specifies the bindings between the Darwin/FSP and the *A\_0* meta-models.

The weaving models presented in this section contain only the minimum number of correspondences to generate the needed model transformations, leaving the models as readable and understandable as possible.

Figure 17 represents a simplified version of the *UMLCC\_A\_0* weaving model. (The full version is implemented in *DUALLY*).

The relevant correspondences of *UMLCC\_A\_0* are:

- A *CcComponentInstance* is translated into an *SAcomponent* and vice versa.
- Each *CcConnectorInstance* corresponds to an *SAconnector* and vice versa.
- *CcPort* and *CcRole* correspond to *A\_0* *SAinterface*. The direction of the *SAinterface* is derived from the direction of the *CcAttachment* connected to the source element, i.e., if only incoming (outgoing) attachments are present then the corresponding *SAinterface* is *Input* (*Output*) and if both incoming and outgoing attachments are present, it has an *InputOutput* direction. From the other side, if the *A\_0* *SAinterface* is owned by an *SAcomponent* it is translated into a *CcPort*, otherwise it is mapped to a *CcRole*. 

January 20, 2009 DRAFT
Each CcAttachment corresponds to an \( A_0 \) S Achannel and vice versa. If the CcAttachment is part of a bidirectional link (i.e., it is coupled with another CcAttachment with the same name and between the same ports) a single bidirectional \( A_0 \) S Achannel is generated. Conversely, from a bidirectional \( A_0 \) S Achannel two CcAttachments are generated. The transformations generated from this weaving model have been manually refined in order to handle this issue.

CcComponentType and CcConnectorType correspond to \( A_0 \) S AstructuredType. In the other direction, if the S AstructuredType is instantiated by S Acomponents then it is mapped into a CcComponentType, otherwise it is mapped into a CcConnectorType. If it is instantiated by both kind of elements it is not considered.

A CcDelegation corresponds to an \( A_0 \) S Abinding and vice versa.

A CcBehavior is mapped into the StateDiagram element of the StateDiagramMM extension of \( A_0 \) and vice versa (we remind the reader that state diagram elements are part of the \( A_0 \) extension illustrated in Section II-A).

A generic UML State contained into a StateMachine stereotyped with CcBehavior corresponds to a StateDiagram State and vice versa.

A UML Transition contained into a StateMachine stereotyped with CcBehavior is mapped into a StateDiagram Transition and the corresponding guard is copied in the condition field;

Each UML initial PseudoState contained into a
texttttextitCcBehavior StateMachine corresponds to a StateDiagram InitialState and vice versa;

- A UML FinalState contained into a StateMachine stereotyped with CcBehavior corresponds to a StateDiagram FinalState and vice versa.

UMLCC_A0 contains also other minor weaving correspondences that we did not include in Figure 17 in order to keep it as readable as possible, they are:

- **UMLModel_SA**: a UML Model is translated into a SoftwareArchitecture containing all its element. It includes also a predefined TypesSpecification element containing all SAtypes within the A0 model. In the other direction, a SoftwareArchitecture element is mapped into a UML Model if it is not contained into an A0 Component or an SAstructuredType, otherwise it is ignored.

- Each UML Property corresponds to an A0 Property and vice versa.

- A UML Package corresponds to an A0 Group. The inverse correspondence is not always feasible since a Group in A0 does not refer to its members with a Composition relationship as UML Packages do. Therefore, we decided to do not map A0 Groups back to UML Packages.

- Each non-stereotyped UML Dependency corresponds to an A0 SArelationship and vice versa (in case the SArelationship is not also an SAbinding or an S Achannel);

- UmlCC_A0 contains other minor correspondences that are not presented in this paper because they are used only for technical purposes, like setting the parents of model elements, bindings between simple attributes like name, labels and so on.

Figure 18 represents a simplified version of the DarwinFSP_A0 weaving model.

![DarwinFSP_A0 weaving model](image_url)
The semantics emerging from this weaving model are:

- Each Darwin/FSP ComponentInstance corresponds to an SAcomponent of $A_0$ and vice versa.
- A Darwin/FSP Portal corresponds to an $A_0$ SAinterface and vice versa. The kind of SAinterface is set according to the type of the Darwin/FSP Portal, i.e., if the portal is either “provide” or “export” then the SAinterface direction is “Output”, if the portal is either “require” or “export” then the SAinterface direction is “Input” and if the portal has no type then also the SAinterface has no direction.
- Each Darwin/FSP Binding is mapped to SAchannel and vice versa. This correspondence contains a condition that evaluates to true if the endpoints of the Binding are not owned by hierarchically nested components.
- Each Darwin/FSP Binding is also mapped to SAbinding and vice versa. In this case a condition checks that the Binding is between hierarchically structured components.
- Each Darwin/FSP InterfaceDeclaration corresponds to an SATYPE. The other direction of this correspondence contains a condition that evaluates to true only if the SATYPE is instantiated only by SAinterfaces.
- Each Darwin/FSP ComponentDeclaration corresponds to an SAstructuredtype. The other direction of this correspondence contains a condition that checks if the SATYPE is instantiated only by SAcomponents.
- Each Darwin/FSP ProcessDeclaration is mapped to a StateDiagram and vice versa.
- ActionPrefix and Choice are both mapped to an $A_0$ State and vice versa. These correspondences are particular since the corresponding rules in the generated transformations act as stubs in which we manually implemented the logic to correctly create FSP processes from state-machines and vice versa.
- ERROR, END and STOP can be considered as special kinds of sink states, so we mapped them to $A_0$ FinalState. In the other direction, we made the assumption that an $A_0$ FinalState always corresponds to a Darwin STOP.
- Darwin/FSP Action is mapped to Transition and vice versa.
- This is a special case of the previous correspondence, GuardedAction corresponds to $A_0$’s Transition and vice versa. A featureEquivalence exists between the “guard” and “condition” attributes of Transition and GuardedAction, respectively.

Minor auxiliary correspondences and bindings have been considered in DarwinFSP-$A_0$, but we do not discuss them in this paper for the sake of readability. Examples of such correspondences are those creating the DarwinSpecification and FSPSpecification default elements in a Darwin/FSP model or those setting obvious attributes like “name” or “owner”.

1) Properties of the defined weaving models: referring to the discussion on the correctness of transformations in Section II-B1, we can observe that if a bidirectional transformation $t$ binds two elements $a$ (in one meta-model) and $b$ (in the target meta-model) and no other transformation exists neither for $a$ nor for $b$, then we do not have correctness problems. Problems can arise when this condition is falsified.

Analyzing the defined weaving model in Figure 17 we can see that only SAinterface and SAstructuredType falsify the condition previously defined. In fact, two transformations exist from SAinterface going into CcPort
and CcRole, respectively. Actually, as explained before this is not a real problem since the transformations are defined tacking into account other conditions that disambiguate the non-determinism. In fact, SAInterface will be translated in a CcPort if and only if it is owned by an SAcomponent, otherwise it will be mapped into a CcRole.

A similar discussion can be made for SAstructuredType that will be mapped into a CcComponentType only if it is instantiated by SAcomponents, otherwise it is mapped into a CcConnectorType.

Focusing on Figure 18, the elements to be analyzed are Binding, State, FinalState, and Transition. Binding is easily disambiguated since it is mapped into an SAbinding if the Binding is defined between hierarchically structured components and into an SAchannel otherwise. State is a bit different since, in order to correctly define an FSP specification, a state must be translated in both ActionPrefix and Choice, as defined before when presenting the weaving models. For FinalState we resolved the ambiguity by assuming that a FinalState is always mapped to a STOP state. Finally, Transition is disambiguated since the bidirectional transformations are defined between Transition attributes and Action and GuardedAction of the other model.

C. Generation of ATL Transformations

The weaving models described above form the logic that generates ATL transformations. Higher-order transformations take as input meta-model specifications along with the related weaving model and return model-to-model transformations. While the ATL transformations generation phase can be the most crucial, our framework makes it totally transparent to the software architect that does not need any knowledge about model transformations.

Our case study contains two generated ATL transformations:

- **UMLCC2A0**: it is generated from the UMLCC_A0 weaving model and returns A0 specifications from UML models profiled with the UMLCC profile.
- **A02DarwinFSP**: it is obtained from the DarwinFSP_A0 weaving model and returns Darwin/FSP models from A0 models;

The semantics of each transformation reflects the correspondences contained into the weaving model it is generated from.

The following text shows an excerpt of the **UMLCC2A0** ATL code also instructed by DUALLY to produce a Lost-in-translation model. The actual application of the mechanism to handle the lost-in-translation problem will be shown in Section IV-D.

```plaintext
module UmlCC2A0;
create OUT : A0, LOSTMODEL : LOST from IN : UML2;
helper def : matchedModelElements : Set(OclAny) = Set{};
helper def : matchedStructuralFeatures : Set(OclAny) = Set{};
... 
```
rule CcComponent2SAcomponent {
from
    s : UML2!"uml::Component" (s.isStereotypeApplied('CcComponentInstance'))
to
    t : A0!"A0::core::SAComponent" {

        ... } do {

            thisModule.matchedElements <- thisModule.matchedElements.including(s);
            ... }
        }
}

endpoint rule finish() {

to
    t : LOST!"Lost::Lost"(name <- 'IECS-MS__UmlCC2A0',
        model <- 'IECS-MS.uml',
        transformation <- 'UmlCC2A0.atl')

        ... } do {

            for (e in (UML2!EObject.allInstancesFrom('IN') - thisModule.matchedModelElements).asSequence()) {

                t.elements <- thisModule.createLostModelElement(e);
            }

        }
}

rule createLostModelElement(element : UML2!EObject) {

to
    t : LOST!LostModelElement {
        name <- if(element.eClass().getEStructuralFeature('name').oclIsUndefined())
            then OclUndefined else element.name endif,
        modelElementRef <- ref,
        ref : LOST!Ref (ref <- thisModule.getRef(element))
    } do {

        t;
    }
}

helper def: getRef(element : UML2!EObject) : String =
    element.__xmiID__;
A description of the UMLCC2A0 is provided below. Its header (line 2) specifies that this transformation takes as input a UML2 model and produces an A0 model (OUT) and a Lost-in-translation model (LOSTMODEL). The helpers in lines 4 and 5 serve to keep track of the source model entities matched during the execution of the transformation. This information is required as part of the lost-in-translation mechanism. CcComponent2SAcomponent (lines 9-23) is a standard matched rule that transforms a UML CcComponentInstance into an SAcomponent. Notice that, conforming to the UML2 Eclipse specification, the source element is a standard UML Component and only later the CcComponentInstance stereotype application is checked through the call of isStereotypeApplied, a helper generated by DULLY in this transformation. Moreover, in the “do” section of the rule, a reference to the matched source element is stored in the matchedModelElements set. Such set is used in the finish endpoint rule (lines 27-42). It generates a Lost entity (lines 29-33) setting (i) its “name” attribute, (ii) the “model” attribute to the name of the source model and (iii) the “transformation” attribute to the name of the transformation being executed. The list of all the elements that have not been matched during the execution of the transformation is also built; this is done in lines 36-37 by computing the set difference between the set of all model elements of the source model and the matchedModelElements set. For each element of this list a LostModelElement is created by calling the createLostModelElement rule (lines 44-57); it creates a LostModelElement and the corresponding Ref entity is generated. The usefulness of the Ref meta-class in the Lost meta-model is clear here: supporting different identification mechanisms consists in re-implementing just the getRef helper (lines 59-60). The name attribute of LostModelElement is set according to the source model element. It is generated from (lines 47-48): if the source element has an attribute called “name” then its value is just copied otherwise it is not set.

Attributes, which are basic datatype values, and references to other elements of the model are managed in a similar way and for the sake of brevity we omit a further description of their management.

D. IECS Modeling in the UMLCC profile and Transformations to Darwin/FSP

The software architecture of our case study models a multi-tier environment capable of maintaining a fail-safe, client-server like communication within a safe and secure environment such as a military vessel: the Integrated Environment for Communication on Ship (IECS) [8]. The case study’s specification comes from a project developed within Selex Communications, a company mainly operating in the naval communication domain.

The purpose of the system is to fulfill the following main functionalities: i) provide voice, data and video communication modes; ii) prepare, elaborate, memorize, recover and distribute operative messages; iii) configure radio frequency, variable power control and modulate transmission and reception over radio channel; iv) remote control and monitoring of the system for detection of equipment failures in the transmission/reception radio chain and for the management of system elements; v) data distribution service; vi) implement communication security techniques to the required level of evaluation and certification.
The IECS software architecture is composed of the Management System (IECS-MS), CTS, and EQUIPMENT components. In the following we focus on the IECS-MS, the most critical component since it coordinates different heterogeneous subsystems, both software and hardware. Indeed, it controls the IECS system providing both internal and external communications. Given its excessive size and complexity, we chose a portion of the whole system, referred to as IECS-MS from now on.

Figure 19 shows the overall structure of the IECS-MS architecture, as modeled by a software architect by using the UMLCC profile. This sub-architecture, involves several key operational consoles that manage the heterogeneous system equipment through Proxy computers. For this reason the high level design is based on a manager-agent architecture, which is summarized in Figure 19, where the Workstation component represents the management entity while the Proxy and the Communication Transfer System Manager (CTSM) components represent the interface to control the managed Equipment and the heterogenous system equipment including an ATM based communication transfer system (this part is considered external to the system and then is not modeled). The DB is maintained aligned by the Workstation and the Proxy components. The User component represents the operator interface to activate and deactivate services and to interact with the Equipment component (always by means of the Workstation that is the entry point of the system). The IECS-MS components’ behavior has been specified through state diagrams, whose complexity ranks from 61 to 4 states. It must be noted however that the complexity of the full state machine of the system reaches up to some 200 states, which clearly marks the complexity of the case-study itself. Figure 19 shows the state machines of the CTSM and DB components. They are the smallest state diagrams, smaller enough to show how DUALLY translates state diagrams into $A_0$ first and then in Darwin/FSP, avoiding to introduce unneeded complexity.

Once the model illustrated in Figure 19 has been created, we give it as input to UMLCC2$A_0$ (the transformation automatically generated by DUALLY). Figure 20 shows the model resulting from such execution.

Inevitably, some elements may go lost in translation if proper mechanisms to handle them are missing. These elements do not find a direct correspondence in $A_0$ and must be maintained using our mechanism for “handling” lost-in-translation situations. This mechanism stores unmatched elements in a model conforming the lost-in-translation meta-model in order to properly redeploy them in the proper diagram, when moving back to the originating technology.

Figure 21 shows the lost-in-translation model produced by the execution of UMLCC2$A_0$ on the previously described IECS model.

The generated lost-in-translation model conforms to the lost-in-translation meta-model in Figure 8. It is composed of two parts: (i) the LostModelElements that are lost, and (ii) the Refs that refer to $A_0$ model elements (to which each lost element refers to). The lost part contains both elements that are specific of the IECS model, such as Regions, and elements automatically added by MagicDraw, such as PackageImports, some ProfileApplications, and Comments. Focusing on IECS specific elements, UML Regions are lost since they cannot be mapped to any $A_0$ element. In particular, UML StateMachines can contain states and transitions only through Regions, while $A_0$ state diagrams, conforming to the $A_0$ extended meta-model in Figure 4, contain
Fig. 19. IECS-MS software architecture modeled using the UMLCC profile
The next step is the transformation of the current $A_0$ model of the IECS management system into a Darwin/FSP specification through the $A_02DarwinFSP$ ATL transformation. The result of this transformation is shown in Figure 22. We report only two state diagrams, i.e., the DB and the CTSM state diagrams.

Also in this case we have unmatched elements, such as the state names (since state names are implicit in the FSP notation) and the name of the $SAchannel$ (since it is not possible to associate names to Darwin bindings). The lost-in-translation mechanisms manage them in the same way as showed before, even though, for the sake of brevity, we do not show them here.

The next step is then to use the LTSA framework in order to make some analysis. We report here the result of the safety check we performed on the IECS-MS subsystem, after having deployed on our diagrams an exemplar seeded fault. LTSA found a deadlock on the system and provided also a trace able to guide us to the error. The error analysis we performed has shown us that the deadlock was caused by the $DB$ component, as it effectively was. In fact, as can be seen in Figure 22 several accesses to the database are not transactional, e.g., for $updateServiceInfo$ and for $readServiceInfo$ (the operations are performed through two transitions). Therefore, the $Workstation$ component and the $Proxy$ component concurrently access to the $DB$ and this causes the deadlock. The $User$ component is blocked.
since it interacts with the Workstation, the Equipment is blocked since it is waiting for requests or for parameters settings, and finally also the CTSM component is blocked since it waits for instructions from the Workstation or can raise alarms but also alarms cannot be managed by the blocked Workstation.

In order to fix the identified deadlock, the DB component has been modified, as reported in Figure 23, in order to make the access to the DB transactional (request and answer are represented with only one message); DUALLY has to propagate back the changes.

The final updated UMLCC model is automatically obtained by executing the “backward” model transformations (see Figure 14) also instructed to reapply the model elements stored in the lost-in-translation models. More specifically, DarwinFSP2A0 is executed to produce the A0 model and then the A02UMLCC transformation is executed to produce the final and updated UMLCC model. These two transformations automatically recover the elements lost and stored by the A02DarwinFSP and UMLCC2A0, respectively. Figure 24 shows the automatically obtained DB state machine once re-imported into MagicDraw. It becomes evident that all the entities created in the form of lost-in-translation content, are again added to the model which is therefore fully consistent with the one it derives from. We have shown how, after a full round-trip journey, all the data in the original model is still present in the final model and also consistent with all the modifications made on either side of the journey itself.

It is imperative to understand that the round-trip journey iterates itself every time a different ADL is called in. In so doing, we may state that DUALLY effectively ensures both information consistency and no information loss.
Fig. 22. Generated IECS Darwin/FSP model

Fig. 23. Changes on the DB component

V. RELATED WORK

In the following section we delineate the ADLs as well as other technologies, frameworks and research that has similar intent to DUALLY, which we took as an inspiration point while working on DUALLY itself. Our main intent is that of showing a glimpse of existing technologies, their particular and personal domain as well as their own support for analysis and other design activities. From these considerations we derive the rationale behind
**DUALLY.**

This section is organized in two parts. In Section V-A we provide insight on a number of ADLs whose purpose we considered closer to that of DUALLY: the Acme initiative [20] from Carnegie Mellon; the xADL project [11], a very effective and extendable description framework and AADL, an industrially adopted and affirmed architectural description framework originally meant mainly as an avionics-focused DSL. In Section V-B we will mention other attempts as well as other closely related efforts at tackling the problems encountered and surpassed by DUALLY.

A. **Sightseeing related ADLs.**

The Acme initiative [20] is famed for being one of the very first technologies to tackle the problem of architectural data interchange. Acme and its supporting tool ACMEStudio were born as a simple, multi-style ADL framework also providing the possibility of using it as a common interchange platform for multiple ADLs. It eventually drifted away from this particular field now standing on its own as an architectural description and analysis framework.

Acme provided foundations and mechanisms to extend the framework. Tooling extension points come from allowing other tools to physically read and write Acme descriptions while semantic extension is enforced by allowing properties to carry ADL-specific data within the model. The provided ACMElib library can also be used to adapt ADL technologies to Acme and to allow their manipulation within ACMEludio. Some additional efforts brought Acme closer to UML by providing a ready-made profile for this task [22]. A number of reasons can be presented for Acme drifting somewhat away from its initial goal, the main of which is that its core technology does not provide direct support to integration with other architectural description frameworks. More specifically, Acme comes with libraries for parsing, unparsing and manipulating the representations that aim to help software architects to integrate new ADL descriptions. However the hard work in relating two different ADLs is not properly supported, as discussed below. Contrarily to DUALLY software architects are obliged in Acme to explicitly write end-to-end transformations among each pair of ADLs. As described in [19], in order to integrate two ADLs, e.g., Wright and Rapide, the Acme language needs to be augmented (through annotations) with specific information coming from Wright, and peculiar to it. Successively, the augmented model needs to be extended even further, with...
Rapide specific information (and therefore, yet again the kernel might itself need augmentation). As claimed in [19],
the hard work occurs in the middle step, i.e., when bridging the semantic gap between Wright and Rapide, and this
step is not properly supported. This brings no evident advantage in having an intermediate notation. It is also not
evident how the annotation mechanism of Acme will work when considering ADLs very different from Acme. Also
contrarily to DUALLY, Acme authors make clear in [19] that the true goal of Acme consists in simplifying the
migration of one technology to Acme (and once there, utilize the Acme analysis and modeling technology present)
by acknowledging the clear difficulty to relate two different ADLs utilizing the Acme approach. Furthermore, Acme
does not provide support for the lost in translation and with the used technology is very difficult to argue about the
correctness of the transformations.

Moreover, efforts invested in integrating technologies within Acme are not reusable outside of ACMEStudio [20].
Descriptions instantiated within Acme will still have problems in “exiting” the technology and will depend heavily
on its intrinsic format.

DUALLY keeps the idea of a base kernel of elements acting as doorway for information migration. By means
of fully automated transformations, DUALLY will allow for semantic information to be maintained and upheld. 
From an interchange specific perspective, DUALLY renders it simple to integrate a technology while keeping it
core-independent, thanks to the mechanisms of profiling and meta-modeling used upon integration. Summarizing
the DUALLY key differences are: (a) DUALLY’s A0 is frozen in its essential core and provides attaching
points within its meta-model so that (independent) extensions may be developed and included in the A0 system;
(b) DUALLYzation within our technology is an agile and architect-friendly modeling step, transformations are
automatically generated by our system once the process is complete, and all the modeled transformations and
the relative weavers are fully reusable in a model-driven fashion; (c) DUALLY’s key base concept is to allow
interchange: we want and stress other technologies to use our baseline as a bridge in order to reach other technologies,
we do not provide any analysis techniques and technologies and we will never stand as a reaching point; rather,
we provide solid technologies to allow migration from one ADL into another.

xADL [11] is an active research effort born and in progress at ISR - University of California, Irvine. Just like
DUALLY, the technology bases itself around a core of elements as a reference: xArch. xADL inherits from xArch
a number of like-to-have features such as direct run-time instantiation of SAs, model grouping, SA hierarchy and so
on. xADL as well as its core xArch, are based on XML and thus fully extendable [10]. The xADL project evolved
into a powerful technology backed by a solid workbench, ArchStudio 4, which maximizes the possibilities the
technology shows. Considerable efforts were poured in developing XTEAM: an MDE (Model Driven Engineering)
focused extension to the xADL framework. Such an attempt shows the importance to support and integrate MDA
(Model Driven Architecture)\textsuperscript{15} and MDE-focused industrial development processes. The possibilities and limitations
exposed by xADL on its own are very similar to those evidenced within Acme. Integration efforts have still to

\textsuperscript{15}MDA specification document: \url{http://www.omg.org/cgi-bin/doc?omg/03-06-01}.
physically augment the common XML architectural description provided within xArch: in this sense, a typical scenario would imply extending the core once for one technology and yet again for another one. The same core will become bigger with the first extension and yet even bigger because of the second one: this will still make the extensions “depending” upon one and the other.

Following these points, a natural comparison is possible between xADL and DUALLY: our approach is very close to the strong-points and novelties introduced by XTEAM as evidenced in [30], and it also provides full compatibility and support to the MDA process. While xADL may be used to define DSLs, i.e., it is a structured language engineered to be extended and “configured” so as to represent desired domain specific architectures, DUALLY may be seen as a multi-technology interchange language, i.e., a framework and an infrastructure to allow intercommunication and architectural data migration between ADLs themselves and/or UML profiles. DUALLY and xADL, however, both share a modular structure and hence neither can be seen as a single monolithic block within which to construct software architectures.

AADL [16] was born as an avionics focused DSL and later on moved to representing and supporting embedded real-time systems: its name and acronym were then re-factored to architecture and analysis description language. Standardized by the SAE (Society of Automotive Engineers) AADL now shows impressive usage in the field of both software and hardware specification and validation. AADL was designed as an extensible core language supporting modeling from multiple aspects and points of view. The validation technology it provides, addresses timing and performance properties of systems. Concurrency and interaction semantic specification is enforced. AADL was also one of the very first ADLs that introduced the concept of “feature” as a component’s way to interact with others at the implementation level. Moreover, the key concept of “aspect” of a system, during its development, is also available within the technology as exposed in [12]. The notation’s extension mechanisms include the definition of custom properties to specify additional ADL-specific analyses and/or generic information to be attached on the architectural design. Additional notation extension efforts are bringing AADL closer to UML friendly notations (via a profile) and are in their final steps [17]: the initiative has developed a UML profile (or rather an xUML profile, i.e., UML with formal action semantics embedded in it) to later on synchronize the two technologies. In addition, the main open source tool of the technology, OSATE tool, is very close to our view of an ideally extensible framework: it supports plug-ins, core-set extensions and might be made to support MDA specific technologies, by coordinating it with the Eclipse MDA initiative. Other 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. Our research effort took AADL as a key reference due to the exposed features and also because of its widespread adoption within dependability critical industrial development processes. Unfortunately, AADL does not provide automated support to its extensibility possibilities. Somewhat like xADL, AADL can be viewed as a modeling notation that can be complemented by description technologies tailored to specific goals of a particular modeling view and it can be considered a language supporting DSL generation. We decided to follow other recent trends and research directions in the ADL field confirming that interoperability
between these domain specific realities is still to be realized effectively. As a consequence, DUALLY’s goal is that of providing automated interoperability and synchronization mechanisms between technologies while also making no compromises concerning user-friendliness.

**B. Related Efforts.**

DUALLY’s basic architecture is centered around a core set of essential descriptive elements: \( A_0 \). This idea derives primarily from the requirement of providing a common minimal ground language to enable all architectural description frameworks to “communicate” between one another. This communication is a mid-way layer between every integrated technology. Previous technologies and some UML-based attempts at integration or architectural description, underlined the need for a basic set of meta-elements to be used as a reference set. A number of different yet close attempts both at the description and formalization of such a set may be found in [26] and [1].

In [26] we may find a perfect example of meta-information exchange between ADL formats via model transformation. This particular case shows a glimpse of the full potential that can be achieved within DUALLY, its transformations and our transformation engine. Considered technologies are Acme and META-H\(^{16}\), two of the previously mentioned mainstream modeling technologies. The paper shows in detail the actual implementation of the meta-interchange to take place from Acme to META-H, as it is carried out within the eclipse framework. The same principles are applied within DUALLY with the chief difference that the shown transformations will execute mainly on model instances whereas DUALLY is able to encompass a much wider scope. In fact, DUALLY’s interchange engine will make sure that semantic communication is on place on both modeling and meta-modeling level.

In [1] Smeda et al. bring around a relatively new concept in the field of architectural description: that of meta-architecture description language, i.e., Meta-ADL (MADL). The paper focuses on the specification of a meta-level to describe ADLs right from their core level, rather than architectures themselves. The paper shows the undoubted potential behind the specification of this meta-meta-model for architectural description languages, and provides a possible implementation of such a technology.

The approach however shows some serious shortcomings: the development of a totally stand-alone architectural “MOF” is both expensive in terms of effort and compatibility with UML. Utilizing such a technology in fact, would mean to sacrifice compatibility with UML right from its MOF. We assume this incompatibility as being unacceptable in the modern architectural description field we previously presented. [1] also provides a detailed assessment of architectural description issues that are yet to be solved (such as full standardization, architectural comparison, integration and interchange of formats, styles etc). While MADL effectively tackles these issues, some newly opened ones remain unhindered: industrial integration, technology roll-out, knowledge re-use and a number of other issues will remain untouched. DUALLY tries to work on the same scope as in [1] while also trying to “think industry” by considering the industrial prospects and problems on architectural description.

\(^{16}\)http://www.htc.honeywell.com/metah/prodinfo.html
VI. Evaluation and Considerations

In this section we provide an evaluation and some considerations about DUALLY. The discussion will be organized in two main arguments: (i) masking the complexity and (ii) integration with MDA technologies.

(i) Masking the complexity: the process suggested by DUALLY is organized in two different phases: typically the first phase (that ends with the transformations generation) is executed only once for each couple of meta-models/profiles and it is managed by a meta-modeler who retains a deep knowledge of the meta-models to weave; the second phase is executed every time a software architect needs to pass from a model source to the corresponding target model. DUALLY works at two abstraction levels, therefore masking to the software architects the model transformations technology. The role played by the software architect is in fact that of a final user that has only to model the system SA on an ADL and he automatically will have the system SA in each DUALLYzed ADL.

(ii) Integration with MDA technologies: since DUALLY is built around AMMA, which is an eclipse project, it could be easily integrated with MDA technologies already available within the eclipse community (e.g., Business Process Modeling Notation (BPMN), Eclipse Model-to-Model Transformation (M2M), Model To Text (M2T), Business Intelligence & Reporting Tools (Birt), Omondo, AndroMDA, etc.)\textsuperscript{17}.

VII. Conclusions and Future Work

In this paper we presented DUALLY, an automated framework that allows architectural languages and tools interoperability through model transformation technologies. DUALLY’s goal and rational is to solve the problem of having a proliferation of ADLs and UML notations for SAs not supported by effective interoperability. DUALLY brings together different architectural languages through a common semantic core, called $A_0$, that provides the infrastructure upon which to construct semantic relations among the different ADLs. DUALLY is implemented as an Eclipse plugin.

In order to fully automate DUALLY’s infrastructure a set of mechanisms for the (semi)automatic generation of the Weaving Model (WM) could be provided. This is possible thanks to the so called matching transformations \cite{15}. Such transformations select a set of elements from the input meta-models and produce links between them; these links are then captured within a weaving model. Matching transformations create these weaving models by executing matching heuristics guided by similarity principles. This value can be calculated by using, for example, the edit distance between the names of the elements of the input meta-models, equality parameters between attributes or classes or by using the structural relationships (e.g., containment) between model elements. An important aspect is that matching transformations may be applied in sequence, thus creating a chain of transformations that progressively forge the definitive weaving model.

Further effort could also be invested in considering DUALLY as base infrastructure upon which to construct new ADLs, which can be created by merging existing characteristics specific to different existing architectural languages.

\textsuperscript{17}BPMN (eclipse.org/stp/bpmn), M2M (eclipse.org/proposals/m2m), M2T (eclipse.org/modeling/m2t), Birt(eclipse.org/proposals/eclipse-birt), Omondo(eclipsedownload.com), AndroMDA(andromda.org).
The challenge is to automatically build new ADLs while keeping firmly the semantics and characteristics of the original architectural languages. It is important to note that in case of different needs, the software architect will be provided with facilities to cut off unwanted features while finely tuning his own instance of "third generation" ADL. This possibility alone, clearly presents itself as a key difference with respect to current extensible architectural languages. A first rough experiment in this direction is reported in [42]: weaving operators are used to extend profiles/meta-models in a conservative way: deletions of constructs are denied while allowing for their specialization or refinement exclusively. Two operators were devised:

(i) the *inherit operator* projects within a profile/meta-model, elements of other profiles/meta-models which are linked through an inherit stereotyped association. As a result, elements already existing within the profile/meta-model to be extended, are augmented with data coming from the elements to which they are related through inheritance.

(ii) the *integrate operator* extends profiles/meta-models with entire blocks of data coming from other existing architectural languages which provide constructs not present already.

In [42] we showed the extension of a first version of $A_0$, which was defined as a UML profile, with fault tolerance information with the aim of capturing in architectural components both normal and exceptional services. The exceptional part implements the responses of the component to exceptional situations, by means of exception-handling techniques.

Concluding, an ultimate future research direction is to reuse the DUALLY framework in contexts other than the architectural languages context. This implies the definition of a new $A_0$, customized for the new context, and thus reusing the DUALLYzation technology to bind the different approaches together as required by the new context itself.

ACKNOWLEDGMENT

The work is partially supported by ARTDECO (Adaptive infRasTructure for DECentralized Organizations), an Italian FIRB (Fondo per gli Investimenti della Ricerca di Base) 2005-2009 Project. The authors would like to thank Davide Di Ruscio and Nenad Medvidovic for their valuable support and comments.

REFERENCES


APPENDIX

List of Acronyms:

AMMA : ATLAS Model Management Architecture;

AM3 : ATLAS MegaModel Management;

AMW : ATLAS Model Weaver;

ATL : ATLAS Transformation Language;

ATLAS : Team at INRIA institute (Nantes, France);

ATP : ATLAS Technical Projects;

BNF : Backus-Naur Form;

BT : Basic Transformations;

CTS : Compatible Time Sharing;

DSL : Domain Specific Language;

DTD : Document Type Definition;
Ecore : Eclipse implementation of MOF compliant meta-meta-model;
EMF : Eclipse Modeling Framework;
FSP : Finite State Process;
HOT : Higher-Order Transformations;
IECS : Integrated Environment for Communication on Ship;
IECS-MS : Integrated Environment for Communication on Ship Manager-Agent;
JET : Java Emitter Templates: http://www.eclipse.org/modeling/m2t/?project=jet;
KM3 : Kernel Meta-Meta-Model;
LTS : Labelled Transition System;
MDA : Model Driven Architecture;
MDE : Model Driven Engineering;
MMx : generic meta-model;
Mx : generic model;
MOF : Meta Object Facility - OMG official Meta-Meta-Model;
OMG : Object Management Group;
UMF : Multi Functional Unit (the acronym comes from the Italian “Unitá Multi Funzionale”);
UML : Unified Modeling Language;
UML2 : implementation of the UML meta-model for the Eclipse platform;
UML 2.0 version 2.0 of UML;
WM : Weaving Model;
WMM : Weaving Meta-Model;
XMI : XML Meta Data Interchange;
XTEAM : an MDE focused extension to the xADL framework.