Towards Weaving Software Architecture Models

Davide Di Ruscio,
Henry Muccini, Alfonso Pierantonio
Dipartimento di Informatica
Università degli Studi dell'Aquila
I–67100 L'Aquila, Italy
{diruscio,muccini,alfonso}@di.univaq.it

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

Abstract

Increasingly, UML metamodels and profiles are adopted to specify software architectures from different angles in order to cope with analysis specific modeling needs. In particular, whenever two or more analysis techniques are required over the same architectural model, different modeling tools and notations are required, and feedback from one model to the other is not propagated since integration is not universally established.

Model-Driven Architecture offers a conceptual framework for defining a set of standards in support of Model-Driven Development, where models are first class entities and play a central role in software development. In this paper, the coexistence and integration of different analysis techniques at the architectural level is reduced to the problem of enriching multi-view descriptions with proper UML elements by means of directed weaving operations.

1 Introduction

Software Architectures (SAs) serve today as useful high-level "blueprints" to guide the production of lower-level system designs and implementations, and later on for guidance in maintenance and reuse activities. A Software Architecture specification captures system structure, by identifying architectural components and connectors, and required system behavior, by specifying how components and connectors are intended to interact.

Over the last years, traditional formal architecture description languages (ADLs) have been progressively complemented and replaced by model-based specifications. The increased interest in designing dependable systems, meant as applications whose descriptions include non-quantitative terms of time-related aspects of quality, has favored the proliferation of analysis techniques each one based on a slightly different UML profiles or metamodels. As an immediate consequence, each profile or metamodel provides constructs that nicely support some specific analysis and leave other techniques unexplored. The resulting fragmentation induces the need to embrace different notations and tools to perform different analysis at the architecture level: for instance, supposing an organization (using UML notations) is interested in deadlock and performance analysis, a comprehensive result is obtained only using two different ADLs. Additionally, whenever the performance model needs to be modified, the deadlock model must be manually adjusted (based on the performance results) and re-analyzed, causing frequent misalignments among models.

Shifting the focus of software development from coding to modeling is one of the main achievements of Model-Driven Architecture [22] (MDA), which separates the application logic from the underlying platform technology and represents them with precise semantic models. Consequently, models are primary artifacts retained as first class entities and can be manipulated by means of automated model transformations.

In this paper, the coexistence and integration of different analysis techniques at the architectural level is reduced to the problem of enriching multi-view descriptions with proper UML elements through directed weaving operations (realized by means of model transformations). In particular, such integration is obtained by firstly setting a formal ground where models and metamodels are specified, then weaving operators are defined for the integration of the **DUALLY** [14] profile with the constructs needed for performing specific analysis. The weaving operators are mathematically specified through Abstract State Machines (ASMs) able to execute the integration according to the semantic of the used operator.

The structure of the paper is as follows: the next section sketches languages available for software architecture specification and provides the preliminaries for the definition of

DUALLY. Section 3 briefly reviews the Abstract State Machines which are used for the specification of model transformations, as illustrated in Section 4, and of the weaving operators presented in Section 5 together with the definition of the DUALLY profile. Section 6 describes a case study which illustrates the use of DUALLY and how it can be integrated following the proposed approach with constructs needed for performing fault tolerance analysis. Section 7 discusses some related work, while Section 8 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.

ADLs are formal languages for SA modeling and analysis. Although several studies have shown the suitability of such languages, they are difficult to be integrated in industrial life-cycles and only partially tool supported. The introduction of UML as a modeling language for software architectures (e.g., [17, 15] has strongly reduced this limitation. However, different UML-based notations are still needed for different analysis techniques, thus inducing the need to embrace different notations and tools to perform different analysis at the architecture level.

Section 2.1 surveys existing ADLs, while Section 2.2 describes the use of UML for modeling SAs. Section 2.3 discusses the problems of these two kind of languages, posing the basis for the solution that we propose for successfully model Software Architectures in practice.

2.1 ADL for Software Architecture modeling

Formal architecture description languages are well established and experienced, generally formal and sophisticated notations to specify software architectures. An (ideal) ADL has to consider support for components and connectors specification, and their overall interconnection, composition, abstraction, reusability, configuration, heterogeneity and analysis mechanisms [30].

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, such as *configuration management*, *distribution* and suitability for *product line architecture* modeling. Structural specifications have been integrated with behavioral ones with the introduction of many formalisms such as pre- and post-conditions, process algebras, statecharts, POSets, CSP, π-calculus and others [18].

Papers have been proposed to survey, classify and compare existing ADLs. In particular, Medvidovic and Taylor in [18] proposed a classification and comparison framework, describing what an ADL must explicitly model, and what and ADL can (optionally) model. A similar study has been performed for producing xArch [1], an XML schema to represent core architectural elements. ACME [2], the architecture interchange language, also identifies a set of core elements for architecture modeling, with components, connectors, ports, roles, properties and constraints.

Although several studies have shown the suitability of such formal languages for SA modeling and analysis, industries tend to prefer model-based notations.

2.2 UML for Software Architecture modeling

UML (with many extensions) has rapidly become a specification language for modeling software architectures. The basic idea is to represent, via UML diagrams, architectural concepts such as components, connectors, channels, and many others. However, since there is not a one-to-one mapping among architectural concepts and modeling elements in UML, UML profiles have been presented to extend the UML to become closer to architectural concepts.

Many proposals have been presented so far to adapt UML 1.x to model software architectures (e.g. [17, 15]). 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. A good analysis of UML1.x extensions to model SAs can be found in [17].

With the advent of UML 2.0, many new concepts have been added and modified to bridge the gap with ADLs. How to use UML 2.0 (as is) for SA modeling has been analyzed in some books. The UML 2.0 concepts of components, dependencies, collaborations and component and deployment diagrams are used. In [23], 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 [25, 15].

2.3 Modeling Software Architectures: a practical perspective

The introduction of UML-based notations for SA modeling and analysis has improved the diffusion of software architecture practices in industrial contexts. However, many

different UML-based notations have been proposed for SA modeling and analysis, with a proliferation of slightly different notations for different analysis. Supposing an industry making use of UML notations is interested in combining deadlock and performance analysis, a satisfactory result can be obtained only using two different notations: whenever the performance model needs to be modified, the deadlock model needs to be manually adjusted (based on the performance results) and re-analyzed. This causes a very high modeling cost, and creates a frequent misalignment among models.

The solution that we propose is a synergy between UML and ADLs proposing a new ADL, called **DUALLY**, which maintains the benefits of the ADLs formality and with the intuitive and fashioning notation of UML. **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 [18, 2, 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. **DUALLY** approaches 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 a 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*.

Going back to the problems stated at the beginning of this section, the definition of the **DUALLY** UML profile allows for an easier integration of software architecture modeling and analysis in industrial processes. However, different notations are still needed for different analysis techniques. To overcome this problem we outline an extendible framework that permits to add models and to extend existing ones in order to support the introduction of analysis techniques. Weaving operations will be introduced and used for the purpose of binding different elements of different models.

While sections 3 and 4 provide the formal ground and technique for model transformations, Section 5 proposes the **DUALLY** profile together with weaving models.

3 Abstract State Machines

Model Driven Architecture (MDA) [22] consists of a set of standards forming an implementation of the Model Driven Development approach even if the way for specifying and executing transformation of models is not yet universally established. The forthcoming QVT [21] language will provide a proper foundation for model transformations even if in parallel with the OMG process, a number of research groups have proposed their own approach giving their contribution in the discussion of a key operation for the success of MDA. Our experience has shown the validity of using Abstract State Machines (ASMs) [6] for the formal specification and execution of model transformations as it will be explained in the next section. In the rest of the section, we only briefly introduce ASMs here insisting on few introductory aspects.

ASMs bridge the gap between specification and computation by providing more versatile Turing-complete machines. The ability to simulate arbitrary algorithms on their natural levels of abstraction, without implementing them, makes ASMs appropriate for high-level system design and analysis. Additionally, ASMs are executable and several compilers and tools are available both from academy and industry. In the sequel of the paper, ASM rules are given in the XASM [3] dialect compiler. ASMs form a variant of first-order logic with equality, where the fundamental concept is that functions are defined over a set $\mathcal{U}$ and can be changed point-wise. The set $\mathcal{U}$, referred to as the *superuniverse* in ASM terminology, always contains the distinct elements *true*, *false*, and *undef*. Apart from these, $\mathcal{U}$ can contain numbers, strings, and possibly anything, depending on the application domain.

Being slightly more formal, we define the *state* λ of a system as a mapping from a signature Σ (which is a collection of function symbols) to actual functions. We write f_λ for denoting the function which interprets the symbol f in the state λ. Subsets of $\mathcal{U}$, called universes, are modeled by unary functions from $\mathcal{U}$ to *true*, *false*. Such a function returns *true* for all elements belonging to the universe, and *false* otherwise. A function f from a universe U to a universe V is a unary operation on the superuniverse such that for all $a \in U$, $f(a) \in V$ or $f(a) = undef$ otherwise. The universe *Boolean* consists of *true* and *false*.

A basic ASM *transition rule* is of the form

$$f(t_1, \ldots, t_n) := t_0$$

where $f(t_1, \ldots, t_n)$ and t_0 are closed terms (i.e. terms containing no free variables) in the signature Σ. The semantics of such a rule is this: evaluate all the terms in the given state, and update the function corresponding to f at the value of the tuple resulting out of evaluating $(t_1, \ldots, t_n)$ to the value obtained by evaluating t_0. Rules are composed in a parallel fashion, so the corresponding updates are all executed at once. Apart from the basic transition rule shown above, there also exist *conditional* rules where the firing depends on the evaluated boolean condition-term, *do-for-all* rules which allow the firing of the same rule for all the elements

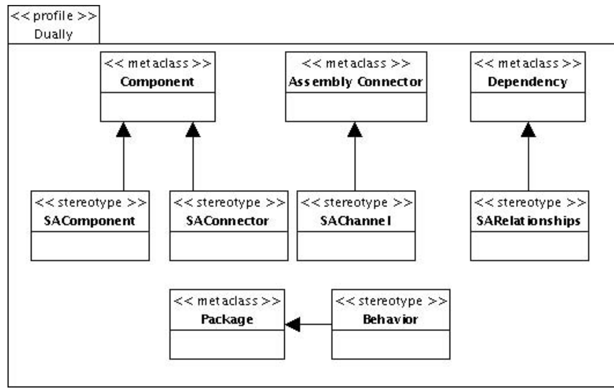

Figure 1. The DUALLY profile

of a universe, and lastly *extend* rules which are used for introducing new elements into a universe. Transition rules are recursively built up from these rules. Of course not all functions can be updated, for instance the basic arithmetic operations are typically not re-definable.

4 ASM based Model Transformations

Over the last years, a number of model to model transformation approaches have been proposed both from academia and industry even if a standardized way for specifying and executing transformations between models has not been reached yet. Different classifications [9, 20, 33] of model transformation approaches have been proposed and all of them recognize at least declarative and hybrid solutions. In the former category, graph transformation approaches (like AGG [32], PROGRESS [29], GreAT [31] and VIATRA [8]) play a key role as inspired by heavily theoretical work in graph transformations, while hybrid solutions offer declarative specifications and imperative implementations, like ATL [5] that wraps imperative bodies inside declarative statements.

Recently, our experience has proven the validity of Abstract State Machines (ASMs) as a formal and flexible platform on which to base a hybrid solution for model transformations: on one hand they combine declarative and procedural features to harness the intrinsic complexity of such task [27]; on the other hand, they are mathematically rigorous and represent a formal basis to analyze and verify that transformations are property preserving (as in [7]). Furthermore, we believe that ASMs could serve also for having better insights on transformation languages in general, on their intrinsic nature and for eventually having a basis for comparative analysis among such complex objects. Possibly, they could also prelude to a common semantic ground (kind of normal form of transformations and models) where evaluating properties of transformations [7] and weaving operations even written in different languages.

ASM based model transformations start from an algebra encoding the source model and return an algebra encoding the target one. The signature of an algebra encoding a model is canonically induced by the corresponding metamodel whose elements define the sorts of the signature. For instance, the meta-class and meta-association elements of MOF give place to the MetaClass and MetaAssociation sorts, i.e. the algebra has two universes containing distinguished representatives for all the meta-classes and meta-associations in the model. Additionally, the metamodels induce also functions which provide with support to model navigation, e.g. MetaAssociation elements have source and target functions

$$source, target : MetaAssociation \rightarrow MetaClass$$

which return the source and the target meta-class of a meta-association. For instance, in Figure 2 the algebraic encoding of the DUALLY profile, graphically depicted in Figure 1 and described in the next section, is given. Such canonical encoding, with some minor considerations, enables the formal representation of any model (conforming to a specified metamodel) which can be automatically obtained. Moreover, the encoding contains all the needed information to translate the final ASM algebra into the corresponding model.

5 Dually

This section introduces the DUALLY profile for SA modeling, by describing the provided constructs. Then, the weaving operators required to extend the core profile are given and specified via the Abstract State Machines formalism.

5.1 UML Profile

Goal of the DUALLY profile is extend UML 2.0 in order to model core architectural concepts: 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. 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.

The DUALLY profile is depicted in Figure 1 and defined in a ≪profile≫ stereotyped package. Within this package the classes of the UML metamodel that are extended by a

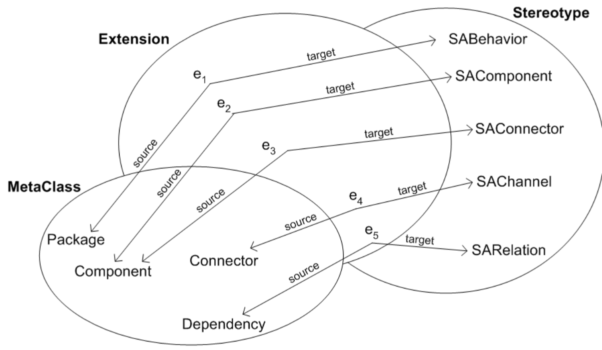

Figure 2. Algebraic encoding of DUALLY profile

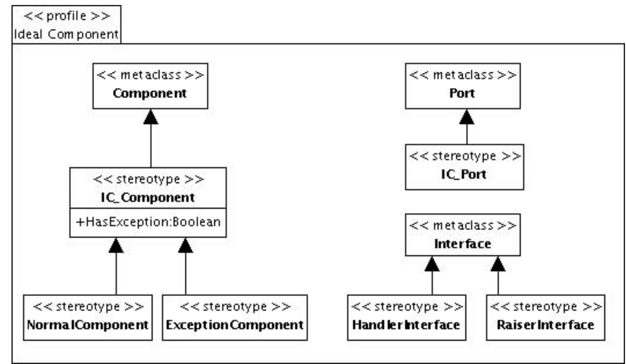

Figure 3. The Internal Component profile

stereotype are represented as a conventional class with the optional keyword *metaclass*. A stereotype is depicted as a class with the keyword *stereotype*. The *extension* relationship between a stereotype and a metaclass is depicted by an arrow with a solid black triangle pointing toward the metaclass. In particular, the new concepts provided the DUALLY profile are discussed in the following:

Architectural components: an SA component is mapped into UML components. "Structured classifiers" permit the natural representation of architecture hierarchy and ports provide a natural way to represent runtime points of interactions. As noticed in [25], SA components and UML components are not exactly the same, but we believe they represent a right compromise.

Relations among SA components: 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.

Connectors: 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. The DUALLY profile makes use of UML (stereotyped) components that, from the architectural point of view, seems the cleanest choice.

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 of architectural channel.

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.

Section 6 describes the use of the profile for the specification of a sample software architecture.

5.2 Weaving Models

The separation of concerns in software system modeling avoids the constructions of large and monolithic models which are difficult to handle, maintain and reuse. At the same time, having different models (each one describing a certain concern or domain) requires their integration into a final model representing the entire domain [24].

The weaving operation [4], typically exploited for database metadata integration and evolution, can be used for setting fine-grained relationships between models or metamodels and executing operations on them based on link semantics. Furthermore, the integration of models or metamodels can be performed by establishing correspondences among them by means of weaving associations specifically defined for the considered application domain. The description of such links consists of precise models conforming to appropriate weaving metamodels obtained by extending a generic one (inspired by [10]) with new constructs needed for the integration purposes.

In the sequel, weaving operators for extending the DUALLY profile are described and an example of their application is also provided. Such operators are inspired by [24] and they aim at extending the profile in a conservative way in the sense that deletions of constructs are denied, and only specializations or refinements of them are allowed. In particular,

- the *inherit* operator used for connecting an element of a UML profile with one of DUALLY is used in weaving models by means of ≪inherit≫ stereotyped associations as the one in Figure 4. The result of its application is the extension of DUALLY with a new stereotype (if it does not exist) having as base class the target element of the stereotyped association and the tags of

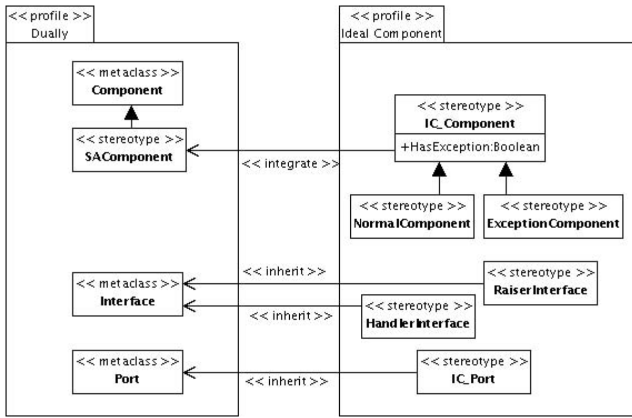

Figure 4. Weaving Models

the source one. The operator can be applied for extending the **D**UA**L**L**Y** elements and all the metaclasses of the UML metamodel.

– the *integrate* operator is used by means of ≪integrate≫ stereotyped associations as for example the one in Figure 4. The aim of such operator is to extend the available **D**UA**L**L**Y** constructs with the characteristics of elements belonging to other UML profiles. For example, the profile depicted in Figure 3 (described later in Section 6) and the one in Figure 1 both extend the standard metaclass *Connector*. The former provides an additional tag not provided in the latter. Connecting these two elements by means of an ≪integrate≫ stereotyped association will result in the addition of the tags belonging to the source element into the target one. In case of conflicts (e.g., tags with the same name but with different types) the elements of **D**UA**L**L**Y** are predominant. Furthermore, the extensions of the source elements are added to the target one.

The semantic and the execution of the discussed operators are defined by means of ASMs rules, like for model transformations as described in Section 4. This allows preserving the same formal ground for model specifications, their transformations and weaving operations among them as well.

The transformation phase which has to extend the **D**UA**L**L**Y** profile starts from an algebra whose signature includes the following universes and functions which are the union of the signatures derived from the metamodels of the involved source models, i.e. the profile specifications and the weaving model respectively.

```
1   universes MetaClass, Stereotype, Extension, Tag
2   universes Inherit, Integrate
3   ...
4   function name(_) → String
5   function source(_) → _
6   function target(_) → _
7   function belong(Tag) → Stereotype
8   function type(Tag)} → DataType
9   function dually() → Bool
10  function icProfile() → Bool
11  ...
```

Some auxiliary functions are used, in particular the function *dually()* and *icProfile()* are defined in order to establish whether, given an element, it belongs to the algebra encoding the **D**UA**L**L**Y** profile or the Ideal Component one. Moreover, the functions *belong()* and *type()*, given an element of the set Tag, return the stereotype to whom it belongs and its data type, respectively.

The weaving operation mainly consists of two ASM rules each devoted to the management of the previously described weaving operators. Specifically, the *Inherit* rule for each element contained in the set Inherit of the algebra encoding the weaving model, extends the algebra encoding the **D**UA**L**L**Y** profile. The updating of the algebra consists of the addition of new stereotypes (see line 3 below) which can have as base class a UML metaclass (see line 8 below), as for the associations depicted in Figure 4 where the *Interface* and *Port* metaclasses are involved, or an existent **D**UA**L**L**Y** stereotype (see line 12 below).

```
1  asm Inherit is
2   do forall x in Inherit
3      extend Stereotype with s
4        name(s):=name(source(x))
5        ...
6        dually(s):=true
7        extend Extension with e
8          choose c in MetaClass : name(c)=name(target(x))
9              source(e):=s
10             target(e):=c
11         endchoose
12         choose c in Stereotype : name(c)=name(target(x))
13             source(e):=s
14             target(e):=c
15         endchoose
16         dually(e):=true
17       endextend
18       propagateExtension(s,source(x))
19     endextend
20  enddo
21 endasm
```

The auxiliary submachine *propagateExtension*(s_1,s_2) recursively updates the sets Extension and Stereotype of the algebra encoding **D**UA**L**L**Y**, in order to extend the stereotype s_1 with the extensions (if available) of the stereotype s_2.

The *Integrate* rule aims at weaving the source element of the ≪Integrate≫ stereotyped associations with the target one. Firstly, all tags of the source stereotype are added to the target one if there are not conflicts (see line 6 of the rule) and in case of overlapping, the elements of **D**UA**L**L**Y** are predominant. In line 15 of the rule the submachine *propagateExtension*(s_1,s_2) is called. For instance, the application of the *Integrate* rule, taking into account the weaving model of Figure 4,

will modify the DUALLY stereotype ≪SAComponent≫ by adding the tag *HasException* and the stereotypes ≪NormalComponent≫ and ≪ExternalComponent≫ as extensions of ≪SAComponent≫.

```
1  asm Integrate is
2   do forall i in Integrate
3      do forall t1 in Tag
4         if ( icProfile(t1) and belong(t1)=source(i) )
5           then
6             if not (exists t2 in Tag: dually(t2) and
7                     name(t2)=name(t1))
8               then
9                  extend Tag with t3
10                        name(t3):=name(t1)
11                        type(t3):=type(t1)
12                        belong(t3):=target(i)
13                        dually(t3):=true
14                 endextend
15                 propagateExtension(target(i),source(i))
16            endif
17        endif
18     enddo
19  enddo
20 endasm
```

Once the weaving operation is performed, the obtained extended algebra contains all the information required to translate it into the corresponding model. The next section describes a case study showing firstly the use of DUALLY for describing a software architecture, then the extended version of the profile, obtained by means of the previously described weaving operation according to Figure 4, is used to design the same system with other constructs needed for performing some fault-tolerant analysis.

6 Using Dually for Designing Fault-tolerant systems

In this section we show how DUALLY can be extended in order to integrate SA-based concepts with fault tolerance information. We make use of the mining control system case study [26], a simplified system for the mining environment. The mineral extraction from a mine produces water and releases methane gas on the air. These activities must be monitored. Figure 5 shows the SA for the control system modeled by using the basic features of DUALLY. It is composed of two components, the *Operator Interface* component, which represents the operator user interface, and the *Control Station*, which is divided in three subcomponent: *Pump Control*, *Air Extractor Control*, and *Mineral Extractor Control*. *Pump Control* is responsible of monitoring the water level, *Air Extractor Control*, switching on and off the subcomponent *Air Extractor*, controls the methane level, and finally the mineral extraction is monitored by *Mineral Extractor Control*.

However, the possible responses of a component when implemented and operating are normal and exceptional. While normal responses are those situations where components provide normal services, exceptional responses correspond to errors detected into a component. Typically, exceptional responses are called exceptions [12]. Therefore, it is natural to design not only the normal behavior, but also the exceptional one. Similarly to the normal behavior, exceptional behaviors can be elicited from requirements and modelled. In order to successfully model fault tolerant systems, the basic features offered by DUALLY are not enough.

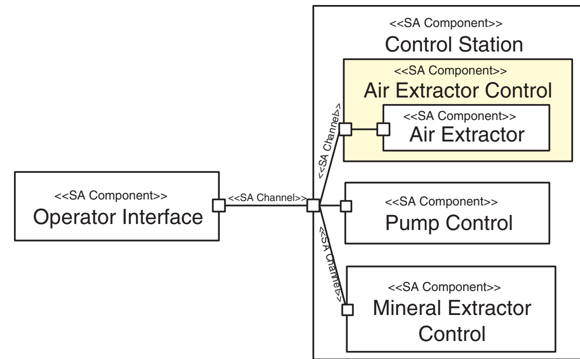

Figure 5. The mining control system SA

Ideally components are composed of two different parts: normal and exceptional activities [16, 26]. The normal part implements the component's normal services and the exceptional part implements the responses of the component to exceptional situations, by means of exception-handling techniques. When the normal behavior of a component signals an exception, called *internal exception*, its exception handling part is automatically invoked. If the exception is successfully handled the component resumes its normal behavior, otherwise an *external exception* is signaled. *External exceptions* are signaled to the enclosing context when the component realizes that is not able to provide the service.

Figure 3 shows the profile for the idealized components. The *SA component* is specialized in the stereotype ≪IC component≫ that contains the boolean tag *HasException* that is true if the component have a description of the fault tolerant behaviour, false otherwise. *IC Component* is even specialized with the stereotypes ≪NormalComponent≫ and ≪ExceptionalComponent≫ describing the normal and the exceptional behavior respectively. Ports are specialized by the stereotype ≪IC Ports≫ in order to model communication ports for signaled exceptions. Finally interfaces are used for the exceptions propagation from the normal to the exceptional part specialized with the stereotypes ≪HandlerInterface≫ and ≪SignalInterface≫ representing the handler and the signaler respectively.

Figure 6 shows the result of the weaving, obtained applying the mega operators, inherit and integrate, between the

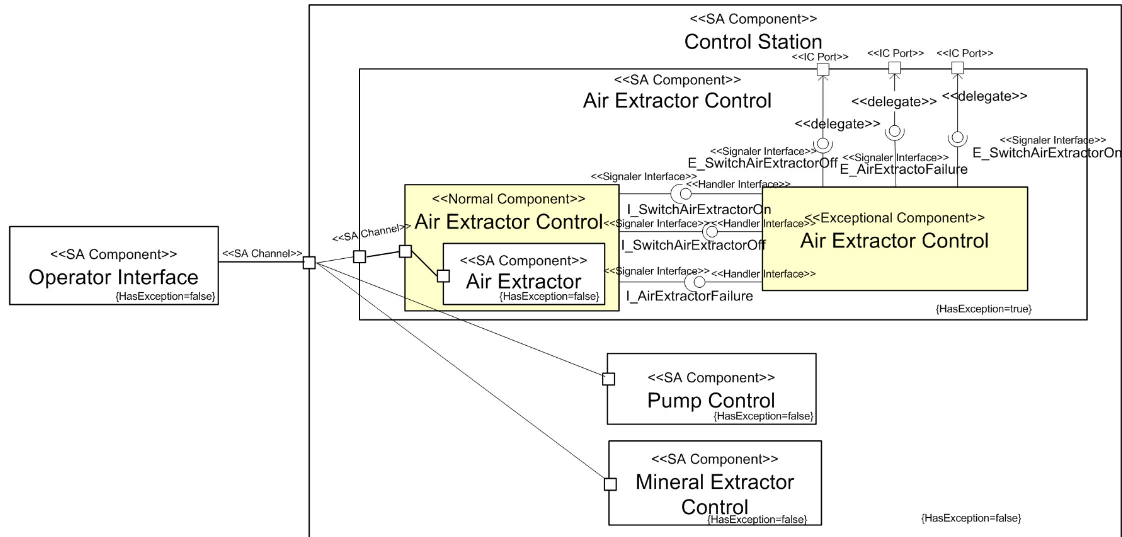

Figure 6. Air Extractor Control component with fault-tolerance information

DUALLY profile, explained in Figure 1, and the idealized component profile shown in Figure 3. As show in Figure 4 the *IC Component* is "integrated" with the SA component of DUALLY. Consequently the components in Figure 6 are SA components extended with the tag *HasException* and they can be specialized in *NormalComponent* and *ExceptionalComponent*, as happens for the *Air Extractor Control* component.

On the contrary the *inherit* operator is used for interfaces and ports. In fact for exceptions propagation we want to use ports and interfaces of the IC profile while for the classical communication between components we want to use elements of the DUALLY profile.

More in details, the exceptions *I_SwitchAirExtractorOff*, *I_AirExtractorFailure*, and *I_SwitchAirExtractorOn* are internal exceptions signaled by the normal part (Signal Interfaces). These exceptions are catched by the exceptional part (Handler Interfaces), which signals external exceptions in the case of the exceptional component realizes that is not able to provide the service (Signaler Interfaces and IC Ports).

The subcomponent *Air Extractor* does not have exceptional behavior and then is modelled as an extended DUALLY component, contained into the normal part of the *Air Extractor Control* component.

7 Related Work

The concept of weaving appears in a number of approaches for model management with the objective of handling fine-grained relationships among elements of distinct metamodels. Typical applications of model weaving are database metadata integration and evolution, as in [19] where the authors propose Rondo, a generic metamodel management algebra which proposes a number of algebraic operators to manage mappings and models. The techniques presented here have a more general purpose flavor since ASMs constitute a formalism for defining algebraic (non-homomorphic) transformations, i.e. macro operations over the algebras.

In [13] a UML extension is introduced to express mappings between models using diagrams, and illustrates how the extension can be used in metamodeling. The extension is inspired by mathematical relations and it is not really conceived for extending or refining UML profiles as the approach proposed in this paper.

Another generic metamodel to support weaving operation is given in [11]. The approach is based on the possible extensibility and variability of mappings among metamodels and it is supported by a prototypical implementation. In [24] a number of operators for model integration are described and they have partially inspired the weaving constructs proposed in this paper.

Finally, the idea of using formal languages for performing model transformations and weaving operations, is slightly related to [28] which proposes a formal approach based on a relational calculus for checking and transforming models of complex systems. Even if these two approaches share a formal flavor, transformations are used for different purposes. While in [28] formal model transformations are

addressed to support a development process from abstract models down to the final software product, in our proposal we apply model transformations at a different layer of abstraction in order to weave a minimal core language with domain-specific concepts for software architecture analysis.

8 Conclusions and Future Work

There is not best or unique language for software architecture specification, since it depends on which aspects of architectural design we want to represent and then analyze. At any time a new slightly different analysis is required, new modeling concepts are needed. This consideration justifies and motivates the need of different profiles and metamodels for specifying software architectures. Unfortunately, the proliferation of such profiles is leading to a fragmentation of tools and techniques for SA specification and analysis, then limiting the adoption of such technologies in industrial contexts.

This paper has discussed the use of (ASM based) model transformation techniques, where *weaving* operators are introduced for extending the **D**U**AL**LY profile. After their definition, we have shown how weaving operators can be applied to extend the **D**U**AL**LY profile with fault tolerance information. It is important to note that the weaving operators here defined are independent from the specific target domain (i.e., fault tolerance modeling) and can be applied to other domains. Further work will contribute to their refinement and generalization to different modeling domains.

The integration among software architecture modeling concepts and fault tolerance information is the first step in our wish list. Our long term goal envisions the integration among different modeling profiles for software architecture modeling, proposed so far for functional and non functional analysis. A plugin-based framework for assembling different analysis tools and for automated model-transformation is also in our desires.

Acknowledgments

This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04.

References

[1] xArch. http://www.isr.uci.edu/architecture/xarch/.

[2] Acme. http://www-2.cs.cmu.edu/~acme/, Since: 1998. Carnegie Mellon University.

[3] M. Anlauff. XASM – An Extensible, Component-Based Abstract State Machines Language. In *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer, 2000.

[4] J. Bézivin. On the Unification Power of Models. *J. Software and Systems Modeling*, 4(2):171–188, 2005.

[5] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. Rougui. First Experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA W. Generative Techniques in the context of MDA*, 2003.

[6] E. Börger. High Level System Design and Analysis using Abstract State Machines. In *FM-Trends 98, Current Trends in Applied Formal Methods*, volume 1641 of *LNCS*, pages 1–43. Springer, 1999.

[7] M. Caporuscio, D. D. Ruscio, P. Inverardi, P. Pelliccione, and A. Pierantonio. Engineering MDA into Compositional Reasoning for Analyzing Middleware-Based Applications. In *EWSA '05*, volume 3527 of *LNCS*, pages 475–490. Springer, 2005.

[8] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA - visual automated transformations for formal verification and validation of UML models. In *ASE*, pages 267–270, 2002.

[9] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, 2003.

[10] M. D. Fabro, J.Bezivin, F. Jouault, E. Breton, and G.Gueltas. AMW: A generic Model Weaver. In *Int. Conf. on Software Engineering Research and Practice (SERP05)*, 2005.

[11] M. D. D. Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a Generic Model Weaver. In *IDM05*, 2005.

[12] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[13] J. H. Hausmann and S. Kent. Visualizing model mappings in UML. In *SoftVis '03: ACM symposium on Software visualization*, pages 169–178. ACM Press, 2003.

[14] P. Inverardi, H. Muccini, and P. Pelliccione. **D**U**AL**LY: Putting in Synergy UML 2.0 and ADLs. In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*. Pittsburgh, PA, 6-9 November 2005.

[15] 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.

[16] P. Lee and T. Anderson. Fault Tolerance: Principles and Practice, Second Edition. *Prentice-Hall*, 1990.

[17] 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.

[18] 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.

[19] S. Melnik, E. Rahm, and P. Bernstein. Rondo: a programming platform for generic model management. In *Int. Conf. on Management of Data*, pages 193–204. ACM Press, 2003.

[20] T. Mens and P. V. Gorp. A taxonomy of model transformation. In *Int. Workshop on Graph and Model Transformation (GraMoT)*, 2005.

[21] Object Management Group. MOF 2.0 Query/View/Transformation RFP, 2002. OMG document ad/02-04-10.

[22] Object Management Group (OMG). OMG/Model Driven Architecture - A Technical Perspective, 2001. OMG Document: ormsc/01-07-01.

[23] T. Pender. *UML Bible*, chapter Part V: Modeling the Application Architecture, page 940. Wiley Pub., 2003.

[24] T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Model Integration Through Mega Operations. 2005. accepted for publication at the Workshop on Model-driven Web Engineering (MDWE2005).

[25] S. Roh, K. Kim, and T. Jeon. Architecture Modeling Language based on UML2.0. In *Proocedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, 2004.

[26] C. M. F. Rubira, R. de Lemos, G. R. M. Ferreira, and F. C. Filho. Exception handling in the development of dependable component-based systems. *Softw. Pract. Exper.*, 35(3):195–236, 2005.

[27] D. D. Ruscio and A. Pierantonio. Model Transformations in the Development of Data–Intensive Web Applications. In *CAISE '05*, volume 3520 of *LNCS*, pages 475–490. Springer, 2005.

[28] B. Schätz, P. Braun, F. Huber, and A. K. Wißpeintner. Checking and transforming models with AutoFOCUS. In *12th IEEE Int. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 307–314, 2005.

[29] A. Schürr, A. Winter, and A. Zündorf. *Handbook on Graph Grammars and Graph Transformation*. World Scientific, 1999.

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

[31] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, and G. Karsai. Domain model translation using graph transformations. In *10th IEEE Int. Conf. and Workshop on the Engineering of Computer-Based Systems*, pages 159–168, 2003.

[32] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *AGTIVE 2003*, volume 3062, pages 446–453. LNCS, 2004.

[33] L. Tratt. Model transformations and tool integration. *J. of Software and Systems Modelling*, 4(2):112–122, May 2005.