Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies

Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien Andrew Tamburri

Abstract—Many architectural languages have been proposed in the last 15 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 focusing 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 interoperability. Given a 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, make some verifications by using LTSA, and reflect changes required by the verifications back to the UML specification.

Index Terms—Software architectures, interoperability, domain-specific architectures, design tools and techniques, model transformations.

1 Introduction

Up to the present day, many languages for specifying and analyzing software architectures have been proposed (e.g., [31], [16], [34]) and historically classified into two generations [29]. The "first generation" of Architecture Description Languages (ADLs), going from 1990 to 2000, had the main purpose of designing an ideal ADL [17] whose chief aim was to enable support of components and connectors' specification and their overall interconnection [33], [17], as well as composition, abstraction, reusability, configuration, heterogeneity, and analysis [35]. 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 [7], [34], 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 [29], 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, hence 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., [30], [14]) 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 [8], [14], [31], it is impractical to have a "universal" notation. Furthermore, when architecting a software system, several significant decisions must be taken. Some of them cannot be taken at the beginning of the architecting phase and are

1. In an ongoing study that we are conducting, we counted more than 50 ADLs proposed both from academia and industry.

• *The authors are with the Dipartimento di Informatica, Università dell'Aquila, Via Vetoio, 67100 L'Aquila, Italy.*
E-mail: {ivano.malavolta, henry.muccini, patrizio.pelliccione, damien.tamburri}@di.univaq.it.

Manuscript received 31 Mar. 2008; revised 30 Mar. 2009; accepted 14 Apr. 2009; published online 6 Aug. 2009.
Recommended for acceptance by R. Taylor.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-03-0123.
Digital Object Identifier no. 10.1109/TSE.2009.51.

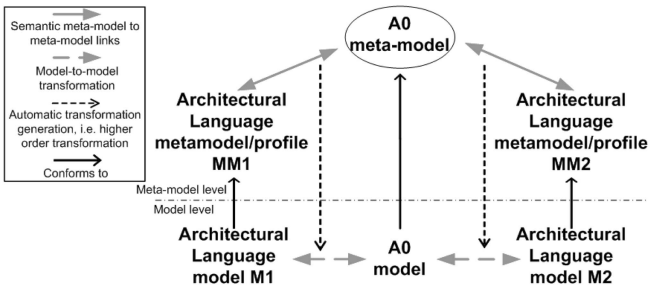

Fig. 1. **DUALLY** conceptual view.

made iteratively by analyzing and studying an architectural prototype [32] 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 nonfunctional aspect must be considered in a system. This implies that the choice of the best ADL (as required by the considered nonfunctional 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 5, 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 [16] is famed for being one of the very first technologies to tackle the problem of architectural data interchange. Acme drifted somewhat away from its initial goal for a number of reasons, mainly because Acme obliges us to explicitly write end-to-end transformations among each pair of ADLs; see Section 5 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 profile, needs to be model checked, a different architectural specification needs to be produced, e.g., in Darwin/FSP, since UML tools do not support model checking. As soon as the model-checking analysis in Darwin/FSP [26], [27] 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 Fig. 1,[2] **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 metamodel or a UML profile. Therefore, **DUALLY** works at two abstraction levels: metamodeling (upper part of Fig. 1) and modeling (lower part of Fig. 1). At the metamodeling level, model-driven engineers provide a specification of the architectural language in terms of its metamodel 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 can be noticed in Fig. 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 2.

DUALLY is implemented as an Eclipse plugin. MMi metamodels are expressed in Ecore. MMi profiles can be realized using any UML tool which exports in UML2,[3] the EMF-based implementation of the UML 2.0 metamodel for the Eclipse platform. A_0 is represented as a metamodel. Transformations among metamodels/profiles and model-to-model transformations are implemented in the context of the AMMA platform [3]. 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 **DUALLY**.

We argue that **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 metamodel in MOF), induced **DUALLY** to enable the transformation among formal ADLs and/or UML model-based notations.
- The evolved notions of software architecture (e.g., [29]) 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. **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 nonfunctional 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 [32]. **DUALLY** avoids, whether the architecture has been already modeled in an ADL, to redesign

2. This figure will be refined in the following portion of this paper.

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

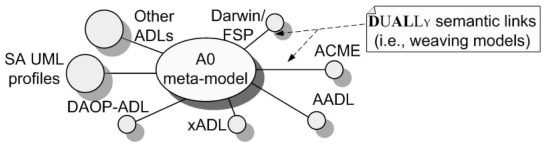

Fig. 2. **DUALLY** high-level conceptual view.

the software architecture in the different analysis-specific ADL.

Technological side

- **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 [6]. 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.
- **DUALLY** works at two abstraction levels, providing a clear separation between the role of model-driven experts and that of software architects. The model transformation engine is completely hidden to a software architect.
- Software architects can continue using familiar architectural notations and tools even reusing legacy architectural models. The **DUALLY**zation of legacy, textual specifications is also possible via a preliminary transformation step producing models from textual specifications.
- **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 2 presents **DUALLY** and its main principles. Section 3 presents the technology used to engineer **DUALLY** and discusses its implementation. Section 4 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 5 describes related work, while Section 6 evaluates **DUALLY** and provides consideration on its implementation and usage. Section 7 concludes the paper and envisions future research directions.

2 DUALLY: The Framework in Concepts

Fig. 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, Fig. 1 clearly represents a full instance of two branches from Fig. 2's more general view. **DUALLY** aims at supporting both languages and tools interoperability.

As far as *tool interoperability* is concerned, 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, it can be argued that all of the tools capable of importing/exporting 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 of importing/exporting 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 Java Emitter Templates (JET)[5] templates, Java classes, or ATL modules. To provide a solution for the cases above, **DUALLY**'s extension point provides the following facilities: 1) a custom **DUALLY** import/export menu entry and 2) an import (export) dialog wizard ending up with the automatic execution of the proper transformation.

In the following portion 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 Figs. 1 and 2: metamodels, UML profiles, semantic links, A_0, architectural models, and model transformations. The two main users intended for **DUALLY** are metamodeling 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 metamodels or profiles, by means of the process we call **DUALLY**zation. As clearly shown in Fig. 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 2.1, 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 2.2.

2.1 DUALLY Core Set: A_0

DUALLY inherits from both xArch [7] and Acme [16] the idea of identifying a core set of architectural concepts, hereafter called A_0 (see Fig. 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 Fig. 2), instead of creating a point-to-point relationship among all languages, a linear

4. XSL Transformations (XSLT) Version 2.0, W3C Recommendation: http://www.w3.org/TR/xslt20/.

5. JET home page: http://www.eclipse.org/modeling/m2t/?project=jet.

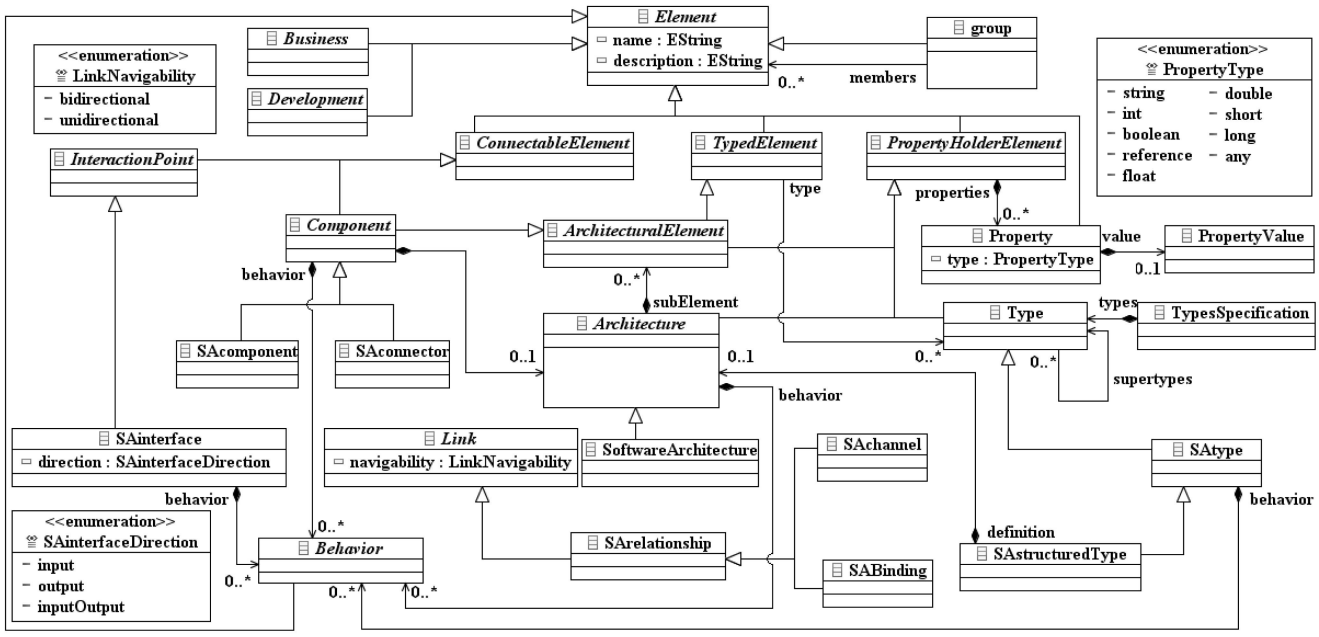

Fig. 3. A_0 metamodel.

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.

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, and Acme), relevant papers (e.g., [29], [10]), 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 of 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.

Here, in the following, we present in further detail the A_0 metamodel (shown in Fig. 3), by providing a description of its main metaclasses:

`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.

`SAcomponent`. A unit of computation with internal state and well-defined interface. A component can contain a subarchitecture 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 subarchitecture in the same way as `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, may contain a set of `Properties`, and 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 bidirectional communication and 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

intentionally do not define 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 metamodel. 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 and 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 subarchitecture, 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 SAstructuredType.

Behavior. Represents the behavior of a component, a connector, an interface, or a system architecture. It is an abstract metaclass that plays the role of a "stub" for possible extensions of the metamodel representing dynamic aspects of the system (e.g., finite-state processes, labeled transition systems, state diagram, and 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 metaclass is aggregated to Element, development details can be associated to every element of the A_0 model. Development is an abstract metaclass and its realization is left to future extensions of the A_0 metamodel.

Business. An abstract metaclass to be specialized via future extensions of A_0 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 A_0 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 A_0. 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 DUALLY 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 metaclass is aggregated to Element, properties can be associated to every element of the A_0 model.

PropertyType. An enumeration containing 1) common primitive datatypes, 2) a "reference" type that represents a reference to another element of the model, and 3) a generic type called "any" inspired by the OclAny type in the OCL language, which is considered the supertype of every PropertyType.[6]

PropertyValue. A metaclass 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 of a string that exposes models to ambiguity and expressivity problems. Moreover, thanks to the extensible nature of the A_0 metamodel, the PropertyValue metaclass 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 metamodel 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 metamodel, 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 metaclasses with very generic semantics. The rationale behind this choice is that of providing different granularity of "placeholders" to which metaclasses of future extensions can "hook-up" to.

Structural abstract metaclasses

Element. The root of the metamodel, i.e., every metaclass 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 metaclass 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.

6. Object Constraint Language (OCL) specification: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL.

ConnectableElement. Represents all of 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. 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 metamodel in order to ease the mapping of ADL-specific concepts to A_0 elements; the well-formedness of the A_0 metamodel 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 subarchitecture, 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 metamodel. Additional constraints can be defined straightforwardly exploiting the extensible structure of the metamodel. In the following, we describe the extensibility mechanisms of A_0.

2.1.1 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 metaclasses, it is always possible to extend A_0 by extending its root metaclass, i.e., Element (please refer to Fig. 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 DUALLY are:

1. 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;
2. 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 metaclass;
3. addition of new elements: for instance, a new diagram used for specifying the behavior or new elements in a diagram;
4. 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.

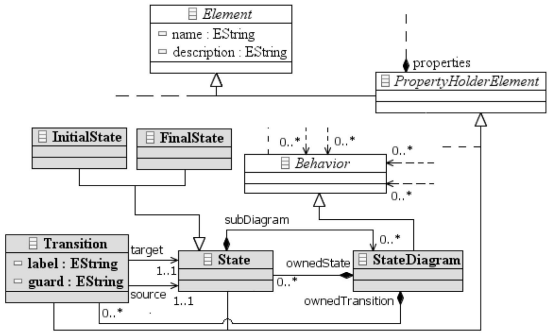

Fig. 4. Extension of A_0 with behavior.

These kinds of extensions are realized by means of the inheritance mechanism. Each element of the metamodel can be extended. For this reason, A_0 contains abstract metaclasses 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 metaclass. 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: 1) definition of StateDiagramMM, a basic state diagram metamodel and 2) relating elements of StateDiagramMM to abstract metaclasses of A_0. For the sake of simplicity, we defined a very simple and generic state diagram metamodel; its semantics is taken from [18]. The lower part of Fig. 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 substate diagram. Each transition may be labeled and a guard may check whether it can occur or not.

It should be noted that we defined a simple metamodel 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 metaclass with other behavioral notations like activity diagrams, sequence diagrams, Finite-State Process (FSP) specifications [24], etc.

The upper part of Fig. 4 shows how we related the StateDiagramMM metamodel to A_0 (note that, for presentation purposes, we only show parts of the A_0 metamodel that are relevant for the extension). As expected, the StateDiagram metaclass extends A_0's Behavior; thus, it has also a name and 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 metaclass so that software architects are allowed to associate properties to them. The

case study presented in Section 4 will make use of this extension of A_0.

In the next section, we will explain how the A_0 metamodel interacts within our technology to achieve **DUALLY**'s goal. The previously identified elements from Fig. 1 will be analyzed and their interaction points within **DUALLY** will be defined.

2.2 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 Fig. 1: Model transformations as well as metamodel to metamodel 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 metamodel to metamodel links that "physically" integrate the **DUALLY**zed technology, in the process of creating the lower level model-to-model transformations.

In the following, we describe metamodel to metamodel links and higher order transformations to create communication among models: It may be noted that only properly skilled metamodelers take part in the specification phase (only involving the software architects for semantically defining the links among the ADL to be **DUALLY**zed and A_0), while software architects work at the modeling level.

Metamodel to metamodel Links: Bringing metamodels together.

What we earlier called metamodel to metamodel links are the main branches that stem out of A_0 and they constitute a mechanism to provide references and semantic relations among elements at a metamodel level and A_0. 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 A_0 and the **DUALLY**zed 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 metamodel and can be defined either manually or by utilizing ad hoc scripting languages. **DUALLY** defines its own weaving metamodel, 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 transformation can be considered a model. Recent research efforts enhanced the concept of model transformation, promoting it to 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 **DUALLY**zation, model-level transformations are automatically obtained. These transformations can be executed directly. Uniting all of these transformations, a network is obtained; this will enable full, level-wide communication.

2.2.1 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 **DUALLY** contains mechanisms to check if a model conforms to its metamodel.

A significantly more complex notion is semantic correctness: Does the produced target model have the expected semantic properties? Each **DUALLY**zed 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 propagating 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) [19]. 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 [19], we report some definitions with the aim of defining 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* trans*: $\mathcal{M}_S \rightarrow \mathcal{M}_T$ is a mapping of elements of $\mathcal{M}_S$, which is the source metamodel, to elements of $\mathcal{M}_T$, which is the target metamodel.*

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

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

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

In order to deal with transformations that are neither total nor injective, [19] 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 *strip* that maps elements to their stripped-down relevant source and target models.

Once having defined the relevant source and target models, it is possible to define synchronization between two models. The following definition is taken from [19]:

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

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 *trans*, $trans^{-1}$, can be used to reobtain the source model starting from the target model, i.e., $trans^{-1}(T) = S$. Since, typically, 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 models: $trans^{inv}(T) = strip(S)$. Actually, round-trip engineering does not aim at recovering lost and unavailable source models, but 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* 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 Δ_T:$(\mathcal{M}_T \rightarrow \mathcal{M}_T)$ such that $\mathrm{trans}^{RT}(S, T, \Delta_T) = S'$, 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 $\mathrm{A} \rightarrow \mathrm{B} \rightarrow \mathrm{A}'$, 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, "loss in translation" is avoided.

2.3 Managing the Loss in Translation

Within the **DUALLY** framework, one of the most important properties to preserve is the synchronization between **DUALLY**zed 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 4, we present a **DUALLY**zation 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 [19], 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. Fig. 5 shows how the mechanism works.

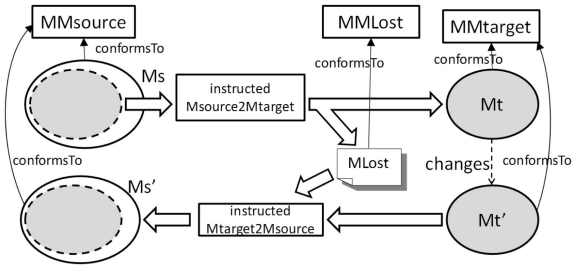

Fig. 5. Basic mechanism to synchronize **DUALLY**zed notations.

Let us suppose that a metamodeling expert **DUALLY**zed a generic source notation `MMsource` and that a model `Ms` conforming to its metamodel `MMsource` has been developed; then, the software architect executes the generated transformation `Msource2Mtarget` producing a model `Mt`. As it can be noticed in Fig. 5, only some of the `Ms` elements have been translated into `Mt` (i.e., the gray part, technically called the *strip* (`Ms`,`Msource2Mtarget`) [19]) 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 readd those elements when closing the round-trip journey, thus obtaining a full-featured source specification, synchronized with the `Mt'` model.

More precisely, executing the `Msource2Mtarget` transformation, the part of `Ms` that is not involved by the transformation is stored in the `MLost` model. When executing the `Mtarget2Msource` transformation on `Mt'`, we obtain a model containing only the elements corresponding to `strip(Ms, Msource2Mtarget)`. Our mechanism allows us to produce a full `Ms'` model that contains also the elements that were in `Ms-strip(Ms, Msource2Mtarget)`. We call such set of elements *lost-in-translation* because they are lost during the execution of `Msource2Mtarget`.

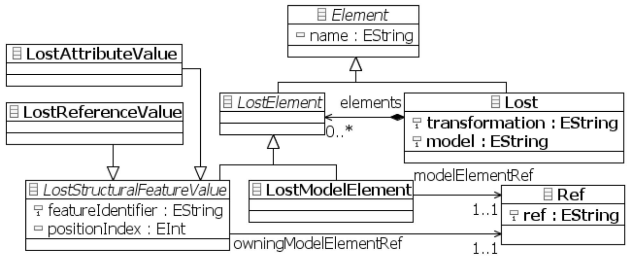

Fig. 6. Lost-in-translation metamodel.

MLost conforms to a single, generic metamodel called MMLost and represented in Fig. 6. Each model conforming to MMLost is composed of a root element containing the URI of the model (Ms in Fig. 5) and the transformation it has been generated from (instructed Msource2Mtarget in Fig. 5). The root element is composed of a set of lost-in-translation elements that may be either 1) model elements containing a reference to the corresponding model element or 2) structural features (i.e., attributes or references to other elements). Each LostStructuralFeatureValue 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 metaclass 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 also provides 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: 1) The generated transformation is "instructed" so that it returns as output a target model and an additional model containing the lost-in-translation elements and 2) the generated transformation takes as input a source model and a previously created lost-in-translation model and readds 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:

- **Not instructed**: the model transformation does not take into consideration the lost-in-translation mechanism of **DUALLY**;
- **Instructed Msource2Mtarget**: the transformation creates the additional lost-in-translation model;
- **Instructed Mtarget2Msource**: 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 **DUALLY** framework. Both Msource2Mtarget and Mtarget2Msource 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 of also checking elements that belong to the target model and making assumptions on them. Msource2Mtarget, the transformation also producing 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. Msource2Mtarget, the transformation that takes as input the auxiliary lost-in-translation model, performs a three-step task:

1. For each LostModelElement, it checks if the parent entity of the lost element in Ms still exists 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 reapplied in step 1, Msource2Mtarget restores all the values of attributes and references that do not have a default value in the original source model Ms.

3 Implementing DUALLY

We developed the current version of **DUALLY**[7] in the context of the ATLAS Model Management Architecture (AMMA) [3]. More specifically, our tool is available as a plugin of the Eclipse platform that extends the ATLAS Model Weaver (AMW) [9]. 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 metamodels are integrated into the same platform with several modeling technologies, such as Ecore (which is the metamodel of the EMF framework for describing models and runtime support for the models) and UML2 (which is an EMF-based implementation of the UML 2.x OMG metamodel for the Eclipse platform). We selected AMMA since this model management architecture is extensible and supports the concepts of weaving model and metamodel independence.

7. The home page of the **DUALLY** project is http://www.di.univaq.it/dually while the source code can be found at http://sourceforge.net/projects/dually, released under the GNU General Public License (GPL).
8. Eclipse project Web site: www.eclipse.org.
9. EMF project Web site: http://www.eclipse.org/modeling/emf/.

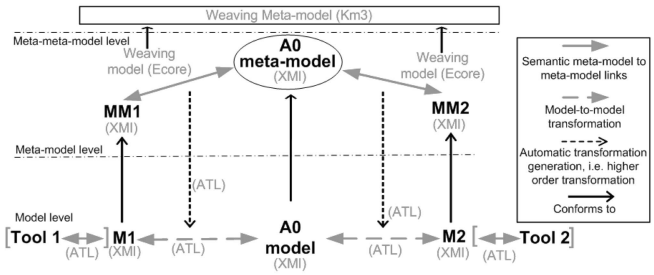

Fig. 7. Adopted modeling technologies.

A high-level overview of the technologies we used is represented in Fig. 7. Weaving models are particular kinds of models that may be used to define semantic links among metamodels or UML profiles. Both metamodels and models (also weaving models via AMW's specific editor) are expressed via XMI, while the transformation engine is based on ATL transformations [23]. Implementation details will be described in the following sections.

3.1 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) [23], which is a QVT (Query/View/Transformation)-like[10] model transformation language, with its own abstract syntax and environment. The transformation is itself a model conforming to a specific metamodel. 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) [9] is the platform that manages weaving models. It allows the definition of correspondences among models (or metamodels) and to establish semantic links among model elements. The links are saved in a weaving model, which conforms to an extensible weaving metamodel. The weaving metamodel input format is Kernel Metameta Model (KM3) [22], a language that provides a textual concrete syntax for the coding of metamodels in a Java fashion. The weaving and woven models are defined by the XML Metadata Interchange (XMI) specification, the OMG format for model and metamodels interchange. Besides, AMW is featured with a set of extension points that enable us to add specific semantics to the weaving mechanisms.
- ATLAS MegaModel Management (AM3) 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, metamodels, tools, services, and their relations as a whole, while ignoring internal details. Within one platform (local or global), a megamodel [4] 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, and metamodels) and provides user interfaces to manipulate them.
- Other peripheral tools are grouped into the ATLAS Technical Projectors (ATP) 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 an MOF-like core metamodel (i.e., Ecore) to define both models and metamodels, 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.

3.2 Realizing the DUALLY Framework

DUALLY is engineered as an extension of AMW. This extension consists of 1) a customized editor for the management of weaving models, 2) a weaving metamodel that defines the types of links that the user can establish between metamodel elements, and 3) a set of higher order transformations to automatically generate ATL transformations at the model level.

Fig. 7 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 metamodels are defined in the KM3 language. The metamodels (or profiles) are expressed in XMI, demanding their import and export to the underlying Eclipse platform. The metamodel level of Fig. 7 shows the intended passageway to integration through the A_0 metamodel.

The weaving model contains the links between elements of the *MMx* metamodel/profile and elements of the A_0 metamodel. 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 links are specified in the weaving metamodel.

As previously stated, **DUALLY** operates both on ADL metamodels and UML profiles. The metamodeler uses the Ecore formalism to define ADL metamodels. These metamodels 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 metamodel for the Eclipse platform.[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.

10. QVT (Query/View/Transformation). Object Management Group: http://www.omg.org/docs/formal/08-04-03.pdf.

11. UML2 project Web site: http://www.eclipse.org/uml2/.

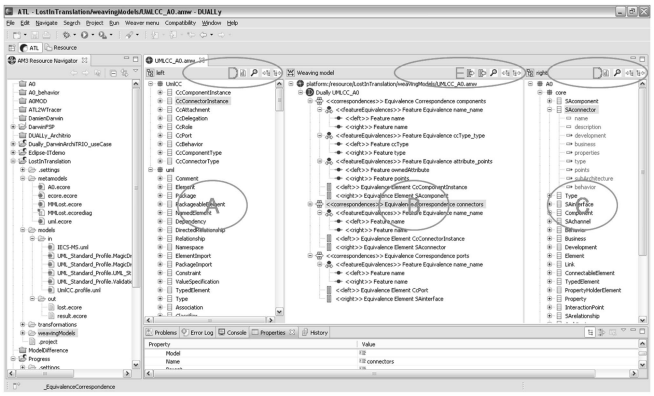

Fig. 8. Graphical interface of **DUALLY**.

As already mentioned in Section 2, there are two main users of **DUALLY**: software architects and metamodelers. 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, and 3) working on the software architecture in the target architectural language. On the other hand, for each metamodel to weave *MMx* (for example, MM1 or MM2 in Fig. 7), a metamodeler usage scenario of **DUALLY** is: 1) creating (or importing) the metamodel/profile *MMx* into Eclipse, 2) graphically developing the weaving model between *MMx* and A_0 through the **DUALLY** weaving models editor (represented in Fig. 8), and 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 metamodeler's scenario while the software architect scenario will be explained thoroughly in Section 4. The first activity of the metamodeler's usage scenario is based on the import/export mechanisms of the EMF framework; in the following sections, we explain how our technology supports the remaining activities. Section 3.2.1 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 Section 3.2.2 describes **DUALLY**'s weaving metamodel. Section 3.2.3 explains the higher order transformations of **DUALLY** and Section 3.2.4 explains the generated first-order transformations.

3.2.1 *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 metamodel panel, and right-woven metamodel panel. In Fig. 8, we illustrate how the weaving model and woven metamodels are represented within the **DUALLY**'s editor. The weaving panel (part B in Fig. 8) is a tree-based editor to create a weaving model, i.e., the mappings between elements of two metamodels (left and right). It is built on the base weaving panel provided by AMW, taking advantage of the reflective EMF capabilities; this avoided 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 metamodel (later described in Section 3.2.2); 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, with respect to the rules imposed by the metamodel.

The basic toolbar of the weaving panel of AMW was extended with two buttons (see part E in Fig. 8) 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 metamodel panels (parts A and C in Fig. 8) graphically represent left and right metamodels, respectively. These panels extend the reflective editor provided by EMF with the aim of representing the metamodels/profiles as hierarchical trees. The extension solves a problem presenting itself when dealing with hierarchical metamodels: Let us suppose that the user is creating a weaving model between a UML profile and a metamodel, namely, *leftProfile* and *rightMM*, and that 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).

Woven metamodel panels have a toolbar (part D in Fig. 8) that allows the user to render the metamodel (profile) as an Ecore diagram (UML profile diagram) taking advantage of EMF and UML2 Eclipse plugins. As mentioned earlier, the metamodels to weave form an A_0-centered star topology; therefore, either left or right-woven metamodel panel should contain A_0. 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 metamodel. In so doing, we avoid having to develop specific HOTs from Ecore metamodels 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 metamodel 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 metamodel, and so it must contain both its elements and the elements of the UML metamodel. **DUALLY** programmatically adds the UML metamodel to the profile to fill this gap. We remark that the UML metamodel is not hard-coded in **DUALLY**, but rather it is dynamically retrieved

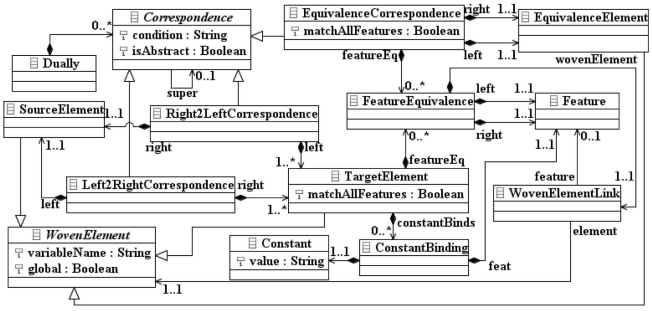

Fig. 9. **DUALLY** weaving metamodel.

from the UML2 Eclipse plugin. Therefore, **DUALLY** is autoupgrading with respect to future UML versions.

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

3.2.2 *DUALLY Weaving Metamodel*

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

This metamodel 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 metamodel by extending the AMW basic one. Our extended metamodel 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).

Fig. 9 illustrates the **DUALLY** weaving metamodel through its Ecore diagram (a graphical formalism to define Ecore-based models).

The **DUALLY** weaving metamodel 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* metamodels and a set of correspondences.
- `Correspondence`: Represents a generic mapping between elements of the woven metamodels. 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`: This element represents a unidirectional correspondence, specifically a correspondence from an element of the *left* metamodel to one or many elements of the *right* metamodel. It contains a reference to one and only one element of the *left* metamodel and a reference to one or many elements of the *right* metamodel.
  - `Right2LeftCorrespondence`: This is the opposite version of the previous element. So, it contains a reference to one and only one element of the *right* metamodel and a reference to one or many elements of the *left* metamodel.
  - `EquivalenceCorrespondence`: It represents a correspondence with bidirectional navigability. The woven elements of this correspondence are exactly two (one from the *left* metamodel and one from the *right* metamodel), and 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.

This is a peculiarity of **DUALLY**: It adds navigability semantics to weaving links. This gives us many advantages: 1) easier automatic generation of different ATL transformations from the same weaving model, 2) creation of more complex and structured weaving models, 3) strict relation between each weaving mapping and the corresponding generated ATL rule.

- `WovenElement`: It extends AMW WLinkEnd and 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`: It is the source element of a directed correspondence. For example, a *Left2RightCorrespondence* must contain a `SourceElement` belonging to the *left* metamodel.
  - `TargetElement`: It is the target element of a directed correspondence. It contains the *MatchAllFeatures* attribute and a set of `FeatureEquivalences`.
  - `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.

3.2.3 *DUALLY Higher Order Transformations*

DUALLY HOTs act as a bridge between the metamodeling and modeling levels. Fig. 10 illustrates the configuration of **DUALLY** while generating ATL transformations between MM1 and A_0. The other symmetric transformations are generated with the opposite configuration.

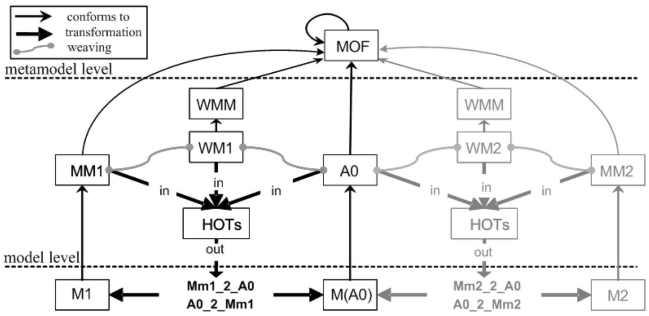

Fig. 10. Generative architecture of **DUALLY**.

The input of a HOT is composed of three models: 1) the weaving model, 2) the left metamodel, and 3) the right metamodel (referring to Fig. 10, WM1, MM1, and A_0, respectively). The output is an ATL transformation generated on the basis of the mappings defined into the weaving model (referring to Fig. 10, 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 of 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 metamodels, respectively. *elA* and *elB* contain 10 attributes with the same name. Fig. 11 illustrates two versions of

Fig. 11. Weaving features in **DUALLY**. (a) Explicit weaving of features. (b) Automatic weaving of features.

TABLE 1
Input/Output of the Generated Transformations

Source meta-model	Target meta-model	Input	Output
Ecore	Ecore	Ecore	Ecore
Ecore	UML profile	Ecore, PRO	UML
UML profile	Ecore	UML	Ecore

a weaving model that binds all of the features of the elements. The weaving model 1) contains a feature equivalence for each feature to map and 2) implicitly binds such features. Weaving model 1) has complete control on the bindings, but weaving model 2) 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* metamodel to models conforming to the *right* metamodel while the latter generates the inverse transformation.

3.2.4 *DUALLY Generated Transformations*

DUALLY automatically generates ATL transformations at the model level; we call them Basic Transformations (BT). These transformations are represented in Fig. 7 by the arrows between M1 and M(A_0) and between M(A_0) and M2. The main difference between HOTs and BTs is that the former ones operate at the metamodel level while the latter ones at the model level. Therefore, a HOT is usually executed only once for each metamodel *MMx* during the generation phase; on the contrary, BTs are executed on each model conforming to *MMx*. The HOTs dynamically verify if one of the input metamodels 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 metamodels they are generated from (Table 1). The element *Ecore* represents an MOF-compliant ADL metamodel, *UML profile* represents a UML profile, *UML* represents the UML metamodel, 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.

We faced other technical issues while generating transformations that handle the UML2 Eclipse metamodel. This is caused by the specification of UML models provided by the Eclipse UML2 platform. UML2 is an 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 metaclass 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 subsystem that manages the names of the 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.

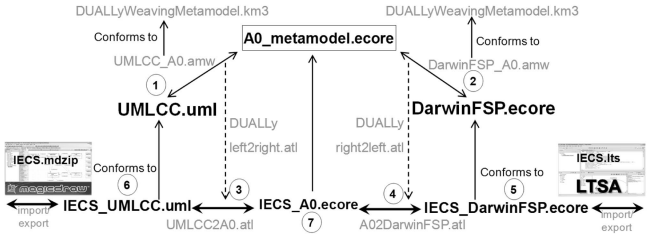

Fig. 12. **DUALLY** superstructure.

4 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:

1. the **DUALLY**zation mechanisms;
2. how the extension mechanisms work in practice (actually, a behavioral extension to A_0 has already been presented in Section 2.1.1, while this section will show its use);
3. the management of the lost-in-translation problem (i.e., how to avoid information loss during round-trip);
4. 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.[12] The final objective of this collaboration was to model check the software architecture of this system (called Integrated Environment for Communication on Ship (IECS)), and for this purpose, in [5], we used the CHARMY [32] 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).

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 [20] and make use of Darwin/FSP [26], [27] 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. Fig. 12 shows 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 metamodel and model level, respectively:

1. the MDE expert, exploiting the software architects' knowhow for conceptually defining the semantic links, develops the weaving models (points 1 and 2 in Fig. 12) and **DUALLY** automatically produces the following ATL transformations: *UMLCC2A0* (point 3 in Fig. 12) that produces models conforming to the A_0 metamodel from UML models, and *A02Darwin* (point 4 in Fig. 12) that takes as input the A_0 model (generated by *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. A_0 models (6) are the means by which the two architectural notations interoperate.

The next sections present the details of the case study.

12. Selex Communications Web site: http://www.selex-comms.com/en/.

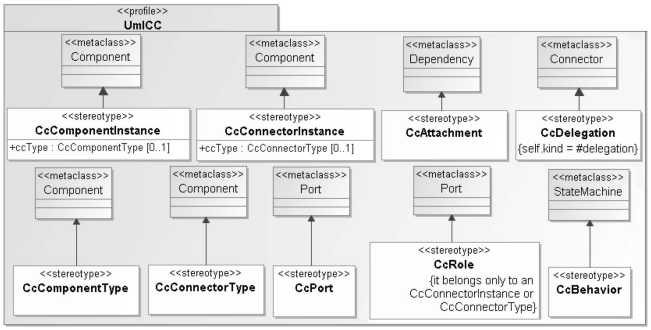

Fig. 13. *UMLCC* profile.

4.1 The *UMLCC* Profile and Darwin/FSP Metamodel

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 [20] and extending them in order to describe also connector roles, behavior, and hierarchically structured systems. The *UMLCC* profile is presented in Fig. 13.

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 modeled. A behavioral description of each component and connector can be also defined. Runtime architectural components are expressed using stereotyped UML Components called `CcComponentInstances`.

We decided to extend UML Components so that ports can be associated to them and since, in the UML2.0 metamodel, `Component` is a subtype of `StructuredClassifier`, the internal structure of each component through its inner components, connector, and attachments can be defined. Moreover, the use of 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 metaclass 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 components to the corresponding ports of the 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 has been defined to assure that `CcPorts` delegate only to `CcPorts` and that `CcRoles` delegate only to `CcRoles`. The *UMLCC* profile also provides 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, reusable component diagram. Fig. 17 shows a high-level IECS UML model conforming to the *UMLCC* profile. It will be described in Section 4.4.

Darwin and Finite-State Process notation (FSP) [24] 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 Labeled Transition System (LTS) and can be analyzed via LTSA (LTS Analyzer).[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 metamodel is presented in Fig. 14.

We constructed the joint Darwin/FSP metamodel directly encoding the two official BNFs.[14] In Fig. 14, the joint metamodel 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 its behavioral specification. This relation is therefore the sole relation bringing together the two metamodels. It should be noted,

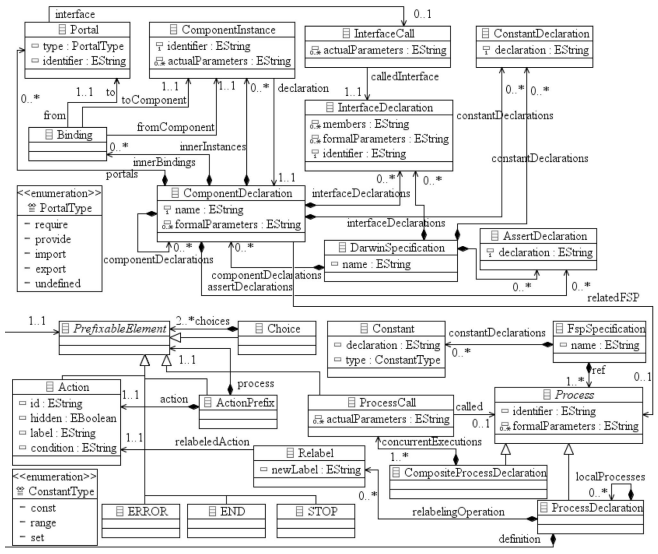

Fig. 14. Darwin/FSP metamodel.

however, that constraints defined in the comprehensive metamodel ensure that FSPs associated to components have the same alphabet of the component `Portals` [28].

Each model conforming to this metamodel is composed of two main parts: a `DarwinSpecification` and its relative `FspSpecification`.

All the elements captured in the metamodel have a counterpart in the textual notation. Once having 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 2. Since those transformations encode plain one-to-one mappings, in the following, we will not provide further details on the import/export mechanism.

4.2 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 metamodeling 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 metamodel and 2) $DarwinFSP\_A_0$ specifies the bindings between the Darwin/FSP and the A_0 metamodels.

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. Fig. 15 represents a simplified version of the $UMLCC\_A_0$ weaving model. (The full version is implemented in **DUALLY**.) Fig. 16 represents a simplified version of the $DarwinFSP\_A_0$ weaving model.

4.2.1 Properties of the Defined Weaving Models

Referring to the discussion on the correctness of transformations in Section 2.2.1, we can observe that if a bidirectional

13. LTSA tool: http://www.doc.ic.ac.uk/ltsa/.

14. **Darwin BNF**: Technical Report 08: The Darwin Language, C3DS Report, 1999. **FSP BNF**: FSP Language reference, available online within the LTSA tool's help guide.

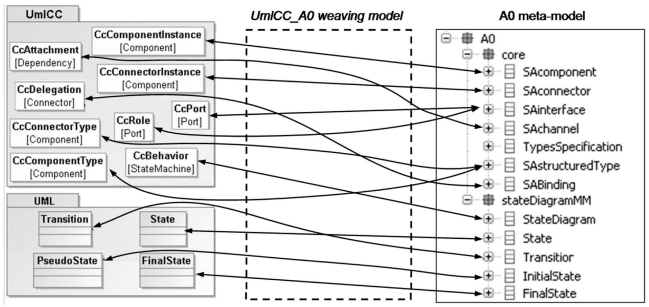

Fig. 15. $UMLCC\_A_0$ weaving model.

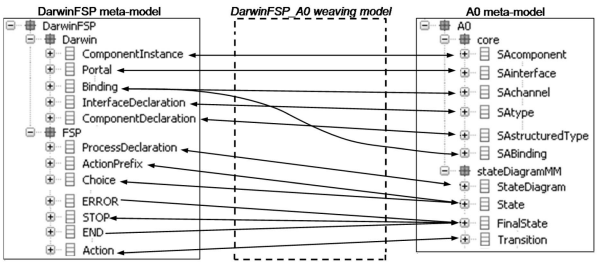

Fig. 16. $Darwin\_A_0$ weaving model.

transformation t binds two elements a (in one metamodel) and b (in the target metamodel) 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 Fig. 15, 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 taking into account other conditions that disambiguate the nondeterminism. 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 Fig. 16, 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 an `STOP` state.

4.3 Generation of ATL Transformations

The weaving models described above form the logic that generates ATL transformations. Higher order transformations take as input metamodel 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: 1) $UMLCC2A_0$ that is generated from the $UMLCC\_A_0$ weaving model and returns A_0 specifications from UML models profiled with the *UMLCC* profile and 2) $A_02DarwinFSP$ that is obtained from the $DarwinFSP\_A_0$ weaving model and returns *Darwin/FSP* models from A_0 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 $UMLCC2A_0$ 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 4.4.

```
1 module UmlCC2A0;
2 create OUT: A0, LOSTMODEL: LOST from IN : UML2;
3 helper def: matchedModelElements: Set(OclAny) = Set{};
4 helper def: matchedStructuralFeatures: Set(OclAny) = Set{};
5 ...
6 rule CcComponent2SAcomponent {
7   from
8     s: UML2!"uml::Component"    (
9       s.isStereotypeApplied('CcComponentInstance'))
10  to
11    t: A0!"A0::core::SAcomponent" (...))
12    do {
13        thisModule.matchedElements <-
14                  thisModule.matchedElements.including(s);
15        ...
16    }
17 }
18 ...
19 endpoint rule finish() {
20  to
21    t: LOST!"Lost::Lost"(
22      name <- 'IECS-MS__UmlCC2A0',
23      model <- 'IECS-MS.uml',
24      transformation <- 'UmlCC2A0.atl')
25    ...
26  do {
27    for (e in (UML2!EObject.allInstancesFrom('IN')
28          - thisModule.matchedModelElements).asSequence()){
29        t.elements <- thisModule.createLostModelElement(e);}
30    ...}
31 }
32 rule createLostModelElement(element : UML2!EObject) {
33  to
34    t : LOST!LostModelElement (
35      name <- element.name,
36      modelElementRef <- ref
37    ),
38    ref : LOST!Ref (
39      ref <- thisModule.getRef(element)
40    )
41  do {t;}
42 }
43 helper def: getRef(element : UML2!EObject) : String =
44  element.__xmiID__;
45 ...
```

The header of $UMLCC2A_0$ (line 2) specifies that this transformation takes as input a UML2 model and produces an A_0 model (*OUT*) and a Lost-in-translation model (*LOSTMODEL*). The helpers in lines 3 and 4 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 6-17) 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 **DUALLY** 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 a set is used in the `finish` endpoint rule (lines 19-31). It generates a `Lost` entity (lines 21-24) setting 1) its "name" attribute, 2) the "model" attribute to the name of the source model, and 3) 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 27 and 28 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 32-42); it creates a `LostModelElement` and the corresponding `Ref` entity is generated. The usefulness of the `Ref` metaclass in the *Lost* metamodel is clear here: Supporting different identification mechanisms consists of reimplementing just the `getRef` helper (lines 43 and 44). The name attribute of `LostModelElement` is set according to the source model element it is generated from (line 35). Attributes, which are basic data type 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.

4.4 IECS Modeling in the *UMLCC* Profile and Transformations to Darwin/FSP

The software architecture of our case study models a multitier 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) [5]. 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:

1. provide voice, data, and video communication modes;
2. prepare, elaborate, memorize, recover, and distribute operative messages;
3. configure radio frequency, variable power control and modulate transmission and reception over radio channel;
4. 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;
5. data distribution service;
6. 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.

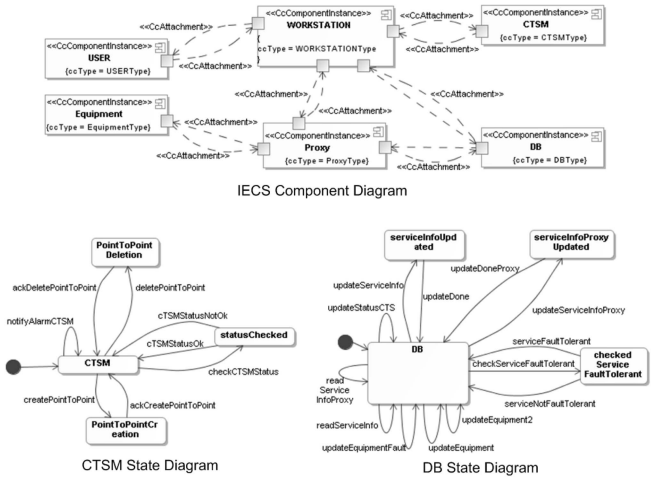

Fig. 17. IECS-MS software architecture modeled using the *UMLCC* profile.

Fig. 17 shows the overall structure of the IECS-MS architecture, as modeled by a software architect by using the *UMLCC* profile. This subarchitecture 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 Fig. 17, 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. Fig. 17 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 introducing unneeded complexity.

Once the model illustrated in Fig. 17 has been created, we give it as input to $UMLCC2A_0$ (the transformation automatically generated by **DUALLY**). Fig. 18 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 metamodel in order to properly redeploy them in the proper diagram, when moving back to the originating technology.

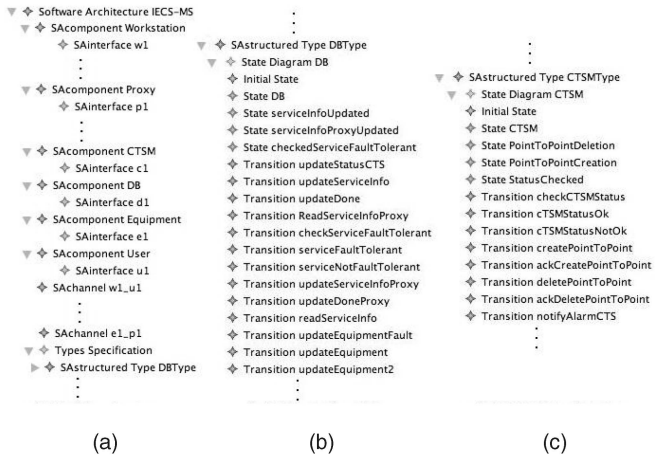

Fig. 18. A_0 model obtained from the execution of the generated transformation. (a) Static view. (b) DB behavior. (c) CTSM behavior.

Fig. 19 shows the lost-in-translation model produced by the execution of $UMLCC2A_0$ on the previously described IECS model. The generated lost-in-translation model conforms to the lost-in-translation metamodel in Fig. 6. It is composed of two parts: 1) the `LostModelElements` that are lost and 2) 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 metamodel in Fig. 4, contain states and transitions directly.

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 Fig. 20. 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 shown before, even though, for the sake of brevity, we do not show them here.

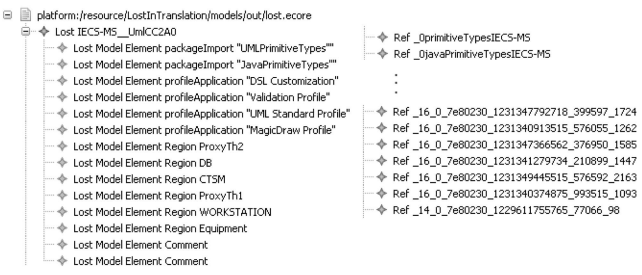

Fig. 19. Lost-in-translation model obtained from the execution of $UMLCC2A_0$ on the IECS model.

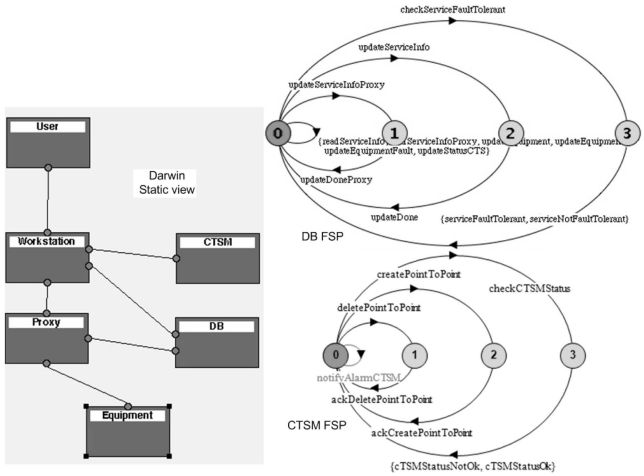

Fig. 20. Generated IECS Darwin/FSP model.

The next step is 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 shown in Fig. 20, 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, the *CTSM* component is also blocked since it awaits instructions from the *Workstation* or can raise alarms. However, these alarms cannot be managed by the blocked *Workstation*.

In order to fix the identified deadlock, the DB component has been modified, as reported in Fig. 21, 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 Fig. 12) also instructed to reapply the model elements stored in the lost-in-translation models. More specifically, $DarwinFSP2A_0$ is executed to produce the A_0 model and then the $A_02UMLCC$ transformation is executed to produce the final and updated *UMLCC* model. These two transformations automatically recover the

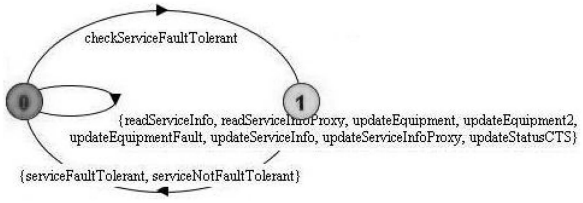

Fig. 21. Changes on the DB component.

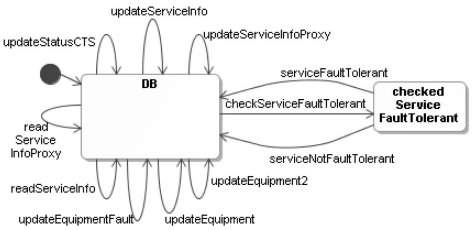

Fig. 22. The updated DB state machine.

elements lost and stored by the $A_0 2 DarwinFSP$ and $UMLCC2A_0$, respectively. Fig. 22 shows the automatically obtained *DB* state machine once reimported into MagicDraw. It becomes evident that all of 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 of the data in the original model are 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.

5 Related Work

In the following section, we delineate the ADLs as well as other technologies and frameworks that have an intent similar to **DUALLY**. This section is organized in two parts. In Section 5.1, we provide insight on a number of ADLs whose purpose we considered closer to that of **DUALLY**, while in Section 5.2, we mention other attempts as well as other closely related efforts at tackling the problems encountered and surpassed by **DUALLY**.

5.1 Surveying Related ADLs.

The Acme initiative [16] is famed for being one of the very first technologies to tackle the problem of architectural data interchange. Acme provided tooling extension points to allow other tools to physically read and write Acme descriptions. 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 ACMEStudio. A number of reasons can be presented for Acme drifting somewhat away from its initial goal, the main being 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. Contrary to **DUALLY**, software architects are obliged in Acme to explicitly write end-to-end transformations among each pair of ADLs. As described in [15], 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 [15], 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, contrary to **DUALLY**, Acme authors make clear in [15] 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 loss in translation and, with the used technology, it 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 [16].

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 metamodeling used upon integration. Summarizing, the **DUALLY** key differences are: 1) **DUALLY**'s A_0 is frozen in its essential core and provides attaching points within its metamodel so that (independent) extensions may be developed and included in the A_0 system; 2) **DUALLY**zation 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; and 3) **DUALLY**'s key base concept is to allow interchange: We stress that we want 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 [8] is an active research effort born and in progress at ISR—the 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 runtime 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 [7].

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 [25], and it also provides full compatibility and support to the MDA process. Furthermore, **DUALLY** and xADL share a modular structure designed to be extensible. However, the main goal of **DUALLY** differs from the one of xADL: 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), whereas

DUALLY is a framework for interoperability between ADLs and/or UML profiles.

AADL [12] was born as an avionics-focused DSL and later on moved to representing and supporting embedded real-time systems. The extension mechanisms of AADL include the definition of custom properties to specify additional ADL-specific analyzes 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 [13]: 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.

5.2 Related Efforts

In [21], we may find a perfect example of metainformation 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,[15] two of the previously mentioned mainstream modeling technologies. The paper shows in detail the actual implementation of the metainterchange 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 in place at both modeling and metamodeling levels.

In [1], Smeda and Oussalah 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 metalevel to describe ADLs right from their core level, rather than architectures themselves. The paper shows the undoubted potential behind the specification of this metameta 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 that we previously presented. Smeda and Oussalah [1] also provide 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 rollout, knowledge reuse, and a number of other issues will remain untouched.

15. http://www.htc.honeywell.com/metah/prodinfo.html.

6 Evaluation and Considerations

In this section, we provide an evaluation and some considerations about **DUALLY**. The discussion will be organized in two main arguments:

1. **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 metamodels/profiles and is managed by a metamodeler who retains a deep knowledge of the metamodels 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 will automatically have the system SA in each **DUALLY**zed ADL.
2. **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.).[16]

7 Conclusions and Future Work

In this paper, we presented **DUALLY**, an automated framework, that allows architectural languages and tools interoperability through model transformation technologies.

16. 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).

DUALLY's goal and rationale 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 [11]. Such transformations select a set of elements from the input metamodels 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. 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. Concluding, an ultimate future research direction is to reuse the **DUALLY** framework in contexts other than architectural languages. This implies the definition of a new A_0 customized for the new context, and thus, reusing the **DUALLY**zation technology to bind the different approaches together, as required by the new context itself.

ACKNOWLEDGMENTS

This work is partially supported by the Adaptive infRasTructure for DECentralized Organizations (ARTDECO), an Italian FIRB 2005-2009 Project. The authors would like to thank Davide Di Ruscio and Nenad Medvidovic for their valuable support and comments.

REFERENCES

[1] T.K. Adel Smeda and M. Oussalah, "Meta Architecting: Towered a New Generation of Architecture Description Languages," *J. Computer Science,* vol. 1, no. 4, pp. 454-460, 2005.

[2] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, "Model Transformations? Transformation Models!," *Model Driven Eng. Languages and Systems,* pp. 440-453, Springer, 2006.

[3] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, "Modeling in the Large and Modeling in the Small," *Model Driven Architecture,* vol. 46, pp. 33-46, Springer, 2005.

[4] J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels," *Proc. Int'l Conf. OOPSLA/Generative Programming and Component Eng.,* 2004.

[5] D. Colangelo, D. Compare, P. Inverardi, and P. Pelliccione, "Reducing Software Architecture Models Complexity: A Slicing and Abstraction Approach," *Proc. Conf. Formal Techniques for Networked and Distributed Systems '06,* pp. 243-258, 2006.

[6] K. Czarnecki and S. Helsen, "Feature-Based Survey of Model Transformation Approaches," *IBM Systems J.,* vol. 45, no. 3, pp. 621-645, 2006.

[7] E.M. Dashofy, A.V. der Hoek, and R.N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language," *Proc. Working IEEE/IFIP Conf. Software Architecture,* 2001.

[8] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, "An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages," *Proc. Int'l Conf. Software Eng.,* pp. 266-276, 2002.

[9] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas, "AMW: A Generic Model Weaver," *Proc. Première Journée sur l'Ingénierie Dirigée par les Modèles,* 2005.

[10] G. Edwards and N. Medvidovic, "A Methodology and Framework for Creating Domain-Specific Development Infrastructures," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* 2008.

[11] M.D.D. Fabro and P. Valduriez, "Semi-Automatic Model Integration Using Matching Transformations and Weaving Models," *Proc. ACM Symp. Applied Computing,* 2007.

[12] H.P. Feiler, B. Lewis, and S. Vestal, "The SAE Architecture Analysis and Design Language (AADL) Standard," *Proc. IEEE Real-Time and Embedded Technology and Applications Symp.,* 2003.

[13] P.H. Feiler, D. de Niz, C. Raistrick, and B.A. Lewis, "From PIMs to PSMs," *Proc. IEEE Int'l Conf. Eng. Complex Computer Systems,* pp. 365-370, 2007.

[14] D. Garlan and A. Kompanek, "Reconciling the Needs of Architectural Description with Object-Modeling Notations," *Proc. Int'l Conf. Unified Modeling Language,* 2000.

[15] D. Garlan, R. Monroe, and D. Wile, "Acme: An Architecture Description Interchange Language," *Proc. Conf. Centre for Advanced Studies on Collaborative Research,* p. 7, 1997.

[16] D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," *Foundations of Component-Based Systems,* pp. 47-68, Cambridge Univ. Press, 2000.

[17] D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Eng. and Knowledge Eng.,* vol. 2, pp. 1-39, World Scientific, 1994.

[18] D. Harel and A. Naamad, "The Statemate Semantics of Statecharts," *ACM Trans. Software Eng. Methodologies,* vol. 5, no. 4, pp. 293-333, 1996.

[19] T. Hettel, M. Lawley, and K. Raymond, "Model Synchronisation: Definitions for Round-Trip Engineering," *Proc. Int'l Conf. Model Transformation,* July 2008.

[20] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J.R.O. Silva, "Documenting Component and Connector Views with UML 2.0," Technical Report CMU/SEI-2004-TR-008, Carnegie Mellon Univ., Software Eng. Inst., 2004.

[21] H. Jonkers, R. van Buuren, F. Arbab, F. de Boer, M. Bonsangue, H. Bosma, H. ter Doest, L. Groenewegen, J.G. Scholten, S. Hoppenbrouwers, M.-E. Iacob, W. Janssen, M. Lankhorst, D. van Leeuwen, E. Proper, A. Stam, L. van der Torre, and G.V. van Zanten, "Towards a Language for Coherent Enterprise Architecture Descriptions," *Proc. Int'l Enterprise Distributed Object Computing Conf.,* 2003.

[22] F. Jouault and J. Bézivin, "KM3: A DSL for Metamodel Specification," *Proc. Int'l Conf. Formal Methods for Open Object-Based Distributed Systems,* 2006.

[23] F. Jouault and I. Kurtev, "Transforming Models with ATL," *Satellite Events at the MoDELS 2005 Conf.,* Springer, 2006.

[24] J. Kramer and J. Magee, *Concurrency: State Models and Java Programs.* John Wiley and Sons, 1999.

[25] P. Kruchten, "Architectural Blueprints—The "4+1" View Model of Software Architecture," *IEEE Software,* vol. 12, no. 6, pp. 42-50, Nov. 1995.

[26] J. Magee, "Behavioral Analysis of Software Architectures Using LTSA," *Proc. Int'l Conf. Software Eng.,* pp. 634-637, 1999.

[27] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," *ACM SIGSOFT Software Eng. Notes,* vol. 21, no. 6, pp. 3-14, 1996.

[28] J. Magee, J. Kramer, and D. Giannakopoulou, "Software Architecture Directed Behaviour Analysis," *Proc. IEEE Int'l Workshop Software Specification and Design,* 1998.

[29] N. Medvidovic, E.M. Dashofy, and R.N. Taylor, "Moving Architectural Description from Under the Technology Lamppost," *Information and Software Technology,* vol. 49, pp. 12-31, 2007.

[30] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins, "Modeling Software Architectures in the Unified Modeling Language," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 1, pp. 2-57, 2002.

[31] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 70-93, Jan. 2000.

[32] P. Pelliccione, P. Inverardi, and H. Muccini, "CHARMY: A Framework for Designing and Verifying Architectural Specifications," *IEEE Trans. Software Eng.,* vol. 35, no. 3, pp. 325-346, May/June 2009.

[33] D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes,* vol. 17, pp. 40-52, 1992.

[34] M. Pinto, L. Fuentes, and J.M. Troya, "DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development," *Proc. Int'l Conf. Generative Programming and Component Eng.*, pp. 118-137, 2003.
[35] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Ivano Malavolta is working toward the PhD degree in computer science in the Computer Science Department at the University of L'Aquila, Italy. His research interests include software architecture languages (ADLs), architectural interchange and interoperability between software architecture notations, and MDE techniques.

Henry Muccini is an assistant professor at the University of L'Aquila, Italy. His research interests include software architecture modeling and analysis, component-based systems, model-based analysis and testing, and global software engineering education. He has published various conference and journal articles on these topics, coedited two books, and co-organized various workshops on related topics. He serves as a program committee member and reviewer for many international conferences and journals. He is currently locally coordinating the Global Software Engineering European Master (GSEEM) and an Erasmus Mundus External Cooperation Window Project. More information is available at http://www.henrymuccini.com.

Patrizio Pelliccione received the PhD degree from the Computer Science Department at the University of L'Aquila, where he is now an assistant professor. His research interests include software architecture modeling and analysis, component-based systems, fault tolerance, middleware, model checking, and formal methods. He has published various journal and conference papers on these topics and serves as a PC member and reviewer for several international conferences and journals.

Damien Andrew Tamburri is working toward the MSc degree at the University of L'Aquila in the Computer Science Department, where he is also engaged in several research initiatives. His research interests include architecture description languages (ADLs) and architectural description, as well as reverse engineering, and architecture-driven modernization and architectural recovery.