Requirements Traceability and Transformation Conformance in Model-Driven Development

João Paulo Almeida[1], Pascal van Eck[2], Maria-Eugenia Iacob[1],

[1]*Telematica Instituut, Enschede, The Netherlands*
[2]*Centre for Telematics and Information Technology, University of Twente, The Netherlands*
joaopaulo.almeida@telin.nl, p.a.t.vaneck@ewi.utwente.nl, maria-eugenia.iacob@telin.nl

Abstract

The variety of design artefacts (models) produced in a model-driven design process results in an intricate relationship between requirements and the various models. This paper proposes a methodological framework that simplifies management of this relationship. This framework is a basis for tracing requirements, assessing the quality of model transformation specifications, metamodels, models and realizations. We propose a notion of conformance between application models which reduces the effort needed for assessment activities. We discuss how this notion of conformance can be integrated with model transformations.

Keywords: requirements traceability, assessment, conformance, model transformation, model-driven design

1. Introduction

Model-driven design holds the promise of improving application development significantly by capturing design steps in explicit model transformations [22]. The design of an application in model-driven design can be seen as the process of building a realization of the application specification that satisfies all application requirements stated in the specification by applying appropriate transformations.

At several stages in the application lifecycle, application maintainers need to know which application models and/or components satisfy requirements that have been explicitly stated. This relation between requirements and elements of the solution (e.g., application models and components) is called requirements traceability. Requirements traceability is for instance used during acceptance testing, when application users (or procurers) are interested in assessing the extent to which an application adheres to its requirements.

We observe that, in a model-driven design process, the great variety of modelling artefacts pose challenges to requirements traceability and assessment. Not only application realizations have to be assessed for requirements satisfaction, but also application models, metamodels and model transformation specifications since these may also be considered products of the model-driven design process.

The main contribution of this paper is to provide a methodological framework which allows designers to relate requirements to the various products of the model-driven design process. This framework is a basis for tracing requirements and assessing the quality of model transformation specifications, metamodels, models and realizations.

Since the model-driven design process may consist of different levels of abstraction (and platform-independence [6]), requirements are traced throughout these levels. We propose a notion of conformance between models which simplifies requirements tracing. The idea is that transformations which are assumed to produce conformant results can be reused, deeming some assessment activities redundant.

This paper is structured as follows. Section 2 provides some background in the area of requirements engineering. Section 3 defines basic notions of model-driven design required in this paper. It defines the notion of satisfaction of requirements in terms of the relation between requirements, the various application models and realizations of an application. Section 4 defines and justifies the notion of conformance between models proposed here. Section 5 extends the view of the model-driven design process defined in section 3 by introducing automated model transformation chains. This allows us to discuss how conformant transformations can simplify assessment activities. Section 6 identifies the role of application-independent requirements in model-driven design. Section 7 illustrates the approach with an example. Section 8 discusses assumptions and limitations of our framework. Section 9 discusses related work, and finally, section 10 presents our conclusions and outlines topics for further research.

2. Requirements Engineering

The term Requirements Engineering (RE) refers to the phase in application development in which requirements of different stakeholders are gathered and processed, in general resulting in a requirements specification or software specification. Requirements can be formulated as either properties of the problem that the stakeholders want to solve using the application under development or desired properties of that application. This phase is called requirements engineering to indicate that more is needed than only requirements elicitation: requirements have to be processed to resolve conflicts, prioritized, and captured in a consistent requirements specification.

We assume in this paper that a requirement specification is verifiable [12, 17], i.e., given a realization, it is possible to conduct assessment activities to determine whether the requirements can be considered satisfied. We use the term "assessment activity" for the act of checking whether a requirement is satisfied. Examples of assessment activities are acceptance testing by end users, model checking or formal correctness proofs.

We conceptualize requirements as implicitly defining a set of application realizations that satisfy them. Figure 1 shows the relation between requirements and the space of possible realizations. An arbitrary grouping of the requirement specification into sets $RS_A \subset RS_B \subset RS_C$ is considered. The realization sets IS_1, IS_2, and IS_3 represent realizations that satisfy RS_1, RS_2, and RS_3 respectively. The realizations IS_C that satisfy the total set of requirements RS_C (the union of RS_1, RS_2, and RS_3) lie in the intersection between IS_1, IS_2, IS_3. Note that this is a conceptual notion, independent of whether requirements are formalized.
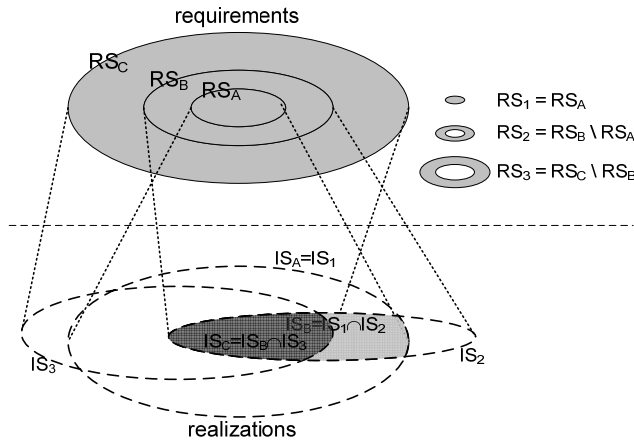

Figure 1 Requirements and realizations

In this paper we address *traceability of requirements*. Several definitions of traceability can be found in [16]. The one most suitable for the purpose of this paper is: "the means whereby software producers can 'prove' to their client that: the requirements have been understood;

the product will fully comply with the requirements; and the product does not exhibit any unnecessary feature or functionality" ([27], as quoted in [16]). Our notion of assessment activities exactly operationalises the notion of 'prove' in this definition. In terms of the IEEE Recommended Practice for Software Requirements Specifications [17], we are interested in forward traceability, in which artefacts (documents, in our case: models) constrained by the requirements specification need to be traced back to the requirements specification.

In order to trace requirements throughout the design process, we partition the set of requirements into subsets as illustrated in Figure 1. The partitioning strategy is discussed in the remainder of this paper.

3. Requirements and Artefacts in Model-Driven Design

Before we discuss how requirements are related to the several different artefacts in model-driven design, we need to guarantee some common understanding of the model-driven design process and of these artefacts.

3.1. Artefacts in Model-Driven Design

Model-driven design is based on capturing different aspects of a (distributed) application into symbolic artefacts known as *models*. Models are manipulated throughout the design process resulting ultimately in one or more realizations of the application. The manipulation of application models in a model-driven design process often entails model transformation activities [25] which may be determined or constrained by (*model*) *transformation specifications*. These specifications or their implementations may be executed automatically, with the purpose of improving the overall efficiency of the design process. In this paper, we consider that transformations are used to relate source and target models at different levels of abstraction. The notions of source and target models are thus relative to a design step.

Models are expressed in suitable (general purpose or domain-specific) modelling languages, with their abstract syntax described in models known as *metamodels*.

Model transformation specifications and metamodels are defined in an application-independent phase of the model-driven design process (known as the preparation phase in [2, 13]). They are used by designers to build specific applications. In this context, model transformation specifications capture reusable design knowledge, and metamodels capture reusable concepts and patterns for application modelling.

Figure 2 shows an example of model-driven design trajectory, depicting schematically the dependencies between the various artefacts. Three levels of models are

shown. In the lowest level of models, two alternative application models are produced (M3 and M3'), which are defined in terms of different metamodels. The figure also depicts model libraries which consist of reusable models.
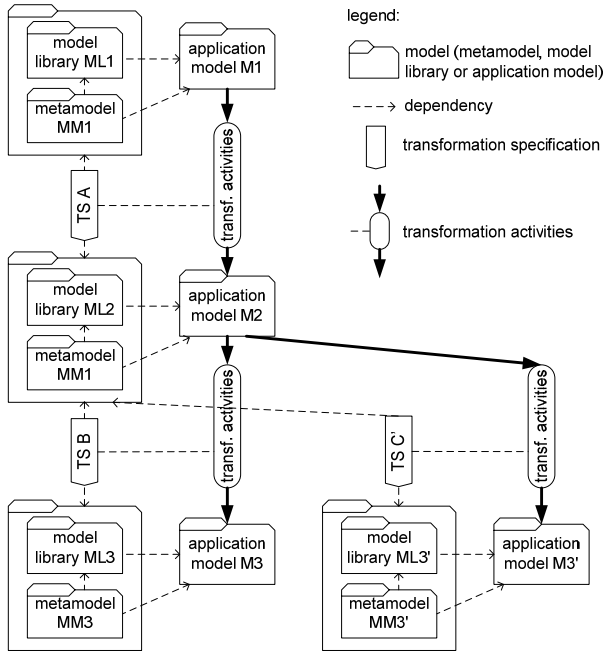

Figure 2 Artefacts in a model-driven design trajectory

3.2. Requirements and Application Models

The multitude of artefacts in model-driven design serves the ultimate purpose of producing application realizations that satisfy a particular set of requirements. Usually, there are (virtually infinitely) many application realizations that satisfy a set of requirements. The design task consists of obtaining a particular application realization that satisfies requirements while respecting implementation constraints and general design principles. Figure 3 illustrates the relation between requirements and application models at different levels of abstraction.
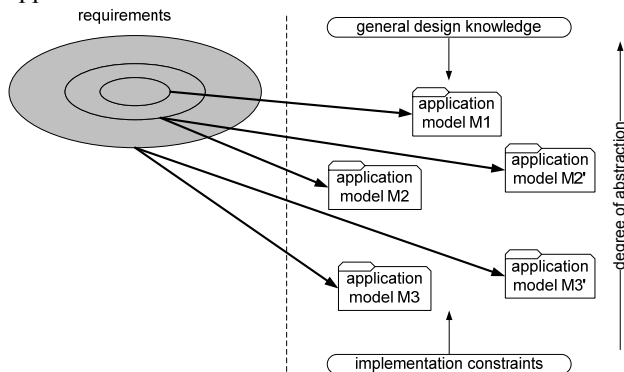

Figure 3 Requirements and application models

We assume that application models capture design decisions, defining characteristics of a potential application realization. Furthermore, we require that models have a well-defined semantics. More precisely, we say that a model has a well-defined semantics, if, and only if, given a realization and a model, it is possible to determine whether the realization exhibits the characteristics as defined in the model. The means by which this semantics is defined (e.g., mapping to a formal domain, natural language descriptions, or basic set of design concepts) is not prescribed by this definition.

We can conceptualize models as implicitly defining a set of realizations that realize them. Figure 4 (adapted from [4, 25]) depicts the relation between models and the space of realizations. In this figure, an oval represents the sets of acceptable realizations for a particular model. Different design decisions may lead to alternative realizations, and this is shown by different sets of realizations (shaded) for alternative models (M2 and M2', M3 and M3').
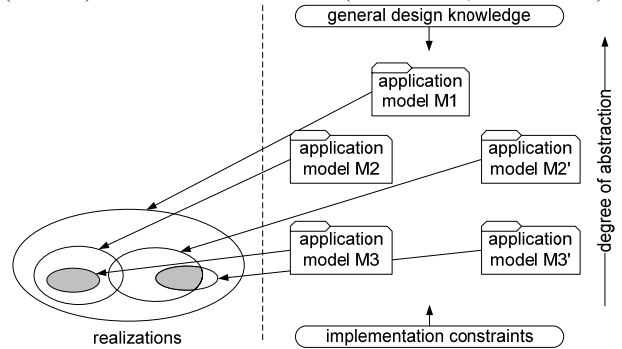

Figure 4 Models and the space of realizations

Design decisions should eventually lead to a design that defines all relevant characteristics of a realization of the system [4], satisfying all stated requirements and implementation constraints. It is not our intention to debate the distinction between realizations and models. For our purposes, a model that satisfies all requirements can be considered a realization. For example, a workflow model executed in a workflow engine can be considered a realization, with no further transformation.

Figure 5 shows requirements, models and realizations in one picture (combining Figures 3 and 4). It reveals the (indirect) relation between requirements and realization.

As can be observed in this figure, the set of realizations for an application model M1 is contained in the set of realizations that satisfy RS1. The set of realizations for an application model M3 is contained in the set of realizations that satisfy RS3. Application models M2' and M3' have been omitted for the sake of conciseness.

At this point, we can formulate the notion of satisfaction of requirements by models. We say that a model M *satisfies a set of requirements* RS, if and only if, the set of

acceptable realizations for M is contained in the set of realizations that satisfy RS.

In order to support requirements traceability, it is the task of the designer to state which requirements are satisfied by which models, and to conduct assessment activities to support such claims of satisfaction. In the remainder of this paper, we work out which claims are required and discuss how they can be managed in a model-driven design process.
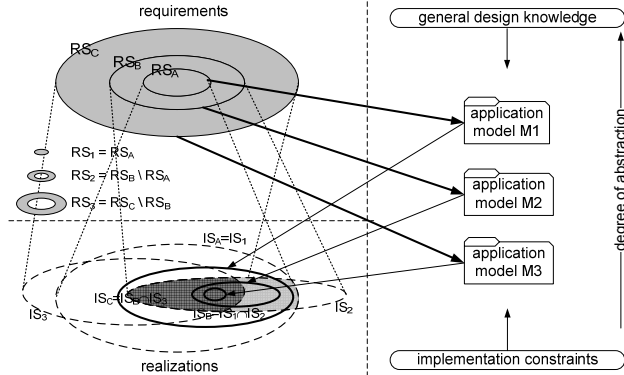

Figure 5 Requirements, models and realizations

4. Preserving Satisfaction of Requirements through Conformance

The notion of conformance between models is central to our approach. We say that a model MT *conforms to another model* MS if, and only if, the set of acceptable realizations for MT is contained in the set of acceptable realizations for MS. Given this definition, we can observe that Figure 4 depicts both conformant and non-conformant pairs of models. For example, M3' does not conform to M2', while M2' conforms to M1.

If a model at a lower level of abstraction (Mi+1) does not conform to a model at the previous level (Mi), a designer is forced to consider both Mi+1 and Mi in a subsequent design step. This problem is exacerbated in the presence of multiple levels of abstraction that are not related by conformance. In the extreme case, a designer has to consider all models in a design step that produces the realization. This problem is addressed with conformant models. Conformant models can be regarded as replacing the models they conform to. For example, Figure 5 shows only conformant models M1, M2 and M3. Thus, in the design step from M2 to M3, M1 does not have to be considered. Further, M3 is sufficient to derive the realization.

Intuitively, a model creates a sort of a "mould" such that all subsequent models should fit into it ("conform"). The same is not necessarily true with sets of requirements, which can be regarded as defining constraints that have to be considered in conjunction.

By populating a hierarchy of models with models that conform to models at a higher level of models, designers can simplify requirements traceability activities. This is possible because requirements satisfied by a model are also satisfied by all models that conform to it. The evidence showing that a model satisfies certain requirements can be reused for all models that conform to it.

In a design step that produces a conformant target model, the designer only has to provide evidence for supplementary requirements that are satisfied in the target model but not in the source model. In Figure 5 this means that assessment of M2 only requires one to show evidence for satisfaction of RS2 instead of both RS1 and RS2.

Further, modification of models at a lower level of abstraction does not affect models at a higher level of abstraction if the modified model remains conformant.

We can now observe that the partitioning of requirements in different sets as depicted in Figure 5 arises from the way in which the various sets of requirements are addressed throughout the model-driven design trajectory.

5. Requirements Traceability with Transformation

This section extends the view of the model-driven design process as described in section 3 with model transformation chains.

5.1. Automated Transformation Chains

We start by considering fully automated transformation chains. Fully automated transformation chains consist of a predefined series of transformation specifications that can be applied to relate different subsequent levels of models. All transformation activities are automated using the various transformation specifications. An application model that is used as input for the transformation chain is sufficient to obtain a realization of the application.

In the case of automated transformation chains, application requirements only influence the application model. This is shown in Figure 6. Note that there are no relations between model transformation specifications (TSA and TSB) and application requirements. The reason for this is that model transformation specifications capture application-independent design operations that can be reused in the development of several applications.

A useful analogy for automated transformation chains is the programming language compiler: source code can be regarded as the application model, and assembler code can be considered the realization on a target hardware platform (with intermediate representations often used for optimization purposes). The specification of the compiler (i.e., the model transformation specification) is independent of the applications compiled by the compiler.

In the particular case of automated transformation chains, assessment activities can be summed up in (i) assessing whether the application model satisfies application requirements, and (ii) whether Mi+1 conforms to Mi for every transformation step (a special kind of assessment we call *conformance assessment*). In case models at intermediate levels are not considered reusable products of the design process, it suffices to assess whether the last model conforms to the first model.
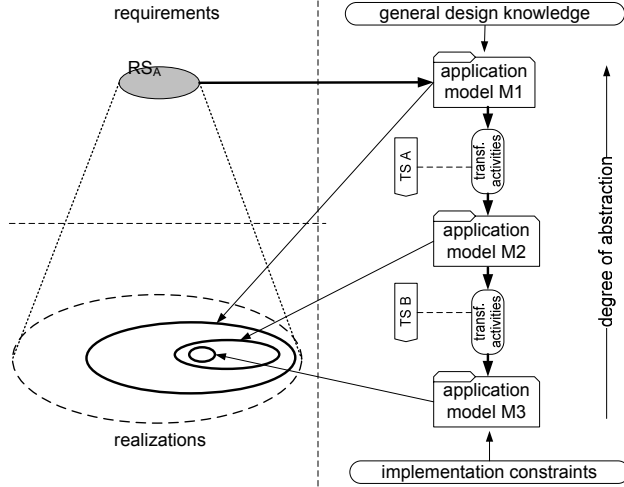

Figure 6 Application-specific requirements only affect the highest level of models

When a transformation chain is assumed to produce conformant results, the only required assessment activity left is assessing whether the application model satisfies application requirements. Other assessment activities are deemed redundant by the assumption of conformance. In the analogy of a programming language compiler, only source code is assessed if the compiler can be trusted.

To capture this reorganization of assessment activities in terms of the quality of a transformation specification, we define the notion of a *conformant transformation specification*. We say that a transformation specification is *conformant*, if, and only if, for every source and target models related by the specification, the target model conforms to the source model.

5.2. Partially Automated Transformation Chains

As we have discussed in the last section, the traceability of requirements can be largely simplified for the case of fully automated transformation chains with conformant transformation specifications. However, full automation of transformations is not always feasible or desirable. For example, it may be impossible to derive relevant design decisions from an high-level application model, or it may be inefficient to specify automated transformations that have a limited reuse potential (see [2] for an analysis on the costs/benefits of automated transformation). We dis-

tinguish the following approaches to decrease the level of automation without manual modification of target models:

(i) *transformation parameterization*, in which case the designer selects values for transformation parameters, i.e., arguments. Transformation parameters capture variation in the way source and target models are related; and,

(ii) *selection of transformations*, in which case a designer configures a transformation chain from a number of alternative predefined transformations. In order to simplify our discussion, we regard selection of alternative transformations as a special case of transformation parameterization, where a transformation specification includes the relations specified by all alternative transformations, and arguments are used to select an alternative.

In this case, application requirements influence transformation arguments. This is depicted in Figure 7.

The definition of a conformant transformation specification can be easily adjusted to incorporate transformation arguments. A transformation specification is said to be *conformant*, if, and only if, for every source and target models related by the specification *under every admissible set of transformation arguments*, the target model conforms to the source model.
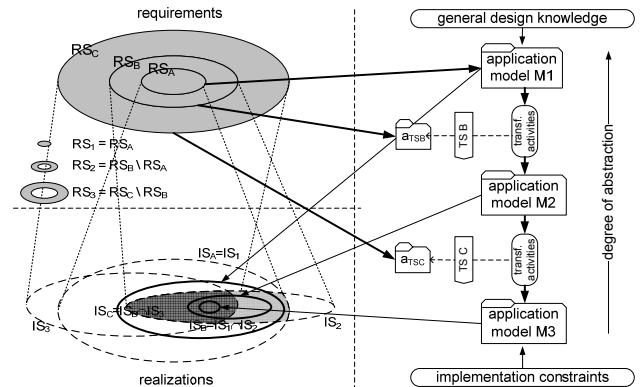

Figure 7 Application-specific requirements affect transformation arguments

For a transformation chain with parameterized conformant transformation specifications, one should assess whether the application model at level-1 satisfies application requirements RS1, and whether design decisions implied by transformation arguments satisfy different partitions of requirements (RSi).

5.3. Manual Modification

If necessary, the level of automation may be further lowered by allowing designers to manually modify target models. We assume in this case that modification is not unconstrained: the relations between source and target models as defined in a transformation specification should be respected. (Although tool support may allow

these relations to be temporarily violated, as long as they are re-established.) Figure 8 shows the relation between requirements and the various levels of models for this case.

Assessment activities in this case include: assessing whether models are conformant, and assessing whether the partitions of requirements (RSi) are satisfied progressively. If transformation specifications can be assumed conformant, this is simplified to assessing the satisfaction of the different partitions of requirements (RSi) at the different levels of models. Manual modification may be combined with parameterization. We do not show that in this paper due to space restrictions.
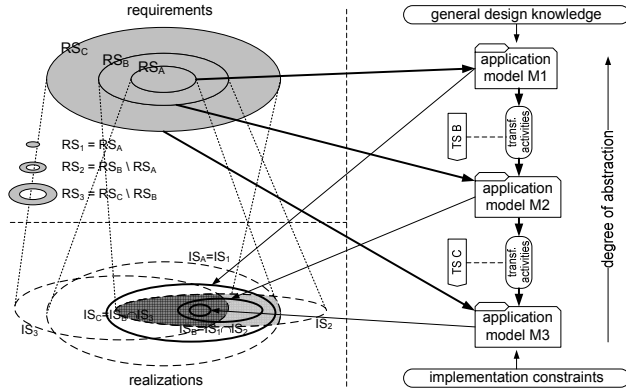

Figure 8 Application-specific requirements affect application models

6. Application-Independent Requirements

So far, we have discussed how application requirements can be traced throughout the design trajectory. In addition to application requirements, in a model-driven design process we also identify *application-independent requirements,* which impact design decisions for a whole class of applications, instead of a specific application. As opposed to application requirements, application-independent requirements are not the concern of the users or procurers of an application. Instead, they are the concern of the designers that oversee the model-driven design process, and arise from the repeated application of certain design manipulations, patterns and structures. Application-independent requirements often entail requirements on the modelling languages used (e.g., "UML shall be used for information modelling"), on architectural styles or frameworks (e.g., "the service-oriented discovery pattern shall be used") and on the platforms used (e.g., "Web Services shall be used for all interactions across firewalls", "CORBA shall be used for all interactions inside the organization").

These requirements are satisfied by reusable design decisions that are captured in transformation specifications, metamodels, reusable model libraries, "abstract

platforms" [5, 6], target platforms and other "reuse infra-structures" [8]. Similarly to the proposed partitioning of application requirements, application-independent requirements can also be partitioned and addressed at different levels of models. Figure 9 illustrates that. Application-specific requirements and their relations with the various models have been omitted.
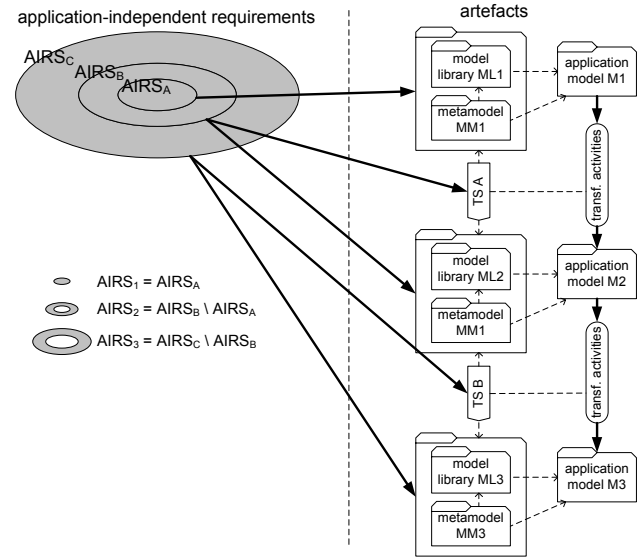

Figure 9 Application-independent requirements and transformation specifications, metamodels and model libraries

Application-independent requirements are not orthogonal to application-specific requirements. For example, if real-time properties of a particular application process are relevant (as defined in an application requirements specification), languages such as BPEL or BPMN may be considered inadequate to describe this process. If application-specific requirements are given priority over application-independent requirements, a design trajectory that supports only these languages (as result of application-independent requirements) should be discarded. Conversely, giving priority to the set of application-independent requirements, would rule out possible (classes of) application-specific requirements, and therefore, restrict the generality of the design trajectory.

7. Example

To show how our approach reduces assessment activity effort for traceability, we present as an example the design of a telemonitoring system ([7]). The goal of this system is to monitor a chronically ill patient continuously and warn the patient and care givers (e.g., at a hospital) of critical health conditions.

7.1. Requirements

Table 1 presents 13 requirements for a specific telemonitoring system, which issues alarms for epileptic seizures.

ID	Description
AR1	Upon detection of an (eminent) epileptic seizure, the patient shall be alarmed.
AR2	Upon detection of an (eminent) epileptic seizure, aid persons in the surrounding of the patient may be alarmed.
AR3	Only aid persons with an available status are alarmed.
AR4	In case no aid persons can be alarmed an emergency health care team will be alarmed.
AR5	In case the epileptic seizure occurs at a speed higher than 8km/h, an emergency health care team will be alarmed (instead of aid persons) (rationale: this may involve high risk, e.g., if the patient is biking, jogging, driving).
AR6	Alarms to aid persons or health team inform them of the last known location of the patient.
AR7	Alarms may be realized through short messaging service or calling aid persons with voice messages (rationale: mobile phones are cheap devices, known to aid persons).
AR8	Patient location and speed may be determined through GPS devices.
AR9	Patient and aid person location may be determined through Parlay-X.
AR10	Aid person availability status may be determined through Parlay-X presence.
AR11	In case patients/aid persons should carry mobile devices for monitoring, these should allow uninterrupted monitoring for 24 hours, without requiring battery recharges.
AR12	As a guideline, costs of mobile communication should not exceed EUR 50,- per month per patient.

Table 1 Application-specific requirements

7.2. Application-independent requirements

The telemonitoring system can be categorized as a context-aware distributed application. Patients and care givers are not only geographically distributed but also mobile. Their location, speed and biosignals are considered context that is relevant for the behaviour of application. This class of applications is supported by a model-driven design trajectory as defined in [3, 7]. We assume that applications in this class have been designed repeatedly, e.g., by a software house that specializes in this class of applications. As a result, the design process has been captured in the form of different levels of models, transformations between these levels and a number of platforms.

Three levels of models are defined: the service specification level, the platform-independent service design level and the platform-specific service design level.

At the level of service specification a service can be described in terms of *events*, which represent contextual changes, *queries* to providers of context information (so-called context sources), and *actions*, which represent actions to be performed in order to provide the service to the user. These elements can be expressed in a domain-specific language (called ECA-DL [3, 7]). In this example, we have chosen to express behavioural aspects at the level of platform-independent service design in ISDL models [26] and OCL constraints. UML class diagrams (omitted here) are used to represent information models.

The transformation between the service specification and the service design level consists of refining events, queries and actions at the service specification level into sequences of interactions in the service design. At the service specification level, an action represents an activity performed by the system as a whole (including any context sources and action services). However, at the service design level the same action has to be performed by co-operation of different services, in a service-oriented design which includes various context and action services. Transformation rules are defined extensively in [3]. TSA is parameterized with constraints for the services that can be used to realize events and actions in M1 (so it can be considered a partially automated transformation chain).
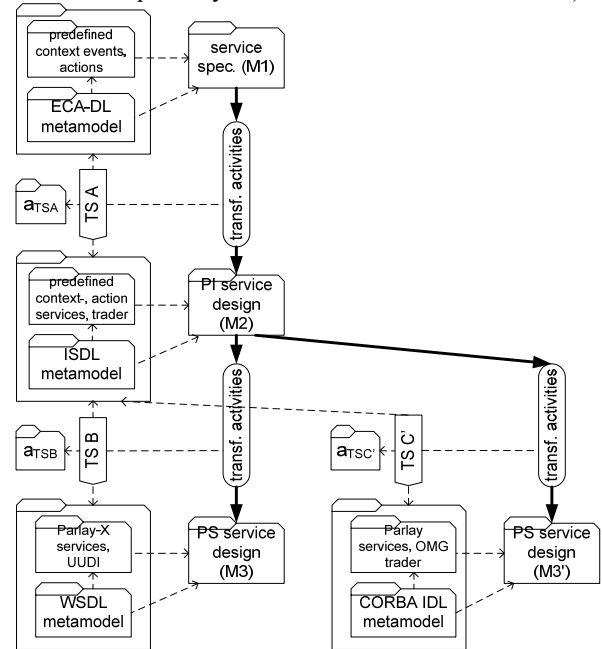

Figure 10 Design trajectory for context-aware mobile services (adapted from [7])

Table 2 presents application-independent requirements for part of the design trajectory depicted in Figure 10.

Thick borders delimit the partitioning of requirements so that they can be traced to the different levels of models: AIR1–AIR5 are addressed at the service specification level (and result in the metamodel of the ECA-DL language); AIR6–AIR11, are addressed at the platform-independent service design level (and result in TSA and the abstract platform at that level); AIR12–AIR16 are addressed at platform-specific level (and result in TSB and the target platforms, which we will denote with P3).

ID	Description
AIR1	Application designers should be able to describe the behaviour of a context-aware application at a high-level of abstraction.
AIR2	This behaviour may refer to the occurrence of context events (such as, e.g., epileptic seizure).
AIR3	This behaviour may prescribe the execution of actions (such as, e.g., alarming patients).
AIR4	Ability to describe constraints on actions based on context information (such as, e.g., that only available aid persons in the surroundings of a patient are alarmed).
AIR5	This behaviour may refer to context information (e.g, an alarm may include patient location information).
AIR6	Application design should be based on the reuse of context and action services.
AIR7	It should be possible to add reusable services at service run-time.
AIR8	Basic reusable services for application design should be provided. These should include services provided by mobile telecommunications networks (sending SMS, establishing calls, determining location).
AIR9	It should be possible to simulate designs.
AIR10	Service designs should be service-platform-independent.
AIR11	Service components should specify required services, which can be discovered and bound at runtime.
AIR12	End-users may access services through mobile phones, smart-phones, PDAs, PCs and plain-old telephones.
AIR13	Service realizations should be supported by a standardized middleware platform.
AIR14	Distribution between client-side and back-end side is supported by Web Services protocols on top of GPRS.
AIR15	Service trading is realized through UDDI in the back-end.
AIR16	Services of the telecommunication network are provided by Parlay-X in the back-end.

Table 2 Application-independent requirements for context-aware mobile services

7.3. Models

We will only discuss the service specification and platform-independent service design levels in order to limit the size of this example.

The Telemonitoring service specification is depicted in Figure 11 (this is M1 in Figure 10). Ovals represent context events, queries and actions. The suffix _indC indicates a context event, the suffixes _reqC, _rspC indicate a request-response query to context sources and the suffixes _reqA, _rspA indicate request-response to action services. Arrows indicate enabling relations between events, queries and actions; white diamonds represent choice (or-split) and white squares denote disjunction. Guards for enabling relations and constraints for information are depicted in boxes attached to context events, queries and actions.

The platform-independent service design is the result of the application of all transformation rules to the service specification. Figure 12 (this is a part of M2 in Figure 10) shows the generated coordination component. The dashed lines represent causality relations already present in the service specifications. Semi-ovals represent interactions in ISDL.

The generated coordination component interacts with a service trader to find context and action services. The service queries are generated from constraints at the service specification level, which are indicated in arguments a_{TSA} to the transformation (in this case, alertAid_reqA.aidperson_xy and alertTeam_reqA.coverageArea as marked with boxes in Figure 11).
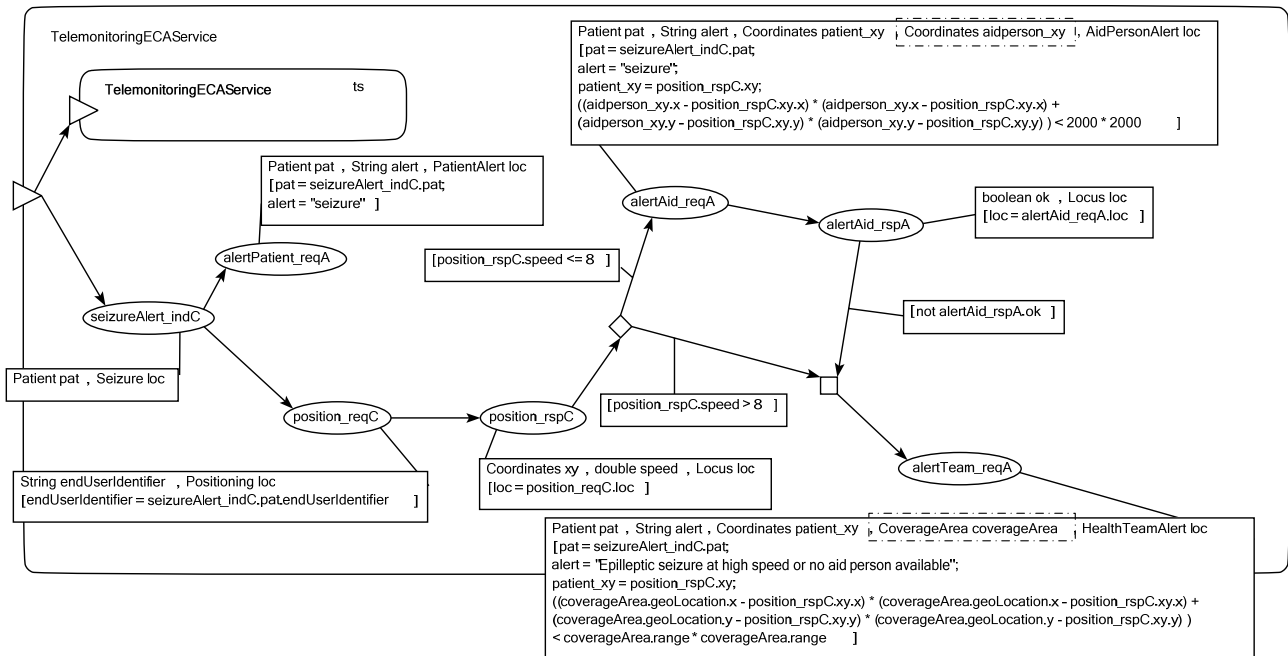

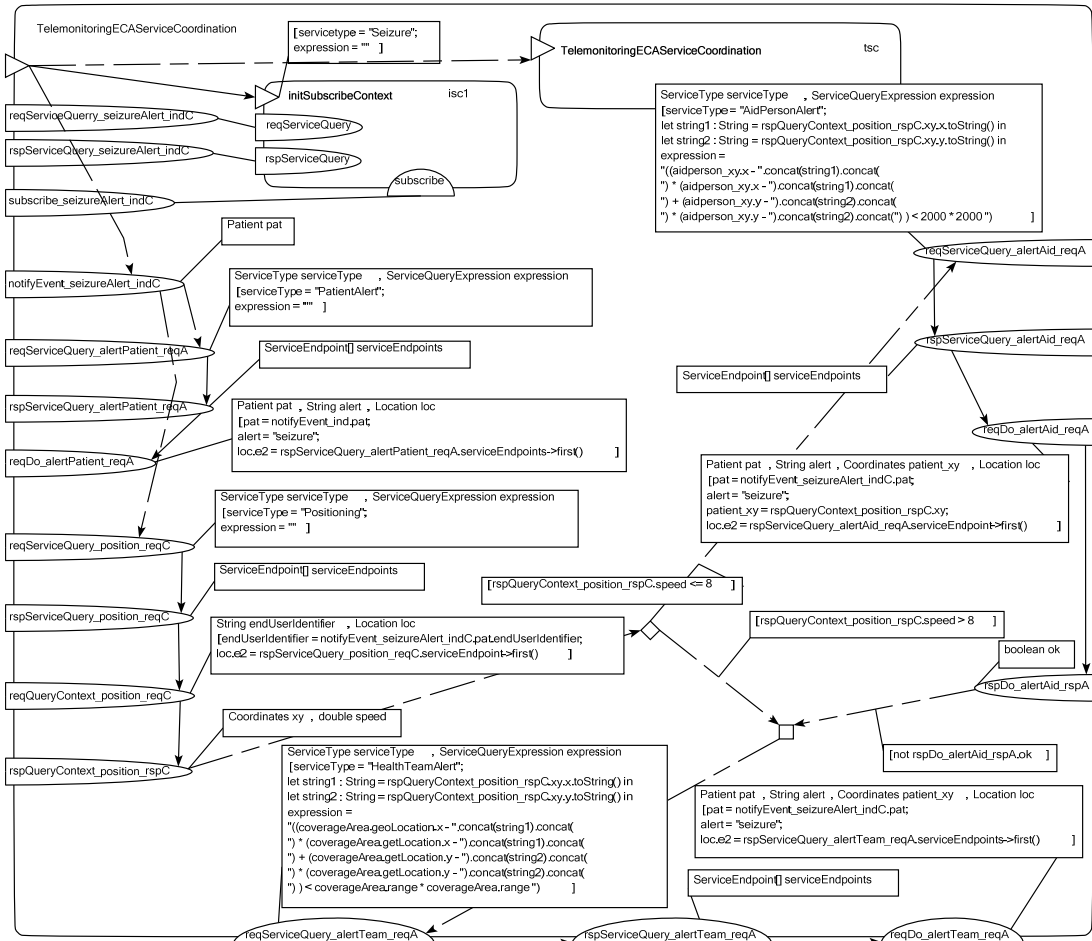**Figure 11 The Telemonitoring service specification (M1) [7]**

Figure 12 Generated coordination component for Telemonitoring service (M2)

7.4. Traceability

In its simplest form, traceability manifests itself in cross-tables in which elements of a design are checked against the requirements that they satisfy. We use cross-tables to illustrate the relations between requirements and models at different levels.

Table 3 presents a traceability cross-table at the level of consecutive models, revealing the partitioning of application requirements. The table shows that the outcome of assessment activities is that M1 satisfies all level-1 requirements, while model M2 satisfies those requirements as well as the level-2 requirements, and so on. It is possible to be more specific: in this case there would be an entire table for M1, with a column for each model element present in M1.

Evidence for all check marks should be provided through assessment activities. For example, marks in the column corresponding to M1 can be justified by directly inspecting M1 against the requirements specification; by simulating M1, etc. Marks in the column corresponding to

M2 can be justified by simulation, by model checking behavioural aspects implied by AR1–AR10, etc. Marks in the column corresponding to a_{TSA} can be justified by considering the characteristics of the context and action services implied by the particular choice of a_{TSA}.

	M1	a_{TSA}	M2	P3	M3
AR1	✓		✓		✓
AR2	✓		✓		✓
AR3	✓		✓		✓
AR4	✓		✓		✓
AR5	✓		✓		✓
AR6	✓		✓		✓
AR7		✓	✓		✓
AR8		✓	✓		✓
AR9		✓	✓		✓
AR10		✓	✓		✓
AR11				✓	✓
AR12				✓	✓

Table 3 Traceability cross-table

Because certain platform characteristics are considered relevant for assessment of application-specific require-

ments, we have also included a row for the platform P3 in this cross-table. Assessing satisfaction of AR11 should be justified by analysing specifications of the target platform P3, in this case the specifications of battery consumption for PDAs or mobile phones, and characteristics of M3 that may influence battery consumption. AR12 refers to subscription costs and require both the amount of traffic generated by M3 and characteristics of the platform used (efficiency of encoding schemes used, cost per traffic generated, etc.). Note that application-independent requirements AIR12–AIR16 may affect P3 and thereby influence satisfaction of AR11 and AR12. Other marks in row M3 can be justified by testing it against test cases for AR1–AR10.

As discussed in section 5.2, these assessment activities can be simplified by employing conformant transformations. We illustrate this with a transformation TSA, which has been designed such that it is conformant under the following assumptions [3]: (i) the service trader is always able to produce a service offer for a service query, (ii) context sources always reply to context query requests, and (iii) action services always reply to action invocation requests (in case action invocation request and action invocation response is used in a pattern). Assumption (i) can be guaranteed by availability of service offers in the service trader that correspond to actions and context queries and events in the service specification level (according to transformation arguments a_{TSA}). Assumptions (ii) and (iii) constrain the design of context sources and action services. These assumptions are necessary to integrate the interactions in the target design into actions and then apply the conformance assessment method described in [21].

Transformation TSA simplifies traceability, which is illustrated by Table 4. Assuming TSA and TSB conformant, all marks in the M2 column and all marks for AR1–AR10 in the M3 column are implied, which is indicated by square brackets. Assessment activities to check them have become redundant, which greatly diminishes the assessment effort needed. In fact, M2 can even be considered a "black-box" by an application designer.

	M1	a_{TSA}	M2	P3	M3
AR1	✓		[✓] TSA		[✓]TSB
AR2	✓		[✓] TSA		[✓]TSB
AR3	✓		[✓] TSA		[✓]TSB
AR4	✓		[✓] TSA		[✓]TSB
AR5	✓		[✓] TSA		[✓]TSB
AR6	✓		[✓] TSA		[✓]TSB
AR7		✓	[✓] a_{TSA}		[✓]TSB
AR8		✓	[✓] a_{TSA}		[✓]TSB
AR9		✓	[✓] a_{TSA}		[✓]TSB
AR10		✓	[✓] a_{TSA}		[✓]TSB
AR11				✓	✓
AR12				✓	✓

Table 4 Traceability cross-table with conformant transformations

8. Discussion

We do not account in this paper for explicit requirements on the design process itself, such as, e.g., cost, delivery schedules, validation and verification criteria (assessment criteria). This is in line with the IEEE Recommended Practice for Software Requirements Specifications (SRS) that defines that "SRS should address the software product, not the process of producing the software product." These should be captured in project requirements which "represent an understanding between the customer and the supplier about contractual matters pertaining to production of software and thus should not be included in the SRS." [17]. However, we do capture requirements on the model-driven design trajectory (the so-called application-independent requirements). These are maintained separately from application-specific requirements, and are relevant only to "suppliers" and their internal organization and are typically not visible to "customers."

We acknowledge that the quality of assessment depends eventually on the quality of a requirements specification. Different characteristics of a "good" specification are defined in [17] including correctness, lack of ambiguity, completeness, consistency, etc. Guidelines for obtaining these qualities are beyond the scope of this paper. However, we believe that application-independent requirements may serve to identify unstated requirements called "necessary design constraints" or "software system attributes" in [17], thus contributing to the completeness of requirements specifications. In our example, the application-independent requirements AIR7 ("It should be possible to add reusable (context and action) services at service runtime") and AIR11 ("Service components should specify required services, which can be discovered and bound at runtime") reveal unstated maintainability requirements for the telemonitoring service.

The simplification of assessment activities results from the way in which requirements are partitioned and addressed at different levels of abstraction. For sets of requirements that cannot be partitioned and that must be partially satisfied at multiple abstraction levels, simplification of assessment by conformance is limited.

We have considered an application model at a particular abstraction level as a unit of design for requirements traceability. Nevertheless, we do not exclude more "fine-grained" traceability strategies which identify parts of application models. Clustering these parts of application models into levels that are related by conformance is sufficient to apply the technique discussed in this paper.

Finally, while we have discussed the potential benefits of conformant transformations, we would like to emphasize that evidence for transformation conformance may be costly to produce. One should therefore consider the pay-

off in terms of assessment activities, depending on the reuse of transformation specifications.

9. Related work

In the area of Requirements Engineering, the standard general introduction of the requirements traceability problem has been provided by Gotel and Finkelstein [15]. The Ph.D. thesis of Gotel provides extensive discussion of requirements traceability, including a number of definitions (see pages 71-72 of [16]).

It has been recognized that requirements tracing is a laborious task and that any assistance in maintaining the interdependencies between requirements and other design artefacts is highly welcome. Egyed [11] presents an approach in which dependencies are discovered automatically from data generated by executing a minimal set of scenarios. This approach requires that an executable version of the system is available to execute these scenarios. In our approach, however, traceability is not dependent on an executable system; therefore, traceability is already possible when the design process has not yet resulted in an executable prototype.

Ramesh and Jarke [24] present a reference model for requirements traceability that they derived from an empirical study. Their reference model comprises a number of possible relations that can be traced between design artefacts and requirements. For different stakeholders (and different ambition levels with respect to traceability), a different subset of those relations can be chosen. In principle, our conformance-based approach is transparent with respect to the choice of this subset. An interesting question for future research is whether subsets can be identified that are particularly suitable for a model-driven design approach.

In the Reference Model of Open Distributed Processing (RM-ODP) [18], the term "conformance" is used as relation between a "specification" and an "implementation". In this paper, we have used the term as relation between two application models. Considering our stance on the distinction between an application model and an application realization (see section 3), our view on conformance does not conflict RM-ODP's approach to conformance. RM-ODP uses the term "conformance testing", and we use the more general term "conformance assessment" to include other forms of assessment activities.

In the area of formal methods, notions of transformation conformance have also been defined. Nevertheless, approaches based on formal methods rely on formal proofs as evidence for transformation conformance (see, e.g., "correct architectural refinement" in [20], and "correctness preserving transformations" in [9, 14]). We believe that formal proofs may not be required in many practical cases. Therefore, we have proposed definitions

for conformance and requirements satisfaction that are independent of proofs of conformance or formalization of requirements. Furthermore, we are neutral with respect to the techniques the designer may chose to trust for assessment.

We have intentionally considered specific techniques to assess requirements satisfaction and conformance outside the scope of this paper. Instead, we have focused on how to manage the relations between models and requirements assuming the existence of some conformance assessment technique. Examples of such techniques are the "conformance rules" for "behaviour refinement" discussed in [21] (and used in our example), "refinement relations" discussed in [10] or "conformant transformations for interaction refinement" presented in [4].

10. Conclusions

We believe that a mature discipline for model-driven design must provide techniques to account for how requirements relate to the various artefacts produced during the design process. In this paper we have proposed a methodological framework that addresses this issue. Our framework can be seen as basis for requirements traceability, but also serves to reveal the intricate relationship between requirements, application models and realizations, model transformation specifications, transformation arguments, metamodels, model libraries and platforms.

In our view it is important for both application users and application designers to be able to produce evidence for satisfaction of requirements. This is realised through assessment activities. We have argued that some of these assessment activities can be deemed redundant under the assumption that conformant transformation specifications are used in the design process. Thus, we have concluded that conformance between models not only simplifies requirements tracing but also has the potential of reducing the amount of necessary assessment activities.

In our future work, we intend to investigate both the specification of conformance relations and model transformations in the same transformation specification framework. More precisely we plan to focus on techniques and tools for capturing, enforcing and assessing conformance between models; and assessing whether transformation specifications respect conformance. This may be feasible by regarding transformation and conformance as relations ([1, 23]) (as suggested in [4]).

Future work could also investigate traceability of requirements in face of changes in requirements specifications, which may be trigged due to changing application requirements and due to improved understanding of requirements in an iterative design process.

Acknowledgements

References

[1] D. Akehurst, S. Kent, and O. Patrascoiu, "A relational approach to defining and implementing transformations between metamodels", *Software and Systems Modeling*, vol. 2. no. 4, Springer-Verlag, 2003, pp. 215-239.

[2] J.P.A. Almeida, *Model-Driven Design of Distributed Applications*, CTIT Ph.D.-Thesis Series, No. 06-85, Telematica Instituut Fundamental Research Series, No. 018, 2006.

[3] J.P.A. Almeida, H. Jonkers, M.E. Iacob, and D. Quartel, *Platform-Independent Modelling of Service Infrastructure Components: Towards the A-MUSE Abstract Platform*, Freeband A-MUSE/D1.6, Telematica Instituut, The Netherlands, 2005; https://doc.telin.nl/dscgi/ds.py/Get/File-59319

[4] J.P.A. Almeida, R. Dijkman, L. Ferreira Pires, D. Quartel, and M. van Sinderen, "Model Driven Design, Refinement and Transformation of Abstract Interactions", *Int'l J. Cooperative Information Systems (IJCIS)*, World Scientific, to appear.

[5] J.P.A. Almeida, R. Dijkman, M. van Sinderen and L. Ferreira Pires, "On the Notion of Abstract Platform in MDA Development," *Proc. 8th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE CS Press, Sept. 2004, pp. 253-263.

[6] J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires and D. Quartel, "A systematic approach to platform-independent design based on the service concept," *Proc. 7th IEEE Int'l Conf. on Enterprise Distributed Object Computing (EDOC 2003)*, IEEE CS Press, Sept. 2003, pp. 112-134.

[7] J.P.A. Almeida, M.-E. Iacob, H. Jonkers, and D. Quartel, "Model-Driven Development of Context-Aware Services", *Distributed Applications and Interoperable Systems (DAIS 2006), 6th IFIP International Conference,* LNCS, vol. 4025, Springer, 2006, pp 213-227.

[8] G. Arango, "Domain Analysis: from Art Form to Engineering Discipline," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 3, ACM Press, May 1989, pp. 152-159.

[9] T. Bolognesi, J. van de Lagemaat, and C. Vissers, eds., *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, 1995.

[10] R.M. Dijkman, *Consistency in Multi-Viewpoint Architectural Design*, CTIT Ph.D.-Thesis Series, No. 06-80, Telematica Instituut Fundamental Research Series, No. 017, 2006.

[11] A.F. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis", *IEEE Transactions on Software Engineering*, 2003. vol. 29, no. 2, pp. 116-132.

[12] D. Firesmith, "Specifying Good Requirements", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 77-87; http://www.jot.fm/issues/issue_2003_07/column7

[13] A. Gavras, M. Belaunde, L. Ferreira Pires, J.P.A. Almeida, "Towards an MDA-based Development Methodology for Distributed Applications", *Software Architecture: First European Workshop (EWSA2004)*, LNCS, vol. 3047, Springer, 2004, pp. 230-240.

[14] J.P. Gibson, T.F. Dowling, and B.A. Malloy, "The Application of Correctness Preserving Transformations to Software Maintenance", *Proc. 16th IEEE International Conference on Software Maintenance (ICSM'00)*, IEEE CS Press, 2000, pp. 108-119.

[15] O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. First Int'l Conf. Requirements Eng.*, 1994, pp. 94-101.

[16] O. Gotel, *Contribution Structures for Requirements Traceability*. Ph.D. Thesis, London, England: Imperial College, Department of Computing, 1995.

[17] IEEE, *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830-1998, 1998.

[18] ISO / ITU-T, *Open Distributed Processing - Reference Model - Part 2: Foundations*, International Standard ISO/IEC 10746-2, ITU-T Recommendation X.902, 1995.

[19] H. Kremer, *Protocol Implementation: Bridging the gap between Architecture and Realization*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, Oct. 1995.

[20] M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, 21(4), IEEE CS Press, April 2005.

[21] D. Quartel, L. Ferreira Pires, and M. van Sinderen, "On Architectural Support for Behaviour Refinement in Distributed Systems Design," *Journal of Integrated Design and Process Science*, vol. 6, no. 1, Society for Design and Process Science, 2002.

[22] Object Management Group, *MDA-Guide, V1.0.1*, omg/03-06-01, June 2003.

[23] Object Management Group, *MOF QVT Final Adopted Specification*, ptc/05-11-01, Nov. 2005.

[24] B. Ramesh, M. Jarke, "Toward reference models for requirements traceability", *IEEE Transactions on Software Engineering*, vol. 27, no. 1, Jan. 2001, pp. 58-93.

[25] J. Schot, *The role of Architectural Semantics in the formal approach of Distributed Systems design*, Ph.D. thesis, University of Twente, Enschede, The Netherlands, Feb. 1992.

[26] *The Interaction Systems Design Language (ISDL)*; http://isdl.ctit.utwente.nl/

[27] S. Wright, "Requirements Traceability - What? Why? and How?", *Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium*, Computing and Control Division, Professional Group C1 (Software Engineering), U.K., Digest Number: 1991/180, December 2, 1991, pp. 1/1-1/2.