Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach

Elena Navarro[1], and Carlos E. Cuesta[2]

[1]Department of Computing Systems, University of Castilla-La Mancha,
Campus Universitario s/n, 020071, Albacete, Spain
enavarro@dsi.uclm.es
[2]Dept. Computing Languages and Systems II, Rey Juan Carlos University,
C/ Tulipán s/n. 28933 Móstoles, Madrid, Spain
carlos.cuesta@urjc.es

Abstract. The impact of architecture is not only significant in the final structure of software, but also in the development process. Architecture itself is assembled by a network of design decisions (DD) composing a design rationale. Such rationale has often been neglected; however it is essential to deal with future change. This is also the role of traceability, the crosscutting relationship describing the evolution of software. The methodology ATRIUM provides the method to manage traceability, by using a Model-Driven Development (MDD) approach where every model element maintains links to related elements in previous and further stages. This proposal defines how these links have been exploited to support the tracing of DDs and their accompanying design rationales (DRs), and study their propagation. We also present how ATRIUM tools support this proposal by introducing DD/DRs and their traceability links from requirements to the target architectural model. These are automatically generated by M2M transformations, avoiding the error-prone task of managing them by hand.

Keywords: Design Decision, Design Rationale, Model-Driven Development, Model-To-Model Transformation, CASE Tool, Traceability.

1 Introduction

Almost two decades have already passed since Perry and Wolf wrote their seminal paper [27] on the foundations of Software Architecture, an event which is considered as the beginning of the modern age of Software Architecture. Already in this pioneer paper, Perry and Wolf defined architecture as a model composed of *elements*, *form*, and *rationale*. The first item refers to the description of components and what would be later defined as connectors; the second item refers to constraints in their properties and relationships. Both of them have been integrated into practice long ago; but the third one has often been neglected. Rationale was defined as the *motivation* for the choice of style, form or elements, which explains why this choice satisfies the system requirements; but it has been scarcely considered until recently.

Just four years ago, a number of researchers including Bosch [1] highlighted this fact, and stressed the importance of overcoming this drawback. They noticed that many architectural designs still lack such an explanation, and that this limitation is a factor hindering further spreading and improvement in the area. They advocate for the insertion of additional first-class assets in architecture description, which explicitly document *design decisions* (DDs) being made.

This emphasis on the management of architectural knowledge has quickly achieved a great popularity, not only within the specific field but also outside it [10]. Much of the research in this topic has focused on the *internal* structure of design assets. For instance, for every asset we can at least separate the *decision* itself (DD) from the *rationale* (DR) for this decision. But their external (compositional) structure is even more interesting. The composition of the set of individual DD/DRs builds up the system's global rationale; and the structure of this *architectural rationale* is not only supported by the final architecture, but it also mimics the dynamic structure of the development *process* itself. The perspective is much richer when it is considered; then a *decision* is a choice, and this means a potential turn during the process. Every decision has the potential to be a *variation point* (VP) for the architecture. This also means that traceability relationships must also be considered.

However complexity grows exponentially in this case, and then automatic support becomes a necessity. Of course, if we are considering automation of the development process, and traceability relationships in particular, there is an obvious connection to MDA [28]. Model-driven tools (should) consider traceability as a basic relationship; therefore they already provide the basis to exploit decisions as described.

The proposal outlined in this article starts from this idea *–supporting architectural knowledge with a model-driven process–* and in doing so, it gathers every feature in research mentioned so far. Indeed, traceability will be the "spine" for the architectural rationale, which will reflect the structure of the process. This building process will be supported by (semi-)automatic model-driven tools, and by applying MDD techniques, one of the most important trends in current software engineering will be used.

Therefore, in this article a specific extension of ATRIUM (Architecture Traced from RequIrements applying a Unified Methodology) [22], a methodology which uses a model-driven approach to generate an architecture definition from the requirements, is presented. The extended version provides an explicit support to describe decisions, and to trace them back to requirements. This is also implemented in its specific tool, MORPHEUS [19], which has been also extended to support the management of additional information and its inclusion within the relevant M2M transformations. The result is explained by means of a concrete, real-world case study, and the support for automation provided by MORPHEUS is described. Finally this is related to the state-of-the-art in architectural knowledge proposals, and some consequences are extracted from this comparison, such as the consistency of the whole approach.

2 ATRIUM in a Nutshell

There are many compelling reasons about why it is necessary to include traceability throughout the process of software development process. Among them, the most

accepted one is that traceability makes available the ability to deal properly with the change as it appears, evaluating its impact, determining the affected elements, etc. Therefore, the exploitation of the traceability from the requirements stage towards the architectural specification (and code) arises as one of the necessary cornerstones to achieve the success of any software development process. In this context is where the methodology ATRIUM (Architecture Traced from RequIrements applying a Unified Methodology) [22] provides support. It is a methodology designed for the concurrent definition of Requirements and Software Architecture, defining the automatic/semi-automatic support for traceability throughout its application.

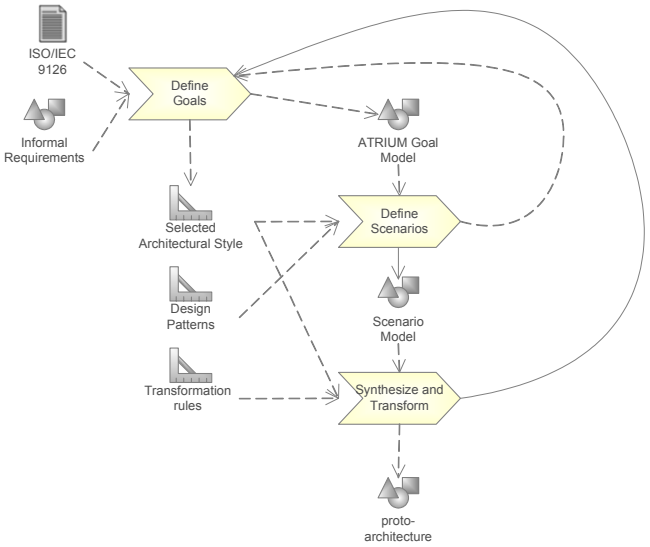

Fig. 1. An outline of ATRIUM

ATRIUM has been described following a MDD approach [28]. Fig. 1 shows its three main activities (described using SPEM [29]) that must be iterated over in order to define and refine the different Models and allow the analyst to reason about both the requirements and the architecture. These activities are described as follows:

- *Define Goals*. This activity allows the analyst to identify and specify the requirements of the system-to-be by using the *ATRIUM Goal Model* [23] (Fig. 4 describes an example). This model is based on KAOS [4] and the NFR Framework [2] proposals. During the specification phase, along with an informal description of the requirements stated by the stakeholders, the ISO/IEC 9126 quality model [11] is used as an instantiable framework in order to provide the analyst with an initial set of concerns of the system-to-be.
- *Define Scenarios*. This activity focuses on the specification of the *ATRIUM Scenario Model,* that is, the set of *Architectural Scenarios* that describes the system's behaviour under certain *operationalization* decisions. A process for its description [21] has been established, to facilitate the automatic analysis of alternatives. Therefore, each Architectural Scenario depicts the architectural and environmental elements that interact to satisfy specific requirements and their level

of responsibility for achieving a given goal. It is worth noting that a profile has been defined to describe solutions, applying both to functional and non-functional requirements [22].

- *Synthesize and Transform*. This activity has been defined to generate the proto-architecture of the specific system. With this purpose, it synthesizes the architectural elements from the ATRIUM Scenario Model, building up the system along with its structure. This proto-architecture is used as a first draft of the final description of the system that can be refined in a later stage of the software development process. This activity has been defined by applying *Model-To-Model Transformation* techniques (M2M, [3]), specifically, QVT Relations [24]. The advantages are twofold. First, the Architectural Style selected during the *Define Goal* activity can be automatically applied so that the constraints imposed by the Style are satisfied. Second, the analyst can generate the proto-architecture he/she deems appropriate for his/her purposes.

It must be pointed out that ATRIUM is independent of the Architectural Metamodel used to describe the proto-architecture. The *Synthesize and Transform* stage has been defined using M2M techniques whereby the analyst only has to describe the needed transformations to instantiate the needed Architectural Metamodel. Currently, the set of transformations [22] to generate the proto-architecture instantiating the PRISMA Architectural Model [25] has been defined because a compiler to generate code from PRISMA Models already exists.

3 MDD for Tackling Design Decisions and Rationales

As stated above, ATRIUM has been defined following the MDD approach, so that every stage of the software development process is described by establishing clearly its associated Metamodels along with forward and backwards traceability links between them. To tackle the description of DDs and DRs, our proposal exploits these links throughout the application of ATRIUM by means of both their incorporation in each Metamodel and the establishment of mechanisms for their propagation through abstraction levels, as it will be presented in the following.

Fig. 2 shows how DDs and DRs are introduced from the very beginning of the software development process by means of its introduction at the requirement stage. *Operationalization* is one of the main concepts used to describe the *ATRIUM Goal Model*. An *Operationalization* is a description of an architectural solution, i.e., an architectural *design choice* for the system-to-be to meet the users' needs and expectations. They are called *Operationalizations* because they describe the system behaviour to meet the requirements, both functional and non-functional. For this reason, two key attributes are included while they are described: *designDecision* and *designRationale*. The former is in charge of describing the architectural design solution and the latter describes why this decision has been made. The *Operationalizations* are related to *Requirements* by means of a relationship called *Contribution* whose main aim is to specify how an *Operationalization* contributes to/prevents the satisfaction of a *Requirement* facilitating the automatic analysis of architectural alternatives [22].

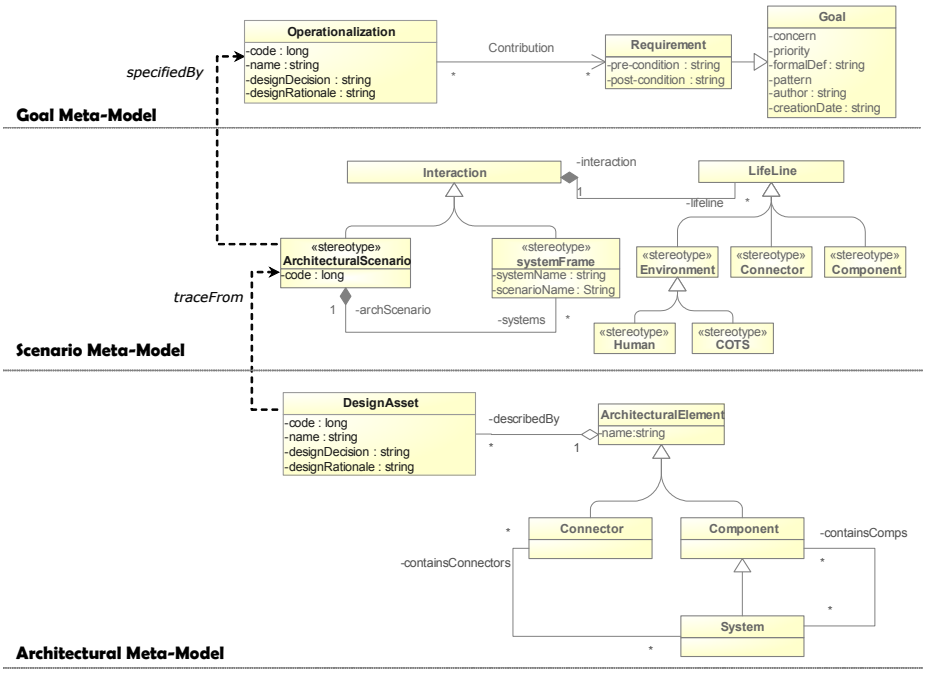

Fig. 2. Traceability Links between the main meta-elements of the involved Metamodels

Fig. 2 shows that *ArchitecturalScenario* constitutes one of the main concepts of the ATRIUM Scenario Metamodel. The *ArchitecturalScenario* is used to describe the system behaviour associated to one or several requirements and under a certain operationalization decision. Unlike proposals about classic scenarios, *Architectural Scenarios* specify interaction between architectural elements together with the environmental elements which play a role in that scenario. As depicted in Fig. 2, in ATRIUM *Architectural Scenarios* are traced from *Operationalizations* by means of a relationship known as *specifiedBy,* thereby stating clearly how (and why) every Architectural Scenario is specified. This relationship is specified by the analyst whenever a new *Architectural Scenario* is described, as it only emerges in the context of a specific *Operationalization*. For this reason, the relationship can be easily maintained thanks to the capabilities provided by MORPHEUS (see section 5).

Finally, at the Architectural Metamodel level, both DDs and DRs should be considered, to properly evaluate the impact of any change on the Architectural Specification. Therefore, the PRISMA Architectural Metamodel has been modified to introduce both of them at this level of abstraction. Fig. 2 shows a partial view of the PRISMA Metamodel, which has been extended to include a new element, the *DesignAsset,* in order to describe the DD and DR associated to each *ArchitecturalElement*. However, as stated above, this approach can be applied to any Architectural Metamodel. Another advantage of the proposal is that the *DesignAsset* could be linked to any other architectural element, when considered necessary, just by establishing properly the relationships inside the Metamodel and applying the process described in the following.

Fig. 2 also shows that *DesignAsset* has a relationship *traceFrom*, used to determine which ArchitecturalScenario originated the description of the Architectural Element. However, the task of specifying the relationship *traceFrom* could be cumbersome and error-prone if it was performed by hand. This has motivated the development of a proposal based on the use of M2M transformations, facilitating that both the *traceFrom* relationship and the *DesignAsset* itself can be automatically generated in the target Architectural Model, as described in the following section.

3.1 Model-To-Model Transformations to Deal with Traceability Links

As indicated in Fig. 1, the *Synthesize and Transform* activity in ATRIUM is in charge of the generation of the proto-architecture. Considering that the Goal Model and the Scenario Model must be transformed into the target Architectural Model, the use of M2M transformation techniques emerges as the most adequate approach to describe a solution for this activity. Several existing languages, such as QVT [24] or GReAT [31], have been proposed as solutions to define M2M transformations that increase the productivity, capture traceability relationships between models, improve maintainability by consistently describing the traceability throughout the lifecycle, etc. These languages were analyzed in [22], where their suitability for ATRIUM was studied by considering their support for several required features, such as the capability to *incrementally* update the target Architecture in response to the evolution in the Scenario and the Goal Model, or the support for automatic *tracing* between the source and target models. This analysis led us to select *QVT Relations* as the most adequate language to describe the required transformations.

To describe how our transformation works, we have to introduce briefly the way in which QVT Relations operates. In this language, a transformation is defined between *candidate models* and specified as a set of *relations*. A candidate model is any model that conforms to some Metamodel referenced in the transformation declaration. Every *relation* describes the constraints to be satisfied by the elements of these candidate models; all relations must hold in order to successfully apply the transformation.

Table 1. Declaring the Transformation to Generate the Architectural Model

```
transformation ScenariosToArchModel(goals: GoalMetamodel,
   scenarios: ScenarioMetamodel, archModel:ArchitecturalMetamodel)
```

Consider, for instance, the transformation in Table 1, which generates the target Architectural Model from two inputs: the Goal Model and the Scenario Model. This transformation has three candidate models, namely: *goals*, a candidate model conforming to the ATRIUM Goal Metamodel; *scenarios*, a candidate model conforming to the ATRIUM Scenario Metamodel; and finally *archModel*, a candidate model which must conform to the target Architectural Metamodel (in our case, the chosen metamodel is that of PRISMA). Specifically, we are going to focus here on how *Operationalizations* and *ArchitecturalScenarios* are used in the generation of both *DesignAssets* and the *traceFrom* relationship in the target Model [1].

[1] Interest reader is referred to [22] to obtain the whole description of the transformation.

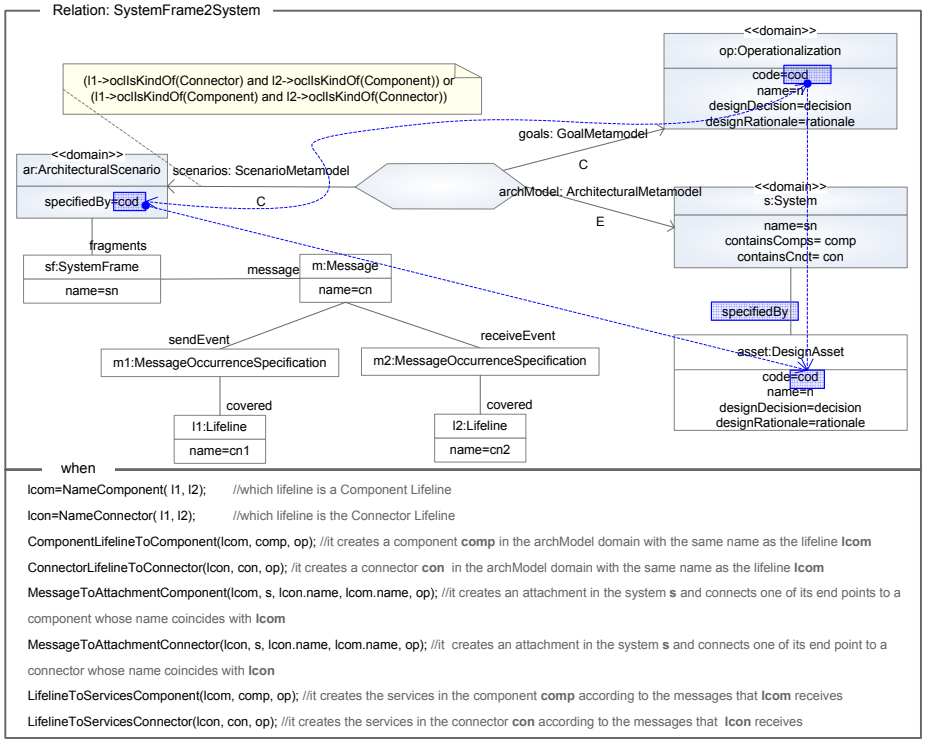

Fig. 3. Describing a Relation to Propagate the Design Decisions and Rationales

The structure of a *relation* is better exposed with an example. Consider the one in Fig. 3, *SystemFrame2System*, one of the relations which compose the transformation described in Table 1. This one was defined to generate "Systems" in the target Architectural Model. Every relation is defined by two or more domains (three in this case), identified by typed variables which must match some of the Metamodels in the transformation declaration. In Fig. 3, these are the same which were defined in the example in Table 1, namely: *goals*, *scenarios*, and *archModel*.

The relation imposes a pattern on every domain, describing the constraints to be satisfied by the elements of the involved model. When the elements contained in each candidate model *simultaneously* fulfill their corresponding patterns, then the matching happens and the relation is held.

In the example in Fig. 3, three patterns are described, using QVT graphical syntax: one for every domain (*goals*, *scenarios*, *archModel*). For instance, in the *goals* domain the relation establishes that every Operationalization (i.e. all elements of type *Operationalization*) must be retrieved to be used. But the pattern also imposes the condition which defines that the *code* of these Operationalizations gets bound to the variable *cod*. Simultaneously, in the *scenarios* domain, every Architectural Scenario (i.e. all elements of type *ArchitecturalScenario*) has a variable *specifiedBy* which is bound to the same variable *cod*. Then, the dotted line in Fig. 3 highlights the matching which might happen between both domains; this means that only Operationalizations

and Architectural Scenarios having the same value in these attributes will be used when the relation (or its container transformation) is applied. A similar pattern is also defined for the *archModel* domain, as can be seen in the Figure. In this case, not only the attribute *code* (which is again bound to the variable *cod*), but the rest of the attributes is also bound to those in the Operationalization.

The lower half of Fig. 3 contains the *when* clause, which describes a condition that must be held before the relation can be successfully applied. Within this example, the *when* clause contains the code required to invoke some additional relations.

In QVT Relations, the transformation can be defined either to *check* the models for consistency or to *enforce* the consistency by modifying one of the models, selected as target. Therefore, every pattern can be evaluated using two different modes: *checkonly* (marked with a *C* in Fig. 3), that just checks if the pattern is satisfied, reporting an inconsistency otherwise; and *enforce* (marked with an *E* in Fig. 3) which first checks whether the pattern is satisfied, and then creates, modifies or erases elements in the target model, as it is necessary to ensure consistency.

In the example in Fig. 3, we can observe that domains *goals* and *scenarios* are marked as *checkonly* but, on the contrary, *archModel* is marked as *enforce*. This means that when *archModel* is the target model, the proto-architecture is generated. The execution of the transformation checks whether there are elements in the target model that satisfy the relations, that is, the patterns described for its domain. If that was not the case, elements in the target model will be created, deleted or modified to enforce the consistency. This allows the analyst either to generate the proto-architecture, or to check whether inconsistencies emerge between the generated proto-architecture, the Goal Model, and the Scenario Model. Therefore, information about our DDs and DRs is automatically registered for Architectural Elements as the target Architectural Model is generated. For instance, in our proposal, every *System* in the target Architectural Model will be related to its corresponding *DesignAsset*, because the relationship *specifiedBy* between them will be automatically generated, as it has been defined in the relation described in Fig. 3.

Another advantage of using QVT Relations is that the language itself automatically generates a *Trace Class* for every relation, facilitating the registration of mappings between the elements in the involved Models, for instance those between elements in the proto-architecture and their corresponding elements in the scenarios and goals models. This means that the already described *traceFrom* relationship (see Fig.2) is automatically generated when the transformation is applied. This makes possible to easily maintain the traceability, both forward and backwards. Therefore, if some DD changes at the requirements stage, the set of architectural elements which will be affected can be determined automatically, and therefore the Architectural Model can easily be maintained up-to-date.

4 Case Study: Operationalizations in a Teachmover Robot

The proposal has been validated in a real case study associated to the European project EFTCoR (Environmental Friendly and cost-effective Technology for Coating Removal) [9]. This project aims at designing a family of robots capable of performing

maintenance operations for ship hulls. The system includes operations such as coating removal, cleaning and re-painting of the hull. Among the subsystems constituting the EFTCoR platform, our case study focuses on the *Robotic Devices Control Unit* (RDCU), which interacts with other robotic devices to obtain the required information to control the different devices (positioning systems and cleaning tools) to be used for maintenance tasks. The RDCU is in charge of commanding and controlling, in a coordinated way, the positioning of devices together with the tools attached to them. However, the use of a real system would be too complex in order to exemplify this proposal, so TeachMover [33], a simplified version of the EFTCoR will be used in the remainder of this article. The TeachMover is a durable, affordable robotic arm used for teaching robotic fundamentals. It has been specifically designed to simulate industrial robotic operations. Similarly to what was said for the EFTCoR, this work focuses on the RDCU in charge of controlling this robot.

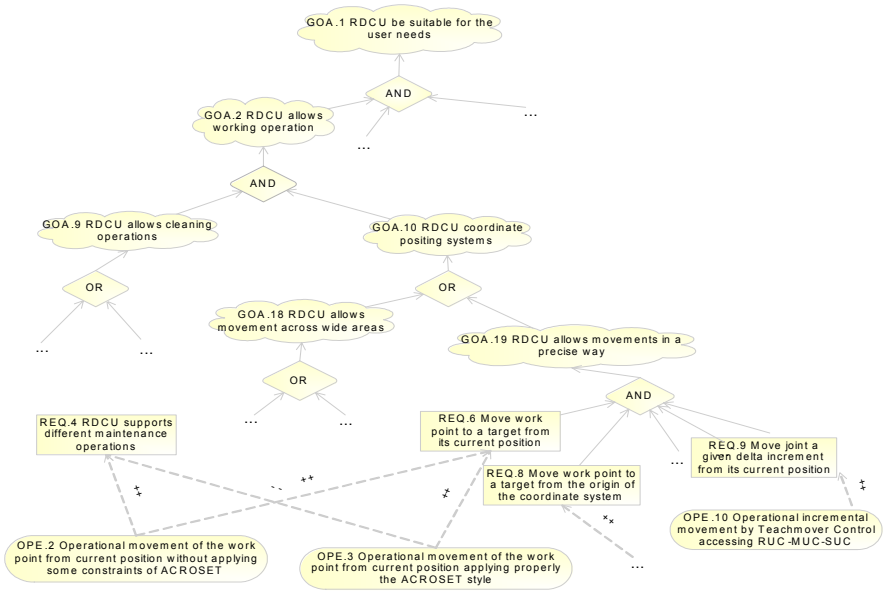

Fig. 4. Partial description of some of the *EFTCoR* requirements

Fig.4 shows a partial view of the Teachmover requirements that have been described using the ATRIUM Goal Model during the *Define Goals* activity. It depicts part of its functional requirements by refining the goal *Suitability*. It can be observed that the RDCU is expected to coordinate its movement, allow different cleaning operations in specific areas, and catch objects. These goals are refined into several goals, and finally into requirements. Fig. 4 also illustrates an example of how the requirement "REQ 6" is refined into two Operationalizations, namely:

- "*OPE.2 Operational movement of the work point from current position without applying some constraints of the ACROSET Style*". The DD described in this Operationalization is: "To perform the operational movement by allowing the direct access between systems RUC and SUC". The associated DR is: "The direct

access facilitates an advantage in terms of the number of operations to be performed, because they are not only limited to the active tool";

- "*OPE.3 Operational movement of the work point from current position, applying properly the ACROSET style*". The DD described along with OPE.3 is: "To perform the operational movement, by means of the interaction between the systems RUC-MUC-SUC". The related DR is: "This alternative is compliant with the ACROSET style; however, it exhibits problems because the number of operations that can be performed is only limited to the active tool".

Both the requirements and the operationalization are related by means of *contributions* relationships, that denote how the solutions contribute positively and/or negatively to meet the requirements. For example, it can be observed that the operationalizations "OPE.2" and "OPE.3" have a positive impact on the "REQ.6". However, the former has a positive impact on "REQ4" whereas the latter has a negative impact. It facilitates the analysis of which alternatives have less negative impact on the set of requirements. This example is used in the following to facilitate the comprehension of the presented work.

Associated to the *Operationalization* "OPE.2" an architectural scenario has been described, depicted in Fig. 5. It can be seen that the recommendations established by the DD have been followed, as the communication between the systems RUC and SUC has been directly established by means of components and connectors. When this scenario is described, the relationship *specifiedBy*, which defines its connection to the *Operationalization* "OPE.2", is established by the analyst that is shown as a contention relation in the Model Explorer situated on the left in Fig.5.

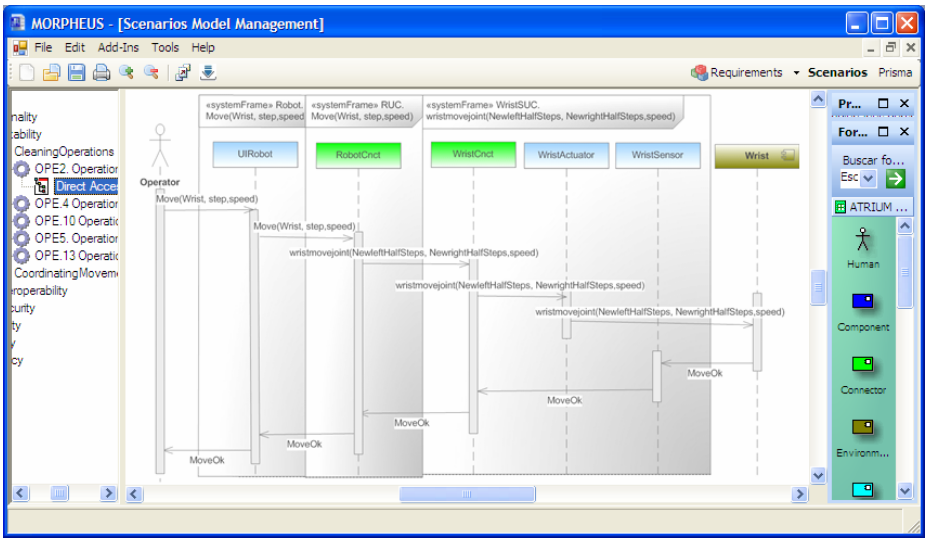

Fig. 5. Describing in the Scenario Environment an Architectural Scenario related to "OPE.2"

Once the Scenario Model has been described the activity *Synthesize and transform* can be applied to generate the proto-architecture. This activity can be performed when at least one scenario has been defined, and thanks to the *incrementality* feature of QVT as new architectural scenarios are defined the proto-architecture can be updated

to introduce the necessary changes. It is during this activity that the *DesignAsset* and traceability links are generated as well. According to the example described above, a *DesignAsset* will be generated in the target Architectural Model that will be related to each one of the generated Architectural Elements; in this case, those are the elements Robot, RUC and WristSUC. The following section describes the way in which this activity is supported by MORPHEUS.

5 MORPHEUS: Supporting the Proposal

Nowadays, automation is becoming one of the principal means to achieve greater productivity and higher quality products. For this reason, its introduction in this proposal was compulsory, both to provide support for the meta-modelling and modelling processes and to assist as much as possible in their description and exploitation. This led us to the development of a tool called MORPHEUS, a graphical environment for the description of the different models, to provide the analysts with an improved legibility. ATRIUM entails three main activities, and this has caused that MORPHEUS[2] has also been structured in three different environments:

- *Requirement Environment* [19] provides analysts with a requirements meta-modelling work context for describing Requirement Metamodels customized according to the project's semantic needs. This Environment automatically provides another work context for the description and analysis of Requirement Models according to the active Metamodel.
- *Scenario Environment* [21] has been expressly developed to describe the ATRIUM Scenario Model. This Environment facilitates both the graphical description of architectural scenarios meeting the established requirements and their later synthesis to generate the proto-architecture of the system being defined.
- *Software Architecture Environment* [26] makes available a whole graphical environment for the PRISMA AO-ADL [25] so that the proto-architecture synthesized from the Scenarios Model can be refined.

Fig. 6 shows the main elements integrating MORPHEUS. As can be observed, the *RepositoryManager* is in charge of controlling the access to the repository where the different Models and Metamodels are stored. Each Environment described above accesses to the repository by using this component. Above these Environments, the *Back-End* allows the analyst to access to the different Environments, and to manage the projects he/she creates. This paper focuses on the Scenario Environment, shown in Fig. 5, that provides access to the main functionality needed to support the proposal.

As depicted in Fig. 6, the Scenario Environment is made up by several components. The *ScenarioEditor* is one these components and provides the analyst with facilities for graphical modelling the ATRIUM Scenario Model. It must be highlighted that the graphical description of the Scenario Models was a must for the design of this component. For this reason, several graphical components were analysed in order to select the one with more capabilities.

[2]Interested readers can download a demo showing how this proposal has been put into practice using MORPHEUS, from http://www.dsi.uclm.es/personal/elenanavarro/research_atrium.htm.

Eventually, as shown in Fig. 6, Microsoft Office Visio Drawing Control 2003[35] was selected, because it allows a straightforward management, both for using and modifying shapes. This feature is highly relevant for our purposes, because all the kinds of concepts that are included in our Scenario Metamodel can easily have different shapes, thus facilitating the legibility of the Model. Fig. 5 shows what MORPHEUS looks like when the *ScenarioEditor* is loaded. It also shows another element, the Model Browser, placed on the left, which allows the user to access the Scenario Model being defined. This browser loads, automatically, the *Operationalizations* defined during the *Define Goals* activity, along with their corresponding trace from the *Requirements*. In this way, the user can easily understand why the scenarios are being defined. In addition, the stencil placed on the right provides the user with all the concepts described in the Scenario Metamodel, so that he/she only needs to drag and drop the relevant concept on the graphical view to perform its description in the Scenario Model.

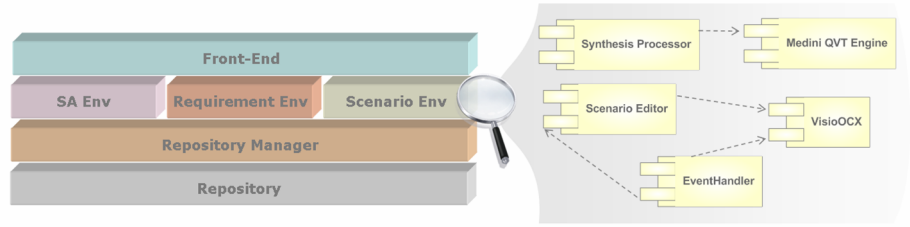

Fig. 6. An Overview of the MORPHEUS Architecture

Fig. 6 also shows that other component introduced in the development of the Scenario Environment was *SynthesisProcessor*, which applies the QVT Relations using a QVT engine called Medini [18]. With this purpose, it uses the Repository Manager to retrieve the active Goal Model and Scenario Model in XMI, and their respective Metamodels in ECORE. *SynthesisProcessor* provides Medini with them, and also with the name of the target Architectural Model and Metamodel, generating as a result a XMI file with the target Architectural Model. Once the Architectural Model has been generated, it can be loaded in the corresponding tool to be refined. If the user has chosen PRISMA as the target Architectural Metamodel to be instantiated, then MORPHEUS can be exploited to refine the Architectural Model, because it provides functionality to load such a description.

6 Related Work

As already explained in the Introduction, the specification of architectural rationales was an early requirement in the software architecture field, dating back to the very first article on the topic [27]. However, even in initial approaches, when architecture was considered largely as a way to document part of the system's design, the rationale was soon neglected. The reason was probably that at this early stage it was perceived basically as textual, unstructured documentation, and conceived as the kind of process information which was too often deemed as unimportant.

However, as software systems kept growing in size and complexity, architecture achieved ever more relevance; not only as a critical design asset, but also as a map or blueprint which would help to provide a global perspective of the system, and also to explain or describe the current stage during the software development process. Even in this context, DDs were still not explicitly documented, and hence all the assumptions and information about the design and its evolution, the *architectural knowledge*, was finally lost ("vaporized"). Indeed, if the architecture has a medium size, it is almost impossible to deduce the reasoning which led to the final structure, as the rationales behind concrete DDs are undocumented, and get ultimately forgotten. This is particularly unfortunate when considering evolution.

In recent times, Lago *et al* [16] were among the first authors in emphasizing the need to recover and maintain this information. Their proposal was not specifically constrained to the architectural level, but tried to include every design *assumption*, a term which would include our current architectural DDs, and which hinted towards an atomic, first-class specific module. Also, traceability was one of the reasons used to advocate the recovery of this information; however, it was not suggested to exploit it to provide a structure for complex DRs.

In any case, it was Bosch [1] who initiated the current interest in this topic. He was already implying that the approach was circumscribed to the context of architecture, and explicitly suggested to maintain this information as first-class *design decisions*, introducing what now has already become a specific term. A consequence of this separation of the rationale into small pieces is the resulting structure, which can be described as a plexus relating the whole architecture to the high-level rationale.

Some other initial works providing suggestions for this structure include Tyree and Akerman's [34], which proposes an ontology from an early industrial point of view, and the early survey by Tang, Babar *et al*, published in an evolved form in [32]. This survey also defines a framework to capture architectural knowledge, expressing the similarities between several approaches, and providing a first attempt to reconcile them. A different ontology is proposed by Kruchten *et al* [15], which divide the architectural knowledge into decisions, assumptions and context. This should be supported by tools, able to maintain also to eventually *evolve* this information.

Indeed, tool support has been one of the main interests in the area. There have been essentially two major approaches. The first one consists of describing DDs with some of the aforementioned structures, and then binding them to the architectural model, to provide an integrated perspective [14]. The other approach also describes those DDs, but it is more interested in defining platforms implementing strategies to share this knowledge between actors involved in the development process [7].

Garcia *et al* [8] use an aspect-oriented approach to tackle the description of architecture knowledge, namely describing DDs in a separate aspect [8]. In fact, this is reminiscent of our own approach, which also uses aspects. However, in ATRIUM aspects are not used just to describe DDs, but for almost everything; the process itself can be considered aspect-oriented, and in fact it covers every conceived *early aspect*, both at the requirements and at the architectural level.

Falessi *et al* [5] propose a goal-based, scenario-driven *decision model* to choose between alternatives, and to document the corresponding DR. They also detail some potential inhibitors which could hinder its use, and provide some hints towards their subsequent value-based approach [6]. Obviously, this has some points of contact with

ATRIUM, a consequence of both being goal-oriented. But their proposal lacks both the automatic support and the integration within a MDD process, which provides the traceability relationship and defines the strengths of our approach.

Our approach also has some common points with the already mentioned proposal by Jansen and Bosch [12]. They also provide a decision model, and even an extended metamodel which includes explicit DDs. The latter is, to some extent, similar to the one presented in section 3. However in our own case the extension was mandatory; otherwise new concepts would not have been explicit, and our DDs could have not been considered inside the MDD process.

Indeed most of the similarities between these proposals and the extended ATRIUM are casual or simply due to their common origin. In fact, in the existing literature there are only two works with a close resemblance. In the first one, Sinnema *et al* [30] suggest to exploit the variability in the architecture to support the definition of DDs. In ATRIUM this is indeed a consequence, not the starting point; but this is still the closest suggestion ever done to (implicitly) support DDs on top of traces.

In the second one, Mattson *et al* [17] remind that capturing design information is a feature of any MDD approach; also, that architecture itself is not fully integrated into MDA. Therefore, they provide a basic architectural framework, and then propose to formalize design rules, which should be enforced by the MDD process. However no more details are provided; though this is the only paper suggesting using MDD in this context, it is far from being complete.

In fact, both Sinnema's and Mattson's are preliminary short papers, sketching some interesting ideas which have not yet being exploited by further research. Indeed, our approach fulfils many of the promises implicit in those papers; for example, in ATRIUM traceability links can be used to define a variation point in any part of the process, and bind it to any explicit decision; and both the architectural framework and the exploitation of traceability within the MDD process and transformations are much more complex than any previous suggestion.

7 Conclusions and further work

As just noted in the previous section, the extended ATRIUM methodology and the tool support provided by MORPHEUS provide a support for traceability which allows a complex and advanced framework for the definition of architectural knowledge. From this perspective, their capabilities exceed those of any existing proposal, at least in terms of automation, generation and tracing. The rationale is defined in a composite structure which can be related to architecture itself, defining an elaborate network of first-class DDs. The consequences of using this approach are also fairly unknown; it obviously supports the definition of complex structures, but the great potential they provide has yet to be investigated.

For instance, there is a line of research which intends to deduce undocumented processes from their architecture [13]. To do so, they must extract the *deltas* between different versions, and deduce the nature of the decisions made. In our approach, this is not necessary, even when DD/DRs have not been originally provided; instead of defining deltas by comparing different versions, MORPHEUS can be used to generate

a whole new version starting from a single different decision. The potential to exploit variability is enormous, but it can be also applied to specific issues like this.

Compared to other tools in the field, MORPHEUS provides a lot of facilities and complex functionality. Most of the others try just to either describing or sharing some static information; only a few of them try to actually integrate the information in the process and let it play an active role [14][17][30]. This even transcends the limits of the specific subfield of architectural knowledge, to be applied for concerns involved in any aspect of the architectural process. In this regard, the presented approach not only fulfils its specific purpose, but also provides a core model which can be applied to fulfil some of the traditional requests in the architecture field.

In summary, the proposal described in this paper provides both the technology and the methodology to define and explicitly manage architectural knowledge; this is done in a consistent and principled way, and providing an optimal integration. But this does not *just* describe a model-driven variant of a known idea, or provides a particular tool to deal with it. In fact, the most important contribution of this paper is the outlining of the explicit relationship to *traceability*, the explanation of how this provides the basic structure for the architectural rationale, and the consequences of this approach.

Acknowledgments. This work has been funded by the Spanish Ministry of Education and Science under the National R&D&I Program, META Project TIN2006-15175-C05-01. Further funding comes from Rey Juan Carlos University and the autonomous Government of Madrid under the IASOMM Project URJC-CM-2007-CET-1555.

References

[1] J. Bosch: Software Architecture: The Next Step. In: 1st European Workshop in Software Architecture (EWSA'04), 194-199, LNCS 3047, Springer, Heidelberg (2004).

[2] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishing, Boston Hardbound (2000).

[3] K. Czarnecki, S. Helsen: Classification of Model Transformation Approaches. IBM Systems Journal, 45(3), 621-645 (2006).

[4] A. Dardenne, A. van Lamsweerde, S. Fickas: Goal-directed Requirements Acquisition, Science of Computer Programming, 20(1-2), 3-50 (1993).

[5] D. Falessi, M. Becker, and G. Cantone: Design Decision Rationale: Experiences and Steps Ahead Towards Systematic Use. In SHARK '2006, ACM DL, New York (2006).

[6] D. Falessi, G. Cantone, P. Kruchten: Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study. In: 7th Working IEEE/IFIP Conf. on Softw. Architecture (WICSA'08), 189-198, IEEE CS, New York (2008).

[7] R. Farenhorst, P. Lago, and H. van Vliet. EAGLE: Effective Tool Support for Sharing Architectural Knowledge. Intl. J. Cooperative Information Syst., 16(3-4), 413-437 (2007).

[8] A. Garcia, T. Batista, A. Rashid, C. Sant'Anna: Driving and Managing Architectural Decisions with Aspects. In: SHARK '2006, ACM DL, New York (2006).

[9] GROWTH G3RD-CT-00794: EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project, 5th Framework Program (2003).

[10] N. B. Harrison, P. Avgeriou and U. Zdun: Using Patterns to Capture Architectural Decisions. IEEE Software 24(4):38-45 (2007).

[11] ISO/IEC Standard 9126-1, Software Engineering- Product Quality-Part1: Quality Model, ISO Copyright Office, Geneva, (2001).

[12] A. Jansen and J. Bosch: Software Architecture as a Set of Architectural Design Decisions. In: 5th Working IEEE/IFIP Conf. on Softw. Architecture (WICSA'05), 109-120, (2005).
[13] A. Jansen, J. Bosch, P. Avgeriou. Documenting After the Fact: Recovering Architectural Design Decisions. Journal of Systems and Software, 81(4), 536-557 (2008).
[14] A. Jansen, J. van der Ven, P. Avgeriou and D.K. Hammer. Tool Support for Architectural Decisions. In 6th Working IEEE/IFIP Conf. on Software Architecture (WICSA'07), p. 4, IEEE CS Press, New York (2007).
[15] P. Kruchten, P. Lago, H. van Vliet: Building Up and Reasoning about Architectural Knowledge. In: 2nd Intl. Conf. Quality of Sofware Architectrures (QoSA'06), LNCS 4214, pp. 43-58, Springer, Heidelberg (2006).
[16] P. Lago, H. van Vliet: Explicit Assumptions Enrich Architectural Models. In: 27th Intl. Conf. on Soft. Engineering (ICSE'05), pp. 206-214, IEEE CS Press, New York (2005).
[17] A. Mattsson, B. Lundell, B. Lings, B. Fitzgerald: Experiences from Representing Software Architecture in a Large Industrial Project using Model-Driven Development. In: SHARK/ADI 2007, IEEE DL, New York (2007).
[18] Medini , QVT Relations, http://projects.ikv.de/qvt .
[19] E. Navarro, P. Letelier, A. Gómez: MORPHEUS: tool support to tailor requirements management to the specific project needs. Inf. & Soft. Technology (2008) (submitted).
[20] E. Navarro, P. Letelier, I. Ramos: Requirements and Scenarios: playing Aspect Oriented Software Architectures. In: 6th Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), (short paper) IEEE DL, New York (2007).
[21] E. Navarro, P. Letelier, J. Jaén, I. Ramos: Supporting the Automatic Generation of Proto-Architectures. In: 1st European Conference on Software Architecture (ECSA 2007), 24 - 28, LNCS 4758, pp. 43-58, Springer, Heidelberg (2006) (Best poster award).
[22] E. Navarro: Architecture Traced from RequIrements applying a Unified Methodology, PhD thesis, Computing Systems Department, UCLM (2007).
[23] E. Navarro, P. Letelier, J. A. Mocholí, I. Ramos: A Metamodeling Approach for Requirements Specification, J. of Computer Information Systems, 47(5), 67-77 (2006).
[24] OMG document ptc/05-11-01, QVT, MOF Query/Views/Transformations. Final adopted specification (2005).
[25] J. Pérez, N. Ali, J. A. Carsí, I. Ramos: Designing Software Architectures with an Aspect-Oriented Architecture Description Language", In: CBSE'06 , LNCS, vol. 4063, pp. 123-138. Springer, Heidelberg (2006).
[26] J. Pérez, E. Navarro, P. Letelier, I. Ramos: A Modelling Proposal for Aspect-Oriented Software Architectures. In: 13th IEEE Int. Conference and Workshop on the Engineering of Computer Based Systems (ECBS), 32-41, IEEE CS, New York (2006).
[27] D.E. Perry, A.L.Wolf. Foundations for the Study of Software Architecture. ACM Software Engineering Notes, 17(4), 40-52 (1992).
[28] B. Selic. The Pragmatics of Model-Driven Development. IEEE Soft. 20(5), 19-25 (2003).
[29] OMG, Software Process Engineering Metamodel (SPEM), Version 1.1 formal/05-01-06, http://www.omg.org/cgi-bin/doc?formal/2005-01-06, (2005).
[30] M. Sinnema, J. van der Ven, S. Deelstra. Using Variability Modeling Principles to Capture Architectural Knowledge. In: SHARK '2006, ACM DL, New York (2006).
[31] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, G., Karsai: Domain Model Translation Using Graph Transformations. In: 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), pp. 159-167, IEEE CS (2003).
[32] A. Tang, M.A. Babar, I. Gorton and J. Han: A Survey of Architecture Design Rationale. Journal of Systems & Software, 79(12), 1792-1804, (2007).
[33] TeachMover, http://www.questechzone.com/microbot/teachmover.htm.
[34] J. Tyree and A. Akerman. Architecture Decisions: Demystifying Architecture. IEEE Software, 22(2):19-27 (2005).
[35] Visio 2003, http://office.microsoft.com/es-es/FX010857983082.aspx.