

DUALLY: Putting in Synergy UML 2.0 and ADLs

H. Muccini, P. Inverardi
Dipartimento di Informatica
University of L'Aquila
Via Vetoio, 1 - L'Aquila, Italy
muccini, inverard@di.univaq.it

P. Pelliccione
Software Engineering Competence Center
University of Luxembourg
6, rue R. Coudenhove-Kalergi, Luxembourg
patrizio.pelliccione@uni.lu

Abstract

Many formal languages to describe software architectures (SA) have been proposed so far, but only very few of them are still supported and used in practical contexts. Many UML profiles and extensions have been provided when UML became a standard, in order to model as much as possible architectural concepts. They allow to specify specific needs, but the effort in creating a sort of unified UML-based language for architectures is still not paying. In fact, since different communities require different information to be put into a diagram, depending on which architectural design aspects should be represented and analyzed, the idea of an unified UML language for SA seems to be not adequate. Building on these considerations, we propose **DUALLY**, a core set of UML concepts, well suited for SA modeling, together with a framework which provides extensibility mechanisms to adapt the initial notation, in order to meet different needs.

1 Introduction

When software engineers realized the importance of Software Architecture (SA), they started proposing languages to specify it. Informal box-and-line notations have been rapidly replaced/complemented by architectural languages. In a few years, the SA community has observed a proliferation of architecture description languages (ADLs) [28] for rigorous and formal SA modeling and analysis, and the introduction of supporting tools. In parallel, the concept of *viewpoint* modeling [25, 20, 26] has been introduced in order to tackle the problem of modeling SA from different perspectives, thus reducing modeling complexity.

Although several studies have shown the suitability of such formal languages for SA modeling and analysis, industries still tend to prefer model-based (semi-formal) notations. In particular, with the introduction of UML as the de-facto standard to model software systems and its wide-

spread adoption in industrial contexts, many extensions and profiles have been proposed to “adapt” UML to model architectures (e.g., [16, 27, 18, 24, 22, 36, 31]). However, while such UML notations are well suited to modeling some aspects of software architectures, they fail in modeling others [13]. These extensions permit to reduce the gap between UML and ADLs, but they still fail in representing all aspects of ADLs [13, 28, 27]. Moreover, based on our academic and industrial experience on SA modeling for analysis, we realized that *modeling for documenting is quite different from modeling for analysis*, and *different analysis techniques usually require different notations*. There is not best formalization since it depends on which aspects of architectural design we want to represent and then analyze. Any time a new slightly different analysis is required, new modeling concepts are needed. This suggests there is neither a unique language for representing SAs, not a unique fit between UML and ADLs.

Then, can we really advocate a stronger synergy between UML and ADLs can be created? We believe this possible and we suggest to approach the problem from a different perspective: *i*) our starting point consists in *identifying a core set of architectural elements always required*; then, *ii*) *we create an UML profile able to model the core architectural elements previously identified*. *iii*) *We provide extensibility mechanisms to add modeling concepts needed for specific analysis*. Finally, *iv*) *we describe how semantic links mechanisms can be kept between different notations*.

From existing ADLs and from our experience on modeling and analysis of SA, we observe that there are some core architectural concepts commonly understood by any software architect, while other modeling elements are needed for specific analysis. We inherit from both xArch [1] and ACME [7] the idea of identifying a maximum common denominator (a core set) of architectural concepts, called hereafter A_0 (point *i*). Then, we create the **DUALLY** UML profile to model such core architectural concepts. Since we cannot expect to create a unique fit between UML modeling elements and architectural concepts, instead of providing a

thorough and precise UML-SA profile, we propose minimal extensions to UML 2.0 in order to make the profile applicable in industrial contexts (point *ii*). Since our profile covers only core architectural concepts, **DUALLY** provides extensibility mechanisms, at different levels: the UML profile notation for core elements can be extended with additional modeling elements. New diagrams can be introduced. Existing modeling and analysis tools can be integrated. The first two extensions are performed through UML profiling and meta-modeling extensions. The latter extension is supported by a plugin framework, which allows to easily add new design or analysis tools (point *iii*). Finally, **DUALLY** permits to specify semantic links, which allow to specify semantic equivalences/differences among model elements. In this paper, we focus on points *i* and *ii*), while providing initial results on *iii*) and *iv*).

DUALLY differs from previous work on ADLs and UML modeling for many reasons: while related work on ADLs mostly focus on identifying “what to” model [28, 7, 1], **DUALLY** identifies both “what to” model (i.e., the core architectural concepts) and “how to” model (via the **DUALLY** UML profile). Differently from related work which extend UML for modeling specific ADLs, the **DUALLY** UML profile focusses on modeling just the minimal set of architectural concepts. Similarly to xArch and ACME, **DUALLY** provides extensibility mechanisms to facilitate modeling extensions.

The paper describes the state of the art in SA modeling in Section 2: we focus on formal languages and UML-based notations for SA specification. Section 3 describes **DUALLY**, the main principles and current work. Section 4 shows initial results on applying **DUALLY** to a specific notation for SA analysis. Section 5 concludes the paper.

2 Modeling Software Architecture

Two main classes of languages have been used so far to model software architectures: formal architecture description languages (ADLs) and model-based specifications with UML. In this section we identify a core set of architectural elements always required.

Section 2.1 analyzes which concepts a formal ADL should *minimally* support. This analysis takes into consideration existing ADLs, still supported ADLs and our experience. This analysis guides the identification of core architectural concepts, represented by the A_0 area in Figure 1.

Section 2.2, instead, outlines how UML 1.x has been used for SA modeling, the novelties in moving from UML 1.x to UML 2.0, and how UML 2.0 is currently used to model SAs.

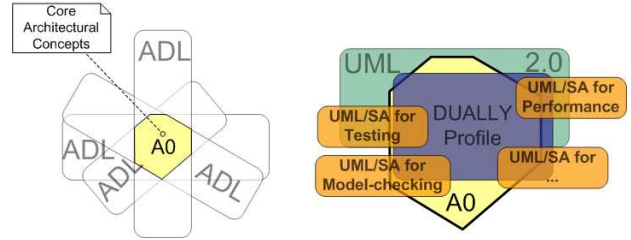


Figure 1. Core Architectural Concepts

2.1 Formal Architecture Description Languages for SA modeling

When the software architecture field was in its infancy, software architects started suggesting what an (ideal) ADL should consider: support for components and connectors specification, and their overall interconnection [32, 17], composition, abstraction, reusability, configuration, heterogeneity and analysis mechanisms [37].

Then, many ADLs have been proposed, with different requirements and notations, and permitting different analysis at the SA level. New requirements emerged, such as hierarchical composition, type system, ability to model dynamic architectures, ability to accommodate analysis tools, traceability, refinement, and evolution. New ADLs have been proposed to deal with specific features [4, 33], such as *configuration management*, *distribution* and suitability for *product line architecture* modeling. Structural specifications have been integrated with behavioral ones [26, 15] with the introduction of many formalisms such as pre- and post-conditions, process algebras, statecharts, POSets, CSP, π -calculus and others [28].

In order to make some clarification on what an ADL is and on how they may be classified, papers have been proposed to survey, classify and compare existing ADLs. In particular, Medvidovic and Taylor in [28] proposed a classification and comparison framework, describing what an ADL must explicitly model, and what an ADL can model. In their work, an ADL must explicitly model components (with interfaces), connectors, and their configurations. A similar study has been performed for producing xArch [1], an XML schema to represent core architectural elements. In xArch, an architecture is minimally composed by components and connectors (with interfaces), links, and groups. ACME [7], the architecture interchange language, also identifies a set of core elements for architecture modeling, with components, connectors, ports, roles, properties and constraints.

In the following of this section, we identify a list of core architectural concepts (A_0 area in Figure 1) by extending/revising/updating previous work in order to take into account architectural concepts supported by still in use ADLs

(Section 2.1.1) and our experience on SA-based modeling and analysis (Section 2.1.2).

2.1.1 Still supported Formal Architecture Description Languages

Table 1 shows the most famous ADLs evidencing the ones still supported. The table contains also approaches that, since lack in some aspects that are considered fundamentals for an ADL, are considered non conventional ADL. Moreover several ADLs focusses on very specific contexts, as for example the avionic context.

Taking into consideration non domain-specific languages for SA, the surviving ADLs are ABC/ADL, ACME, Darwin, DAOP-ADL, Jacal, Koala, and Rapide together with the ADML, xArch, xADL, and xACME XML representation formats.

We believe that one of the reason which made those ADLs survive, is because software architects found on them some concrete help with their quotidian problems in modeling architectures.

ABC/ADL is an architecture language recently proposed to support component composition. *Acme*, born as an architecture exchange language, today has become a well supported ADL, tool supported by AcmeStudio and integrated with Java code generation features [8]. *Darwin* describes software structures in terms of primitive and composite components, encouraging hierarchical composition. Darwin possesses both a textual and graphical notation and a process algebra (the Finite State Process (FSP)) permitting to model behavioral aspects of an SA and allowing for analysis at the architecture-level. *DAOP-ADL* [33] is a component- and aspect-based language to specify the architecture of an application in terms of components, aspects and a set of plug-compatibility rules between them. *Jacal* is an ADL with the aim of permitting the execution (animation) of described architectures. *Koala* is an architecture language for software product line designed for modeling embedded software for consumer electronics. *Rapide* is a general-purpose system description language that allows modelling of component interfaces and their externally visible behavior.

ADML [6] (the Architecture Description Markup Language) is an XML-based representation language for architecture, originally developed by the Micro-electronics and Computer technology Consortium, based on Acme. *xArch* is an extensible XML-based notation for software architectures. *xArch* has been thought to be extensible, that is, to be augmented with new XML schemas to introduce additional information. Both *xADL2.0* [4] and *xAcme* [5] are extensions to the *xArch* core to handle configuration management and product families, and Acme architectural concepts (respectively).

2.1.2 What an architectural language should support: our Experience

The authors of this paper are members of the Software Engineering and Architecture Group (SEA Group), which mostly works on software modeling and analysis at the architecture level. Based on our experience on SA-based analysis in industry, we here summarize what we consider to be fundamental aspects in SA modeling:

ADLs concepts and UML modeling: an architecture language to be used in industrial contexts, needs to expose ADLs architectural concepts, while using UML as the modeling notation. In this way, UML is used as the modeling language complying with architectural concepts exploited by ADLs.

Structural and Behavioral Viewpoint: whenever functional analysis at the architectural level is required, behavioral models need to be provided [11, 15]. In our experience with SA-based testing [29], state machine or Labelled Transition System models are needed to express functional properties of the SA. In our work on SA-based model-checking [2], state machine-based notations are used to specify the SA behavior while scenarios are used to identify properties of interest. In [12] we may notice how scenarios are used for reliability and performance analysis. In the Fujaba project [3], UML class diagrams and UML state diagrams are combined to provide a specification language for hard real-time systems.

Tool support: whenever an architecture has to be specified in industrial contexts, tool support is a must. As pointed out in Section 2, while many formal or UML-based languages have been proposed so far, very few tools are available. In the past few year, we have developed a tool called CHARMY [2] to model and analyze architectural specifications. Its use has evidenced the importance of tool support in our modeling and analysis activity in industrial contexts.

Different abstraction levels in the same specification: some ADLs provide features to specify nested components and hierarchical architectures. In our experience, we realized the importance of such features, and the importance of being able to specify an SA at different abstraction levels. This is due to different factors: *i*) a given specification may be too abstract to be useful for analysis purposes. However, moving towards next abstraction levels (i.e., providing a more detailed specification) the system global model may become too complex to be analyzed as a whole. *ii*) We cannot assume components to have the same complexity. While some components may be small computational units, others may be complex subsystems. *iii*) We cannot assume components have the same importance. Some components may be of higher interest for analysis with respect to others. *iv*) Different components may require different specification types. In fact, while some components may require black- or white-box specifications.

ADL	Born Data	Tools	Still Supported	Notes
ABC/ADL	2002	ABC tool (prototype)	YES	ADL for component composition
Acme	1995	AcmeStudio + Armani	YES	Interchange Language between ADLs
Adage	1992	—	NO	Avionics navigation and guidance Architecture Description
Aesop	1994	Aesop System	NO	Emphasis on Architectural Styles
ArTek	1994	—	NO	Non conventional ADL
C2 SADEL & C2 AML	1996	Dradel, SAAGE, ArchStudio	NO	ADL based on C2 style
Darwin	1991	LTSA + SAA	YES	Focus on dynamic SA
DAOP-ADL	2003	DAOP-ADTools	YES	Component and Aspect-based ADL
GenVoca	1996	P3	NO	Non conventional ADL
Jacal	1997	Jacal 2	YES	Focus on prototyping SA
Koala	1997	Koala tools	YES	ADL for product families
LILEANNA	1993	LILEANNA	NO	Modules connection language
Little-JIL	1998	Little-JIL 1.0	NO	Non conventional ADL
Maude	1998	Maude 2.0	YES	Non conventional ADL
MetaH & MetaS	1993	MetaH	YES	ADL for avionic domain
Rapide	1990	Rapide	YES	ADL and simulation
Resolve	1994	Resolve	NO	Focus on Component Specification
SADL	1995	Sadl tool	NO	Focus on Refinement
UniCon	1995	UniCon	NO	Focus on connectors and Styles
Weaves	1991	Weaves	NO	Data-flow-architectures with high-volume of data
Wright	1994	Wright	NO	Focus on communications
ADML	2000	ADML Enabled Tools	YES	XML-based ADL
xArch/xAcme	2001	AcmeStudio	YES	Acme in XML
xArch/xADL	2000	xADL 2.0	YES	XML-based ADL

Table 1. Still supported ADLs and XML representations

Typed components and connectors: a type system permits to semantically group architectural elements with the same properties and constraints. When a type system is missing, properties and constraints must be assigned to each single element, thus increasing the effort in SA modeling, and making much more difficult maintainability when the SA evolves.

2.1.3 Summarizing: what goes in the Core architectural concepts A_0

Based on what expressed in this section, we claim that the minimum concepts a language for modeling architectures must take into consideration are the following: components (with required and provided interfaces, types and ports), connectors (with required and provided interfaces and types), channels, configuration (with hierarchical composition), tool support, and behavioral modeling.

2.2 The Unified Modeling Language for SA modeling

Many proposals have been presented so far to adapt UML 1.x to model software architectures [34, 27, 16, 25, 20, 18, 23, 36]. In the seminar paper proposed by Robbins et al. [34], UML has been extended in order to make UML semantically equivalent to the C2 architectural style. Garlan and Kompanek in [16] proposed five approaches for

modeling software architectures, considering classes, objects, components and packages to model SA components. The concept of software views/perspectives has been implemented inside UML with many proposals [20, 9]. Since such initial works, many other papers have compared the architectural needs with UML concepts, extended or adapted UML, or created new profiles to specify domain specific needs with UML. Even if such extensions allow to reduce the gap between UML and ADLs, they cannot fully represent all aspects/features of every ADLs, neither there is a unique way to model a single ADL [13, 16, 28]. A good analysis of UML1.x extensions to model SAs can be found in [27].

With the advent of UML 2.0, many new concepts have been added/modified to bridge the gap with ADLs. Five fundamentals novelties have been introduced for SA descriptions:

1. In UML 2.0 a *component* is a subtype of class¹, then it may now have attributes and methods, as well as interfaces, ports and internal structure.
2. UML 2.0 extends UML1.4 concept of *interface*, by explicitly including *provided* and *required* interfaces, which allow to wire components together.
3. *Ports* describe how components interact with the environment. A port specifies that any request arriving to the

¹differently from UML 1.4 where a component is defined as a subclass of Classifier

port itself, will be handled by the component which owns the port. The interfaces associated with a port, specify the nature of the interaction that may occur over the port. A port can be enriched with a behavior description.

4. Decomposition of components, useful to represent the internal structure, is performed thanks to *structured classifier*. Structured components can be composed with interfaces and ports.

5. A UML 2.0 *connector* is a communication link between two or more instances. There exist two kind of connectors: (i) *assembly* connector represents the binding between a provided interface and a required interface or port; (ii) *delegation* connector binds an internal representation of a component with its external behavior, i.e., maps a request from one port to either another port or to a realization that provides the implementation for the request. Ports and connectors formalize the interaction points on a component, and provide explicit mapping between published interfaces and internal implementation mechanisms (realizations).

How to use UML 2.0 (as is) for SA modeling has been analyzed in some books. In [14], how UML 2.0 can be used to represent logical architectures, patterns and physical architectures. The UML 2.0 concepts of components, dependencies, collaborations and component and deployment diagrams are used. In [30], components in a component diagram are used to model the logical and physical architecture. In order to bridge the gap between UML 2.0 and ADLs, some aspects still require adjustments. Therefore, much work has been proposed in order to adapt and use UML 2.0 as an ADL [19, 35, 22, 31].

3 DUALLY

The main limitation of existing ADLs is that each of them allows for a particular analysis technique, leaving other techniques unexplored: supposing an industry is interested in deadlock analysis and performance analysis, a complete result is obtained only using two different ADLs. The process is complicated and made worse by the fact that each ADL uses a different notation for SA specification, thus making difficult any integration.

Two problems hamper the success of strategies based on traditional ADLs: 1) languages used by ADLs are generally formal and sophisticated, making difficult their integration in industrial life-cycles, 2) it is impossible to construct an ADL able to support every kind of analysis, since any analysis technique requires additional analysis-specific notations and models.

DUALLY merges UML 2.0 and ADLs through a UML-based notation for Software Architecture descriptions. This choice is motivated by the fact that UML is the de-facto standard to model software systems and it is largely approved by the industries. The definition of the UML profile,

able to model the core architectural elements identified in Section 2.1.3, allows to solve problem 1. The profile for SA modeling and analysis is presented in Section 3.1.

To overcome problem 2 we outline an extendible framework that allows to add models and to extend existing ones in order to support the introduction of analysis techniques. Semantic relations are proposed to bind different elements of different models. Semantic links will allow to specify semantic equivalences/differences between model elements. Extensibility mechanisms and semantic links are outlined in Section 3.2 and will be refined in future work.

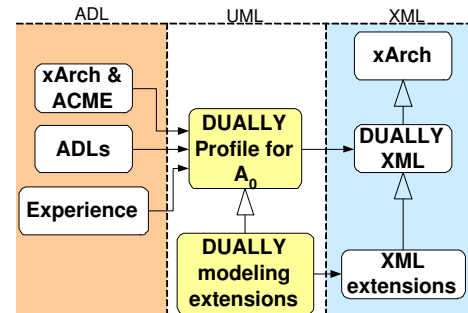


Figure 2. The roles of ADL, UML and XML

Figure 2 summarizes how **DUALLY** is organized. Exploiting ACME as interchange language, all the ADLs proposed till now, xArch, and the feedback generated by our experience in SA analysis, we build a basic UML profile for SA analysis (the **DUALLY Profile for A_0**). While the UML profile for A_0 represents the (core) modeling notation, an extension to xArch represents the XML representation format. Whenever a new UML-based notation is required for a different analysis, a new profile is built by extending the **DUALLY** profile and its XML representation is obtained by extending the **DUALLY XML** representation.

3.1 The UML profile for SA description

Goal of the **DUALLY** UML profile is to extend UML 2.0 in order to model core architectural concepts described in Section 2.1.3. This profile is not meant to create a perfect matching between UML and architectural concepts. Instead, it wants to provide a practical way, for software engineers in industry, to model their software architectures in UML, while minimizing effort and time and reusing UML tools. We analyze how UML 2.0 standard notation can be used for SA modeling of core architectural concepts and propose some needed extensions.

Architectural components: an SA component can be mapped into both structured classes and UML components. Even if there is a little distinction between the two considered elements in UML 2.0, we use UML components for

modeling SA components since a component contains all class elements (a component is a subtype of class), plus, it is possible to attach deployment descriptors, property files and any kind of document to a component. Moreover, it is more natural to map SA components into UML components because they have the same name and a similar granularity. The “Dependency” relationship between components in UML 2.0 may be used to identify relationships among components, when interface information or details are missing or want to be hidden. “Structured classifiers” permit the natural representation of architecture hierarchy and ports provide a natural way to represent runtime points of interactions. As noticed in [35], SA components and UML components are not exactly the same, but we believe they represent a right compromise.

Ports and Interfaces: UML 2.0 as is explicitly models ports to formalize components interaction points. Moreover, UML component diagrams also explicitly handle required and provided interfaces. In summary, UML 2.0 provides native notations for ports and interfaces.

Types: a type model identifies properties and constraints which must be applied to a set of elements of the same type. UML 2.0 offers the concept of Collaboration, which explains the component role (its properties) in achieving a joint task with other classifiers. We believe the use of collaborations and roles to model types is not appropriate and a more explicit notation for types is required. In [35] the authors provide a well thought solution, however, it seems to be of difficult (practical) application. In [22] the authors propose to use class or components diagrams to describe types and instance diagrams to define component and connector instances. Our proposal refines this one, by using component diagrams and property files to describe component types and instance diagrams to identify architectural elements instances.

Connectors: in current architectural languages, a connector represents communication glue and may include a set of interfaces or properties. While a connector is frequently used to capture single connecting lines (such as channels), they may also serve as complex run-time interaction coordinators between components. UML 2.0 provides the concept of *delegation* connector and *assembly* connector. However, the UML 2.0 definition of connector is far away from the SA definition of connector. An extension of UML 2.0 connectors is the solution proposed in [35]. However, we do not believe this solution is easily applicable. Many other alternatives may be chosen. For example, a Port may have an associated behavior, and a protocol state machine may describe the conditions under which the port may be used successfully [30]. However, binding the concept of port as connector with the one of component does not seem a reasonable solution. Reasoning about the proposal presented in [22], three different alternatives are proposed: using UML as-

sociations of assembly connectors, using UML association classes, or using UML (stereotyped) classes. While the first solution for modeling architectural connectors may be used to model the concept of Channel, the make use of a solution lied to the third solution proposed in [22], but, instead of using UML (stereotyped) classes, we make use of UML (stereotyped) components that, from the architectural point of view, seems the cleanest choice. The second solution (i.e., using association classes) is dismissed in our view since it requires information similar to the one needed in the third solution, while introducing multiplicity under-specifications, as discussed in [30], ch. 6.

Channels: a channel is usually considered as a simple binding mechanism between components, without any specific logic. UML 2.0 provides the concept of *assembly* connectors which is semantically equivalent to the concept or architectural channel. We assume in our formalism assembly connectors are attached to ports (with or without associated interfaces). When refining an SA, the need of converting channels into connectors may appear. In this case, the assembly connector is replaced by a stereotyped component, and the multiplicity is adjusted in conformance to what suggested in [30], ch. 6.

Configurations: an architectural configuration shows the overall structure of the SA, in terms of components and connectors instances assembly. In UML 2.0, component instances may be created and assembled through ports, interfaces and architectural connectors.

Behavioral viewpoint: depending on the kind of analysis required, state-based machines or scenarios notations are usually utilized to specify how components and connectors behave. As a core element, we take UML 2.0 state machines and sequence diagrams as native notations for behavioral modeling.

The **DUALLY** modeling profile XML representation is performed by extending the xArch XML Schemas.

3.2 Extensibility mechanisms and Semantic Rules

The **DUALLY** modeling profile can be used as a reference modeling notation to describe architectures. In a basic scenario, a software architect can document its architecture by drawing a diagram, can check the diagram compliance to the **DUALLY** profile, and can run analysis tools to validate the architectural model.

Indeed, what we expect in a more typical scenario is that the software architect needs a more expressive architectural model, for documentation or analysis purposes. **DUALLY** extensibility mechanisms allow to handle this scenario, by permitting to add new modeling elements, new diagrams, or integrating different analysis tools.

Whenever a new notation wants to be added in

DUALLY, three activities are required: *i*) a new UML profile has to be written, by extending the **DUALLY** profile to take into account the new desired notation. *ii*) Its XML representation needs to be created, extending the **DUALLY** XML Schema. *iii*) Tool support for drawing diagrams and to perform analysis has to be provided.

The modeling extension is realized through UML profiling and meta-modeling extensions (point *i*). Whenever a new conceptual element is needed, UML extension mechanisms (tagged values, stereotypes, and constraints) can be utilized to create a new profile. The profiling solution is a lightweight extension to UML, since it does not require to change the UML metamodel, thus allowing to maintain the compatibility with existing UML tools. Some existing UML tools (e.g., MagicDraw) allows to create a new profile or extend an existing profile. However, UML extension mechanisms can be utilized only when the new modeling element extends (via inheritance) an existent element. Whenever this condition does not apply, meta-modeling extensions are required.

The **DUALLY** XML schema requires to be extended in order to take into account the new UML modeling concepts (point *ii*). Since **DUALLY** extends xArch, **DUALLY** XML Schema can be easily extended by using existing XML tools.

Finally, tool support can be provided to edit and analyze the new required notation (point *iii*). The analysis tool integration is supported by a plugin framework, which allows to easily add new analysis tools. The plugin framework allows to plug different editors or analysis tools.

Figure 3 summarizes how the **DUALLY** architecture handles such extensions. Initially, the **DUALLY** architecture contains the **DUALLY** profile, the **DUALLY** XML Schemas, and tools for editing and analysis of architectural models written in the **DUALLY** profile (Figure 3.a). When a new profile *P* is created and wants to be inserted in **DUALLY** (Figure 3.b), an input filter is created to validate the *P*'s conformance to the **DUALLY** profile, and to distinguish the *P*'s specific modeling concepts from the **DUALLY** basic modeling concepts (Figure 3.c). It is important to notice that an input filter is required for each notation extending the **DUALLY** profile. The input filter generates from the *P*'s model a **DUALLY** model: a model-to-model transformation is utilized in this phase to convert *P*'s models into **DUALLY** models (Figure 3.d). The transformed model can be analyzed with **DUALLY** standard analysis tools. New modeling and analysis tools can be introduced to deal with *P*. The introduction of new tools is handled by a plugin-based framework which allows to extend the **DUALLY** standard features.

The introduction of new models requires to define semantic relations between the added/modified models and the models already contained in **DUALLY**. This seman-

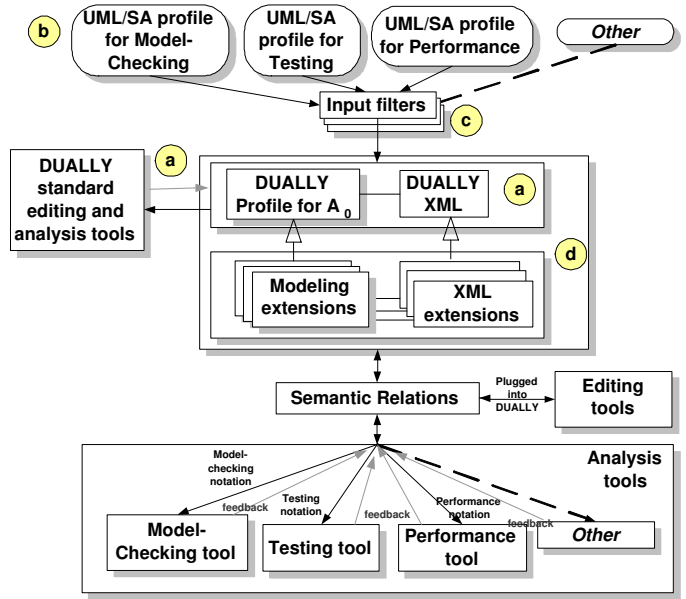


Figure 3. DUALLY Software Architecture

tic relations allows to use the same modeling and analysis framework (i.e., **DUALLY**) to perform many different analysis, by exploiting equivalences/differences among UML-SA notations. In particular, it allows to embed the feedback resulting from analysis into the models.

For example, it is intuitive that upon detecting a deadlock in a software model, a critical component may be split in two components, and this refinement may heavily affect the software performance. Traditionally, we have to select an ADL which allows for deadlock analysis (P1) and an ADL which allows for performance analysis (P2). We have to provide two specifications in two different languages. We have to run the deadlock analysis tool, get the feedback, modify the SA and then modify the performance specification to take into account such changes.

With **DUALLY**, instead, we create the two UML profiles for P1 and P2. The input filters allow to transform P1 and P2 models into standard **DUALLY** models which can be submitted to **DUALLY** standard analysis. Then, P1 is submitted to deadlock analysis. If some core architectural elements in P1 are modified, the *Semantic Relations* component propagates this change to the core part of P2. If changes in P1 affect only P1-specific elements, than such changes do not affect P2. The relations that allow the *Semantic Relations* component to propagate the changes are embedded into each UML/SA profile for the specific analysis. In fact, since each profile must extend the **DUALLY** profile, it is implicitly required to relate the new introduced models with the ones contained into **DUALLY** (e.g. the behavior of the components specified in a specific profile is related to the **DUALLY** core components). These re-

lations are exactly what we mean to be a semantic relation. When, by using the created input filter, the modeling extensions and the XML extensions are accepted by the **DUALLY** core, the semantic relations component is able to calculate the relations that regulate the propagation feedback forwarding.

4 Introducing CHARMY into DUALLY

CHARMY [2] is a project whose goal is to apply model-checking techniques to validate the SA conformance to certain properties. In CHARMY the SA is specified through an UML profile where state diagrams are used to describe how architectural components behave while sequence diagrams are used to explain how components interact.

How CHARMY works

Starting from these dynamic views, CHARMY synthesizes, through a suitable translation into Promela (the specification language of the SPIN [21] model checker) an actual SA complete model that can be executed and verified in SPIN. This model can be validated with respect to a set of properties, e.g. deadlock, correctness of properties, starvation, etc., expressed in Linear Temporal Logic (LTL). Instead of writing directly temporal properties that is a task inherently error prone, CHARMY allows to describe temporal properties by using sequence diagrams that are successively translated into a temporal properties representation understandable by SPIN. To describe state and sequence diagrams the used notation encompasses those used in current software development practice and rich enough to allow for analysis. Figure 4 illustrates the notation CHARMY uses for state diagrams and scenarios.

State Diagrams

The state machine-based formalism used by CHARMY is a subset of UML state diagrams, and is graphically shown in Figure 4.a: labels on arcs uniquely identify the architectural communication channels, and a channel allows the communication only between a pair of components. The labels are structured as follows:

$$['guard']event('parameter_list')/op_1; op_2; \dots; op_n$$

where *guard* is a boolean condition that denotes the transition activation, an *event* can be a message sent or received (denoted by an exclamation mark “!” or a question mark “?”, respectively), or an internal operation (τ) (i.e. an event that not requires synchronization between state machines).

Both sent and received messages are performed over defined channels *ch*. An event can have several parameters as defined in the parameters list. op_1, op_2, \dots, op_n are the operations performed when the transition fires.

Sequence Diagrams

In CHARMY, sequence diagrams are used to describe temporal properties we want to check on the SA. They are described using a UML notation, stereotyped so that i) each rectangular box represents an architectural component, ii) each arrow defines a communication line (a channel) between the left and the right components. Both synchronous and asynchronous communications are taken into account. Between a pair of messages we can select if other messages can occur (loose relation) or not (strict relation). Graphically, the strict relation is realized with a thick line that lies the messages pair. Constraints, graphically represented as a filled circle, are introduced to define a set of messages that must never occur in between the message containing the constraint (i.e., the one with the filled circle) and its predecessor. Messages are typed as *regular messages* (identified by the *e* label), *required messages* (identified by the *r* label) and *fail messages* (identified by the *f* label).

4.1 Integrating the CHARMY notation models into DUALLY

Since CHARMY makes use of its own stereotyped notation for state and sequence diagrams, a CHARMY profile needs to be created by extending the **DUALLY** profile. Informally, the CHARMY profile revises the concept of transition label for state machines and extends the concept of sequence diagram message, while adding new modeling notations for constraints. When identifying the semantic relationships among the CHARMY and the **DUALLY** profiles, we identify the following semantic links:

Rule 1. Each state diagram describes the behavior of a component or a connector.

Rule 2. The state machines transition labels identify communications between components.

Rule 3. Each lifeline in the scenario corresponds to a state machine.

Rule 4. Sequence diagrams can contain only messages already inserted into the state diagrams of the involved components.

Rule 5. The sender and the receiver of a message must be the same (components) in the sequence diagrams and in the state diagrams.

Thus, the CHARMY profile extends the **DUALLY** profile embedding inside the relations before itemized. Operatively, the process of introducing CHARMY is composed of the following steps: the CHARMY *Input filter* gets the

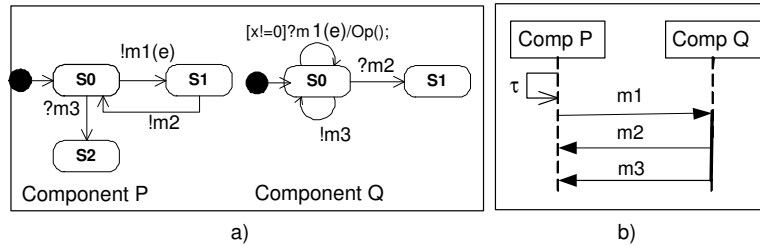


Figure 4. CHARMY State and Sequence formalisms and UML 2.0 State Diagram

CHARMY profile and (i) extracts the models and introduces them into the *Modeling extension* component; (ii) in according to the profile, an extension of the XML of **DUALLY** is performed.

The *Semantic Relations* component plays its role when **DUALLY** runs and a change performed by an analysis tool impacts on the **DUALLY** core. In this case an analysis feedback is forwarded to the other analysis tools. For example, if both CHARMY and a performance analysis profile/tool have been inserted into **DUALLY**, CHARMY can be run on the architectural model proving the SA conformance to selected properties. However, the performance tool may suggest to split a given component *C* into two components for improving the overall SA performance. What happens to the CHARMY results? **DUALLY** informs the CHARMY tool that component *C* has been split and two scenarios may apply: if CHARMY has been applied to a subsystem of the given SA, not including *C*, then previous results are still valid. Instead, if CHARMY has been applied to the entire SA, compositional verification techniques can be applied to the modified architectural model [10].

5 Conclusions and Future Work

DUALLY is an ongoing work which provides an extensible UML-based notation for SA modeling and analysis. **DUALLY** is composed by an UML profile for SA modeling, which allows to easily specify SA concepts slightly extending the UML 2.0 notation. **DUALLY** provides extensibility mechanisms and semantic rules which allow to introduce and integrate UML-based notations and tools.

In future work, we will validate this initial work by extending the **DUALLY** profile with our UML-based notations for SA-based model-checking and testing [29, 2]. While a plugin framework for SA-based analysis tools is already available [2], we have to implement the Input Filters and Semantic Relations components.

References

- [1] xArch. <http://www.isr.uci.edu/architecture/xarch/>.
- [2] CHARMY Project. Charmy Web Site. <http://www.di.univaq.it/charmly>, 2004.
- [3] Fujaba Project. <http://wwwcs.uni-paderborn.de/cs/fujaba/publications/index.html>, 2005. University of Paderborn, Software Engineering Group.
- [4] xADL 2.0 Architecture Description Language. <http://www.isr.uci.edu/projects/xarchuci/>, 2005.
- [5] xAcme. <http://www-2.cs.cmu.edu/~acme/pub/xAcme>, Last Modified: 2004. Carnegie Mellon University.
- [6] ADML: Architecture Description Markup Language. http://www.opengroup.org/architecture/adml/adml_home.htm, Last Modified: December 2002. Proposed by the Open Group.
- [7] Acme. <http://www-2.cs.cmu.edu/~acme/>, Since: 1998. Carnegie Mellon University.
- [8] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng. Improving System Dependability by Enforcing Architectural Intent. In *ICSE 2005 Work. on Architecting Depend. Systems*, 2005.
- [9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, second edition*. SEI Series in Software Eng. Addison-Wesley Professional, 2003.
- [10] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE 2004)*, Edimburgh, 2004.
- [11] R. Chatley, S. Eisenbach, J. Kramer, J. Magee, and S. Uchitel. Predictable dynamic plugin systems. In *FASE*, pages 129–143, 2004.
- [12] V. Cortellessa, A. D. Marco, P. Inverardi, H. Muccini, and P. Pelliccione. Using UML for SA-based Modeling and Analysis. In *Int. Workshop on Software Architecture Description & UML. Hosted at UML 2004*. October 11-15, 2004, Lisbon, Portugal.

- [13] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *ICSE '02: Proceedings of the 24th Int. Conf. on Software Eng.*, pages 266–276, New York, NY, USA, 2002. ACM Press.
- [14] H.-E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit*, chapter Ch. 7, Representing Architecture, pages 251–279. John Wiley and Sons, 2004.
- [15] D. Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In *Formal Methods for Software Architectures*, pages 1–24. Lecture Note in Computer Science, 2804, 2003.
- [16] D. Garlan and A. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In *Proc. of the Third Int. Conf. on the Unified Modeling Language*, 2000.
- [17] D. Garlan and M. Shaw. An Introduction to Software Architecture. In *Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering*, volume 2, pages 1–39. World Scientific Publishing Company, 1994.
- [18] H. Gomaa and D. Wijesekera. The Role of UML, OCL and ADLs in Software Architecture. In *Proc. Of the Workshop on Describing Software Architecture with UML, in ICSE 2001*, Toronto, Canada, 2001.
- [19] M. Goulo and F. Abreu. Bridging the gap between Acme and UML for CBD. In *Specification and Verification of Component-Based Systems.*, 2003.
- [20] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 1998.
- [21] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Sept. 2003.
- [22] 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 University, Software Engineering Institute, 2004.
- [23] M. M. Kande', V. Crettaz, A. Strohmeier, and S. Sendall. Bridging the gap between IEEE 1471, Architecture Description Languages and UML. *Software and System Modeling*, 2:98–112, 2002.
- [24] M. M. Kande' and A. Strohmeier. Towards a UML profile for Software Architecture Descriptions. In Springer-Verlag, editor, *UML 2000*, York, UK, 2000.
- [25] P. Kruchten. Architectural Blueprints - The “4+1” View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.
- [26] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *First Working IFIP Conference on Software Architecture, WICSA1*, 1999.
- [27] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1), January 2002.
- [28] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.
- [29] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. *IEEE Trans. on Software Engineering*, 30(3):160–171, March 2003.
- [30] T. Pender. *UML Bible*, chapter Part V: Modeling the Application Architecture, page 940. Wiley Pub., 2003.
- [31] J. E. Perez-Martinez and A. Sierra-Alonso. UML 1.4 versus UML 2.0 as languages to describe Software Architectures. In *Proc. EWSA 2004. LNCS n. 3047*.
- [32] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. In *SIGSOFT Software Engineering Notes*, volume 17, pages 40–52, 1992.
- [33] M. Pinto, L. Fuentes, and J. M. Troya. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proc. of the 2 int. conf. on Generative programming and component engineering*, pages 118–137, 2003.
- [34] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proc. 20th Int. Conf. on Software Engineering*, 1998.
- [35] S. Roh, K. Kim, and T. Jeon. Architecture Modeling Language based on UML2.0. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, 2004.
- [36] B. Selic. On modeling architectural structures with uml. In *Proc. of the Workshop on Describing Software Architecture with UML, in ICSE 2001*, Toronto, Canada, 2001.
- [37] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, 1996.