An Approach to Relate Viewpoints and Modeling Languages

Remco M. Dijkman Dick A.C. Quartel Luís Ferreira Pires Marten J. van Sinderen
University of Twente
Centre for Telematics and
Information Technology
Enschede, The Netherlands
{dijkman|quartel|pires|sinderen}@cs.utwente.nl

Abstract

The architectural design of distributed enterprise applications from the viewpoints of different stakeholders has been proposed for some time, for example, as part of RM-ODP and IEEE 1471, and seems now-a-days to gain acceptance in practice. However, much work remains to be done on the relationships between different viewpoints. Failing to relate viewpoints may lead to a collection of viewpoint models that is inconsistent and may, therefore, lead to an incorrect implementation. This paper defines an approach that helps designers to relate different viewpoints to each other. Thereby, it helps to enforce the consistency of the overall design. The results of this paper are expected to be particularly interesting for Model Driven Architecture (MDA) projects, since the proposed approach can be used for the explicit definition of the models and relationships between models in an MDA trajectory.

1. Introduction

Now-a-days many researchers and designers tend to agree that the design of sophisticated and software-intensive distributed applications has to be performed according to different viewpoints. This allows the designers to manage the complexity of the development process [7,5]. In particular in the scope of the Model-Driven Architecture (MDA) development approach, designers are required to produce collections of models from different viewpoints, such as business domain, business process, platform-independent and platform-specific models [14].

Viewpoints give some guidance on the models to be produced during a design process and the objectives of these models. One can also prescribe the languages to be used in order to represent each particular model. However, this plethora of models all refer to the same system, and as such they should be kept aligned and consistent with respect to each other.

This paper introduces and motivates an approach to keep models from different viewpoints aligned and consistent. This approach aims at facilitating the development process in a number of ways. It helps to improve the communication between different stakeholders by relating the terminology and concepts they use. It allows designers to use the same or different modeling languages to represent models from different or the same viewpoint, by clearly defining mappings between the concepts that underlie these languages and viewpoints. Based upon these mappings, techniques can be defined to analyze and enforce various types of relations between different views and models of the same system (e.g., equivalence and refinement relations). And finally, it facilitates the creation of tool support to (partially) automate the application of these techniques.

The results of this paper are expected to be particularly interesting for MDA projects, since the success of these projects very much depends on the designers' ability to document the viewpoints and models to be produced, and the relationships between models.

This paper is further structured as follows. Section 2 discusses architectural design and explains that a system-under-design has a triple existence: in the real world, in the conceptual world and in the symbolic world. Though there is obviously only one real world system, there can be many conceptual world *views* on the system and many symbolic world *models*. The architectural design activity must ensure that these views and models can be related to each other, either directly or indirectly, and to the real world system. Section 3 discusses viewpoints and modeling languages as important elements of architectural design that facilitate the conception of views and the definition of models, respectively. These elements are also instrumental to relate different views and their corresponding models in an architectural design process. Viewpoints, modeling languages and their relationships are defined using *meta-models*. Section 4 discusses several approaches to relating views and models, and elaborates on one approach. Section 5 illustrates this approach by relating different models of an example application. Section 6 summarizes the main contributions of this paper and gives an outlook on future work.

2. Architectural design

Architectural design is the process of defining the desired properties of a (prospective) software system, such as its structure and behavior, while considering the role of this system in its environment. Many different *stakeholders* may be involved in the architectural design of a software system. Each of these stakeholders focuses on certain *concerns* and considers these concerns at a certain *level of detail*.

To assess whether his concerns are addressed in a satisfactory way, each stakeholder forms a mental image of what properties the system should have and how it should interact with its environment. We call a mental image of a stakeholder that addresses certain concerns of the system at a certain level of detail a *view* on the system (and the environment). Figure 1 illustrates the use of three views on some system, by showing three stakeholders that focus on different concerns. Each of these stakeholders forms his own mental image of the system.

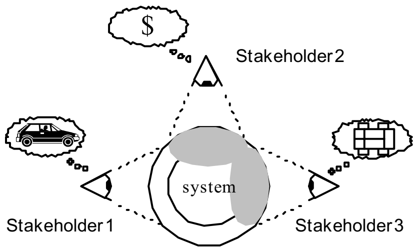


Figure 1. Stakeholders have different views on a system

Since it is hard to discuss and share designs in terms of mental images that only exist in the minds of the stakeholders, these mental images are made concrete in the form of *models*. For example, models may be expressed as linear text or as a composition of graphical symbols. Hence, architectural design takes place in three related 'worlds': the *real world*, where the real system and its environment exist, the *conceptual world*, which is the conception of the real world in our mind, and the *symbolic world*, which is the concrete representation of the conceptual world on some medium (e.g., paper or a computer screen).

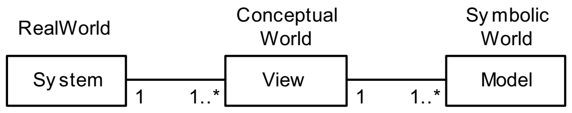


Figure 2. The triplet of architectural design

Figure 2 shows the three related worlds of architectural design. An architectural design process may produce many different views, being different conceptions of the same system that consider different design concerns. In addition, each view may be represented by different models that use different symbolisms.

Since different views and their associated models refer to the same system, views as well as models are related in one way or another. This is illustrated in figure 3. We distinguish between two basic types of view relations[1].

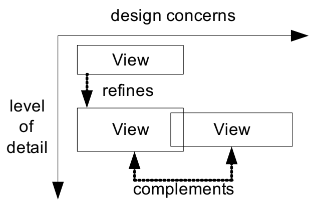


Figure 3. Basic view relations

Refinement relation. Two views may be related because they consider the same design concerns at different levels of detail. For example, an external system view may consider the externally observable behavior that is provided by the system as a whole to its environment, while another view considers the behavior of internal system components. These views are related because the internal view refines the external view, i.e., adds design detail, by providing an internal decomposition of the system that defines how the external view can be implemented.

Complement relation. Two views may complement each other by considering complementary concerns. For example, one view may consider the structuring of a system in terms of parts and how they are interconnected, while another view considers the behavior of each part. These views are complementary in the sense that the structure view merely identifies the parts, whereas the behavior view considers their behavior.

In general, it may be difficult, if not impossible, to separate concerns such that they are fully complementary, in the sense that they have no system properties in common. Therefore, views are likely to consider partly overlapping concerns. For example, the structure and behavior view are not fully complementary. Because the behavior view should conceive the individual behavior of each part identified in the structural view, both views should consider the same system decomposition.

Section 4 elaborates on how the correctness and consistency of both view relation types can be enforced.

3. Viewpoints and modeling languages

To use views and models in the design process, we must be able to construct them. To do this, we use viewpoints and modeling languages respectively. Figure 4

[1] For brevity, we will use the term 'view relations' to denote both relations between views and between their associated models.

illustrates the role of viewpoints and modeling languages in architectural design.

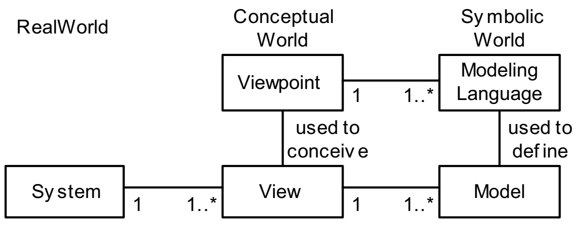


Figure 4. Elements of architectural design

This section explains how the viewpoints and modeling languages themselves can be defined. It also explains how modeling languages can be used to represent viewpoints.

3.1. Viewpoints

A viewpoint defines the means to conceive views at a certain level of detail and regarding certain design concerns. To this end, a viewpoint consists of a set of so-called design concepts, and rules for composing these design concepts. A design *concept* models some common and essential properties of a system. The design concepts are the means to construct a view.

For example, consider a business process viewpoint. Relevant properties of a business process are the tasks to be performed, and how these tasks are related. Therefore, examples of candidate design concepts are 'task', 'sequence', and 'or-split' (choice). In addition, an example of a rule for composing these concepts would be that an instance of the 'sequence' concept is related to two instances of the 'task' concept, defining that one task happens before the other.

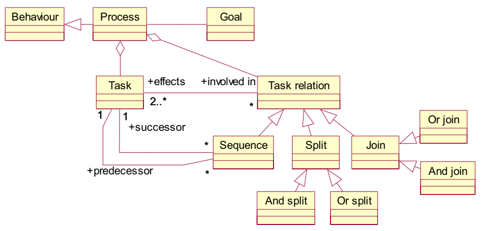


Figure 5. Business process design meta-model

To define a viewpoint, we should agree on the meta-concepts used to construct a viewpoint. Typically, we use the *meta*-concepts 'concept', 'attribute of a concept', and 'relation between concepts'. In order to represent viewpoint definitions, we use UML class diagrams (compliant to the OMG standard for meta-concepts, the Meta Object Facility (MOF) [13]). A concept, concept attribute and concept relation is represented as a UML class, a UML class attribute, and a UML class association, respectively. We call the class diagram representing a viewpoint definition, a *design meta-model*.

Figure 5 illustrates the design meta-model of the business process viewpoint discussed before. The 'and-split' and 'or-split' task relations represent that all involved tasks are executed 'in parallel', or a 'choice' is made between one of them, respectively. The 'and-join' and 'or-join' task relations represent that 'all' or 'at least one' of the involved tasks must have been executed.

A design meta-model defines the *abstract syntax* for constructing a view. The meta-model itself, however, does not define the semantics of the represented viewpoint concepts. The semantics of a design meta-model should define what real-world system properties are modeled by each viewpoint concept and by each relationship between viewpoint concepts. In other words, the semantics defines the interpretation of a meta-model, and the views constructed from it, in terms of real-world system properties. We call this the *architectural semantics* of a design meta-model. For example, the architectural semantics of the concept 'task' is: the smallest unit of work that is meaningful to an actor. Typically, architectural semantics is described informally, as an annotation to the design meta-model.

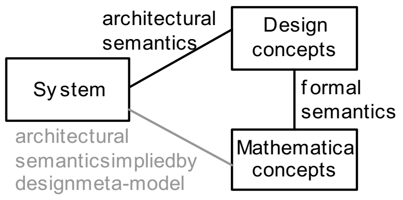


Figure 6. Architectural and formal semantics

In addition, one may define a formal semantics, which defines a mapping from the system properties represented by the design concepts in the design meta-model onto the mathematical properties represented by the mathematical concepts in a mathematical model. This is often used to add precision and facilitate the development of analysis techniques and supporting tools. Figure 6 illustrates the notions of architectural and formal semantics.

3.2. Modeling languages

A modeling language defines the means to construct models. These means consist of language concepts, which define what can be modeled, and notational elements to represent (express) the language concepts. For example, UML statecharts define the language concept 'action', which is represented by the notational element 'rounded rectangle'.

A language concept models some system properties, similar to a design concept. For example, the language concept 'action' represents some unit of activity that can be executed by a system. Therefore, language concepts are to modeling languages what design concepts are to viewpoints. This also means a modeling language implicitly defines its own viewpoint.

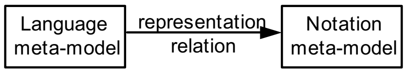


Figure 7. Elements of a modeling language

Because of the separation between language concepts and their notation, a modeling language can be defined by two related meta-models: a *language meta-model*, which defines the language concepts and their relationships (similar to a design meta-model), and a *notation meta-model*, which defines the notational elements and their relationships. A notation meta-model is composed from meta-concepts like 'notational element', 'attribute of notational element', and 'relation between notational elements'. Figure 7 depicts both meta-models.

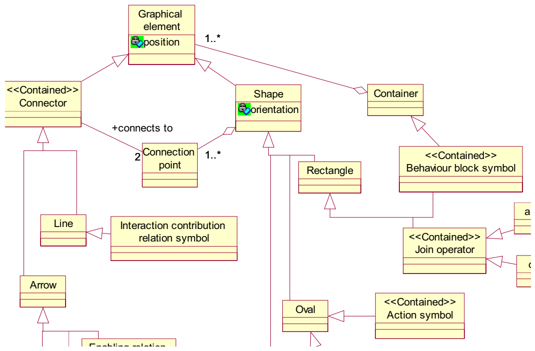


Figure 8. Example of (part of) a notation meta-model

The representation relation between the language and notation meta-model defines how each language concept is represented by one or a composition of notational elements. In this way, the modeling language provides a *concrete syntax* for the *abstract syntax* defined by the language meta-model. An abstract syntax may be associated with more than one concrete syntax. For example, it is common for languages to define both a graphical and a textual concrete syntax for the same abstract syntax.

The benefit of a distinct notation meta-model is to clearly separate between conceptual aspects and notational aspects of a modeling language. However, quite often the notation meta-model is left implicit, because a one-to-one mapping exists between language concepts and notational elements. Figure 8 depicts part of the notation meta-model of the ISDL modeling language introduced in section 4.

3.3. Representing viewpoints

In order to use a modeling language to represent (the views according to) some viewpoint, the relationship between the language meta-model of the modeling language and the design meta-model of the viewpoint has to be defined. This relationship should clearly define how (compositions of) design concepts from the viewpoint are represented by (compositions of) the language concepts underlying the modeling language.

As an example, we consider how the example business process viewpoint of figure 5 can be represented by UML activity diagrams. For this purpose, we first present the language meta-model that defines the abstract syntax of UML activity diagrams. For the current discussion, the meta-model has been simplified. Figure 9 depicts the UML activity diagram language meta-model.

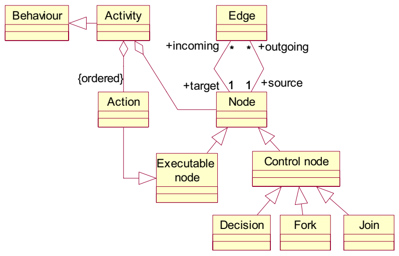


Figure 9. UML activity diagram language meta-model

Subsequently, we define the relation between the design meta-model of figure 5 and the language meta-model of figure 9. This relation can be defined by means of associations between the elements of both meta-models (possibly extended with OCL constraints). Figure 10 illustrates this for the representation of the business process concepts 'task' and 'or-split' in terms of the activity diagram concepts 'action', 'edge' and 'decision'.

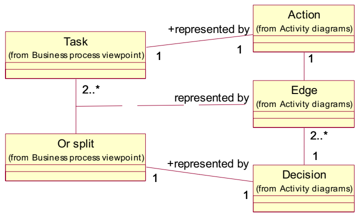


Figure 10. Representing the business process viewpoint with UML activity diagrams

The distinction between viewpoint definition and modeling language definition is not common practice. For example, design projects often choose a set of popular modeling languages, like UML diagrams, and leave the definition of viewpoints implicit. Alternatively, designers and researchers may define Domain Specific Languages (DSL). A DSL is a concrete syntax developed for a particular application domain (that can be seen as a viewpoint or a set of viewpoints). Hence, a DSL does not define its own concepts, but instead uses the concepts from the domain that it represents.

The benefit of making a distinction between viewpoint definition and modeling language definition is the clear separation of concerns. Using the distinction, viewpoint and modeling language can be defined separately (by different expertise groups). Also, the same modeling language and its tool-support can be re-used to represent many different viewpoints.

An example of a standard that does make a distinction between viewpoint definition and modeling language definition is the Reference Model for Open Distributed Processing (RM-ODP) [9]. The RM-ODP standard itself only defines viewpoints, while other standards and papers (e.g. [12,1,2]) define modeling languages that can be used to represent the RM-ODP viewpoints, or define how existing modeling languages can be used to represent the RM-ODP viewpoints.

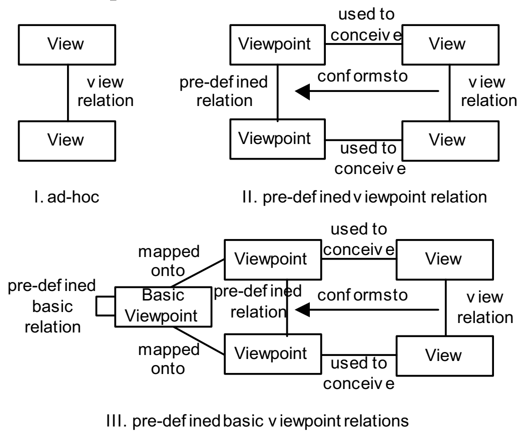


Figure 11. Approaches to relate views

4. Relating views and relating models

The selection and documentation of viewpoints and modeling languages helps to manage the views and models that are produced in a design process, but is not sufficient. Also the relationships between views as well as models should be clearly defined. We present three approaches to define and enforce view relations[2], and elaborate on one of them. Finally, we present the basic modeling language ISDL, which is used to illustrate the latter approach in section 5.

4.1. Approaches

Figure 11 illustrates three distinct approaches to relate views.

Approach I relates views directly in an ad-hoc manner. This means that the correspondences and differences between the system properties of these views are described specifically for a single architectural design process. For example, consider a functional view, which conceives the system functions and their relations, and a performance view, which conceives the performance properties of the system functions that effect system performance. In this case one may want to relate both views directly in order to assess whether the conceived functions correspond, and whether the performance view considers all system functions identified in the functional view that may impact system performance significantly.

Approach II relates views indirectly via pre-defined relations between the corresponding viewpoints. This means that when some view is conceived, this view must conform to the pre-defined relations in which the corresponding viewpoint is involved. For example, consider a structure viewpoint, which conceives system parts and how they are interconnected, and a behavior viewpoint, which conceives the behavior of system parts. A pre-defined relation between these viewpoints could be that for each system part conceived in a structure view an associated behavior is conceived in a behavior view, and that interactions in the behavior view can only take place between system parts that are interconnected in the structure view. Other examples are RM-ODP, which pre-defines the relations between its viewpoints in [9, part 3 clause 10], and the MDA development approach, which proposes the definition of mappings between design meta-models. Such a mapping can be seen as an example of a pre-defined viewpoint relation.

Approach III assumes that a basic viewpoint exists that defines re-usable basic relations. These basic relations may be used to pre-define relations between viewpoints. This approach requires us to define mappings from the viewpoints to the basic viewpoint and to define the relations between the viewpoints in terms of the basic relations. For example, [3] defines a conformance relation between a business process and the composition of component behaviors that implements it. It does this by defining a mapping from the business process viewpoint and the component viewpoint onto a basic viewpoint and by defining the conformance relation between the business process and the component viewpoint as the conformance relation of the basic viewpoint. To prevent that the designer has to create a basic view for each view that exists in a design, the mappings from the viewpoints to the basic viewpoint should be supported by automated transformations.

To be able to apply the third approach, a basic viewpoint must be found that matches the concepts used in the viewpoints one wants to relate. Preferably, such a basic viewpoint should define generic and elementary concepts, such that the concepts of other viewpoints can be mapped onto specializations or compositions of the basic viewpoint concepts. Examples of approaches that define a basic viewpoint, but only for modeling

[2] by view relations we also denote the corresponding relations between the associated models. In fact, in a design process a view relation is often considered by its corresponding model relation.

languages, are: UML, which defines a set of core concepts, and the 3C proposal for UML 2.0 [11], which defines an even more basic set of core concepts. An example of an approach that defines a basic viewpoint, and applies it to both UML and RM-ODP, is the Systemic Enterprise Architecture Methodology (SEAM) [21]. Its basic viewpoint is defined in [10].

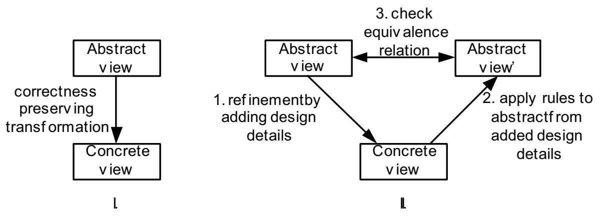
Advantages of the third approach over the second approach are that a smaller number of basic relations have to be pre-defined (that can be re-used) and that one can reason about these relations within a small, consistently defined set of concepts.

4.2. Enforcing view relations

Two basic types of view relations have been identified in section 2: refinement and complement. Here we will discuss some techniques to enforce these relations in a design process. In principle, these techniques can be applied in combination with any of the approaches discussed in section 4.1. However, since these techniques are most often applied within the scope of a single modeling language, they are most likely to be used in combination with the third approach.

4.2.1. Refinement relation. In an architectural design process, we want to be able to assess the correctness of a refinement relation, by checking whether the more detailed (concrete) view implements the less detailed (abstract) view. Since the concrete view implements the abstract view, and should conform to all properties conceived in the abstract view, a refinement relation is also called implementation relation or conformance relation.

Figure 12 illustrates two basic techniques to obtain correct refinements (implementations).



Abstract view
correctness preserving transformation
Concrete view
I.
3. check equivalence relation
Abstract view
Abstract view'
1. refinement by adding design details
2. apply rules to abstract from added design details
Concrete view
II.

Figure 12. Refinement techniques

Technique I applies so-called correctness preserving transformations, which are built from pre-defined mappings from abstract concepts applied in the abstract view onto concrete concepts applied in the concrete view. Because the mappings are defined at conceptual level their correctness have to be proven only once, and can subsequently be instantiated in any view. This technique is proposed by the MDA development approach. For further readings and examples of this technique we refer to [6,8,17].

Technique II distinguishes three steps to assess the correctness of a refinement relation. Step 1 is the refinement step, in which certain design details are added to the abstract view. The technique does not prescribe how this step must be performed. This is left to the creativity of the designer. Step 2 applies pre-defined rules to abstract again from these design details. And step 3 checks whether the resulting abstract view is equivalent to the original abstract view. An example of the application of this technique can be found in section 5. For further readings and examples we refer to [16,20].

The first technique is generally easier and faster to apply, since implementation decisions have been pre-defined and proven to be correct. However, they are often limited by these implementation decisions. The second technique is more generic in the sense that it allows any design decision to be made in the refinement step, while the abstraction rules can be applied to any concrete view. This technique is motivated by the observation that during a refinement step many alternative implementations of an abstract view are possible, but when one abstracts from the alternative design details that have been added to the abstract view, the abstraction of all these implementations is unique.

4.2.2. Complement relation. In an architectural design process, we want to be able to assess the consistency of a complement relation by checking whether the overlap of both views is equivalent. This means that one first has to delimit both views to the part that they have in common, and subsequently assess whether the system properties conceived for this part in both views are equivalent.

For example, consider the example discussed under approach II of section 4.1. Both the structure viewpoint and the component view should consider equivalent system decompositions. This equivalence could be defined in terms of rules like 'each behavior is assigned to a system part' and 'an interaction between behaviors must happen at an interconnection point connecting the system parts to which the behaviors are assigned'. For more examples we refer to section 5.

Much theory has been developed about equivalence relations in the scope of particular modeling languages [19]. Approach III of section 4.1 allows one to re-use this theory for assessing the consistency between overlapping views, by defining a mapping from (the part that is common for) their corresponding viewpoints onto these modeling languages.

4.2.3. Combination of relations. In general, two views may consider partly complementary design concerns at different levels of detail. In this case, we want to be able to assess the correctness of the refinement relation only between the overlap of the abstract view and the concrete view. This means that one first has to delimit both views to the part they have in common, and subsequently apply one of the refinement techniques discussed above.

For example, consider the RM-ODP enterprise and computational viewpoints. A computational view may conceive the implementation of some application, while the enterprise view conceives the externally observable application behavior and its embedding in the enterprise. In this case, the observable application behavior has to be isolated from the enterprise view in order to assess the refinement relation between the observable application behavior and its implementation as conceived by the computational view [4].

4.2.4. Concluding remarks. We deliberately do not define the precise design criteria to determine when a concrete view is a correct refinement of an abstract view or when the overlapping concerns conceived by two views are equivalent. These criteria depend on the specific design objectives and the architectural semantics of the views. Here, we show how existing work on the validation of conformance and equivalence relations can be used to relate views.

4.3. ISDL: a basic modeling language

We introduce the Interaction System Design Language (ISDL) as a basic modeling language. We use the ISDL as a basic viewpoint as well. In this role it is used in section 5 to illustrate the third approach discussed in section 4.1 to relate different views and models in a design process. In order to show that ISDL can be used as a basic modeling language, and as preparation to section 5, we show how UML activity diagrams and UML statechart diagrams can be mapped onto ISDL.

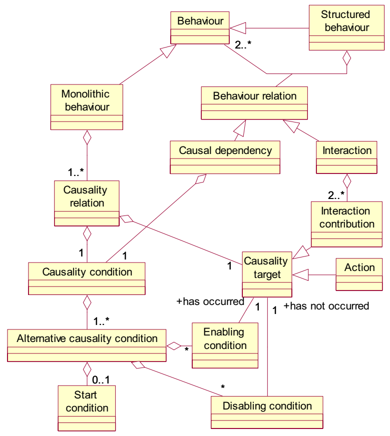


Figure 13. ISDL language meta-model

Figure 13 depicts part of the language meta-model underlying ISDL. We do not explain the notation meta-model (see figure 8) here due to space limitations, instead we introduce the ISDL notation in an ad-hoc manner.

An ISDL behavior is either monolithic or structured, where a structured behavior consists of sub-behaviors. A monolithic behavior consists of causality relations that each define for a single causality target the condition under which this target may occur. A causality target may be an action, which represents the completion of a unit of activity performed by a single monolithic behavior, or an interaction contribution, which represents the participation of a monolithic behavior in some joint activity involving multiple monolithic behaviors.

We distinguish between three basic causality conditions for the occurrence of some causality target *a*:

- enabling condition $b \rightarrow a$: *b* must occur (happen) before *a*. Graphically represented as $\textcircled{b}\!\rightarrow\!\textcircled{a}$;
- disabling condition $\neg b \rightarrow a$: *b* must *not* occur before, nor simultaneously with *a*. Graphically represented as $\textcircled{b}\!\nrightarrow\!\textcircled{a}$;
- start condition $\surd \rightarrow$ a: *a* is always enabled. Graphically represented as $\rightarrow\!\textcircled{a}$.

Basic causality conditions are composed into alternative causality conditions. Each of the basic conditions must be satisfied for an alternative condition to be satisfied (logical *and*). In turn, alternative causality conditions are composed into causality conditions. At least one alternative condition must be satisfied for the causality condition to be satisfied (logical *or*).

Figure 15 (ii) and (iii) depict an example of a monolithic behavior consisting of four actions. The causality condition of each action consists of a single alternative causality condition. Action *a* depends on the start condition and hence is always enabled. Action *b* is enabled by the occurrence of *a* and disabled by the occurrence of *c* or *d*, meaning that *b* may occur after *a* has occurred and *c* and *d* have not occurred yet. An analogous explanation applies to actions *c* and *d*. Consequently, the example defines that after the occurrence of *a* a choice is made between *b*, *c* or *d*, such that only one them occurs.

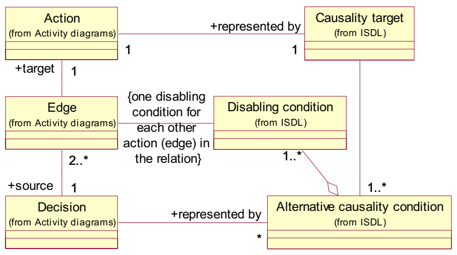


Figure 14. Mapping: UML activity diagrams - ISDL

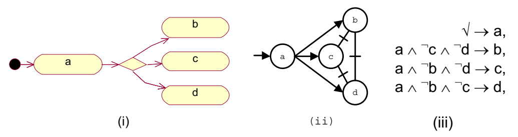


Figure 15. Example: choice relation

4.3.1. Mapping UML activity diagrams onto ISDL. Having explained briefly how a monolithic behavior is defined in ISDL, we describe how a mapping from UML activity diagrams to ISDL can be defined. Figure 14 shows part of such a mapping: the mapping of a decision between actions in Activity diagrams onto disabling conditions of causality targets in ISDL. Figure 15 (i) depicts an example application of the decision operator. Each action involved in the decision is mapped onto a causality target in ISDL. The decision operator implies a certain alternative causality condition for each causality target. This condition consists of the conjunction of disabling conditions, one for each edge from the decision node to an action, except for the edge to the action represented by the causality target itself. These disabling conditions define that the actions that are the targets of a decision mutually disable each other, such that only one of them can happen.

Figure 15 (iii) depicts the ISDL linear text representation of the activity diagram in (i). Figure 15 (ii) depicts the ISDL graphical representation of (iii). Mutual disabling conditions are represented by a short-hand: a disabling condition without the arrow head. Normally, the *and*- and *or*-operator are explicitly represented by a filled and an open square, respectively (e.g., see Figure 32). However, in case a causality condition consists of a single alternative condition, the *and*-operator can be omitted.

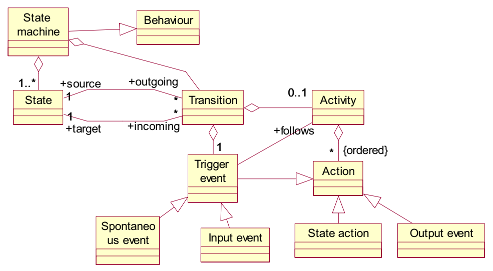


Figure 16. UML statechart language meta-model

4.3.2. Mapping UML statechart diagrams onto ISDL. Figure 16 depicts a simplification of the language meta-model underlying UML Statecharts. A state machine is defined as a collection of states that are related through transitions. Associated with a transition is an activity that is initiated by some trigger, e.g., an input event, which may be followed by multiple state actions and output events.

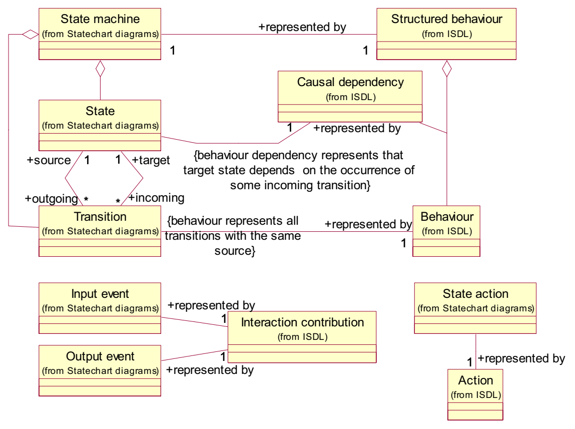

State machine (from Statechart diagrams)
+represented by
1
1
Structured behaviour (from ISDL)
Causal dependency (from ISDL)
State (from Statechart diagrams)
1 +represented by
+source 1
1 +target
{behaviour dependency represents that target state depends on the occurrence of some incoming transition}
+outgoing *
* +incoming
Transition (from Statechart diagrams)
{behaviour represents all transitions with the same source}
+represented by
1
Behaviour (from ISDL)
Input event (from Statechart diagrams)
+represented by
1
Interaction contribution (from ISDL)
1
Output event (from Statechart diagrams)
+represented by
State action (from Statechart diagrams)
1 +represented by
Action (from ISDL)

Figure 17. Mapping: UML statecharts - ISDL

Figure 17 depicts part of a mapping from statecharts onto ISDL. A state machine is represented by a structured behavior, which consists of one sub-behavior for each state representing the (activities associated with the) transitions that can be triggered in this (source) state. A causal dependency represents for each sub-behavior the possible transitions that may enable this behavior (as target state). In addition, the diagram shows that input and output events are mapped onto interaction contributions, since state machines interact with their environment via these events, and state actions onto actions in ISDL.

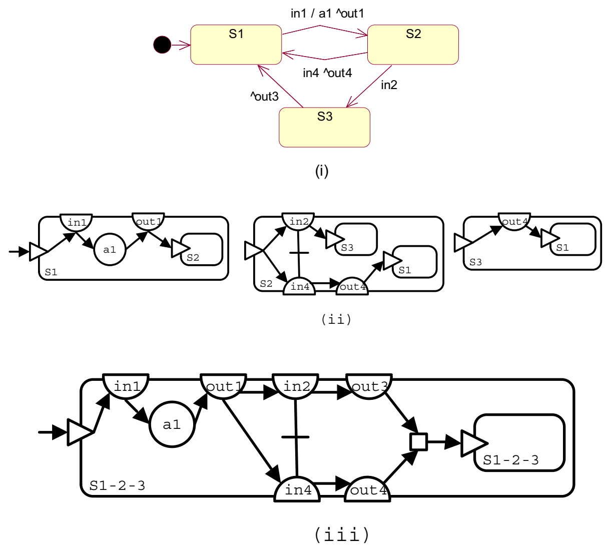


Figure 18. Example statechart simplification

Figures 18 (i) and (ii) depict a simple statechart and its mapping onto ISDL, respectively. A rounded rectangle represents a (sub-)behavior, and a circle segment represents an interaction contribution. Each sub-behavior corresponds to one (source) state. Sub-behavior *S1* represents the only possible transition in state *S1*, which consists of the sequential occurrence of *in1*, *a1* and *out2*, followed by a new instance of *S2*. Sub-behavior *S2* represents the choice between two alternative transitions. A triangle represents a behavior dependency, which can be considered as a place holder for the causality condition that must be satisfied to enable a sub-behavior (target state). For example, the causality condition of the new instance of *S2* is that the transition "*in1*, *a1* and *out1*" has occurred. Figure 18 (iii) depicts the integration of the sub-behaviors into a single monolithic behavior.

5. Example: on-line shopping

The application of a basic viewpoint to relate different system views is illustrated by the design of a simple on-line shopping application. This design is considered from four different viewpoints:

- a *business process viewpoint*, which concerns the design of the business processes that implement the goals of some business (enterprise), and the embedding of application support within these processes;
- a *component structure viewpoint*, which concerns the decomposition of applications in (software) components and their dependencies;
- a *component behavior viewpoint*, which concerns the design of the behavior of the identified (software) components;
- a *component interaction viewpoint*, which concerns the design of the interactions between components, and their relationships.

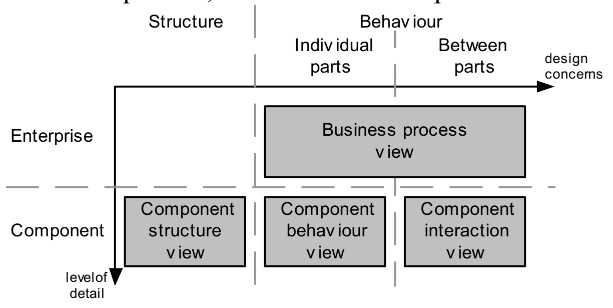


Figure 19. Application views

These viewpoints basically represent two design concerns: structure and behavior, and two levels of detail: enterprise and component. At component level, the behavior concern is divided into two sub-concerns: individual component behavior and component interactions. Figure 19 depicts the concerns and levels of detail being considered, and the associated views on the example application. In this example, the structure concern is not considered at enterprise level, since we consider only a single business process, abstracting from its distribution over enterprise parts.

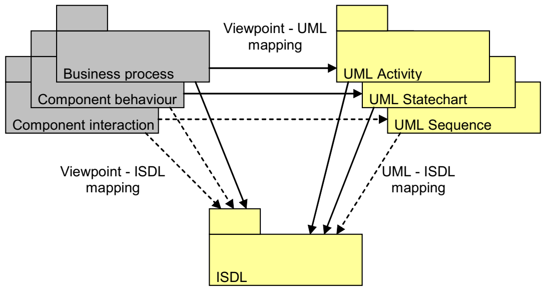


Figure 20. Viewpoint and language mappings

Figure 20 depicts the UML modeling techniques being used to represent the application behavior views. We aim at demonstrating how these behavior views can be related via the basic modeling language ISDL (introduced in section 4.3). For this purpose, we have to define the mappings depicted in Figure 20, and verify their consistency. Due to space limitations, this paper only explains the mappings illustrated by the solid arrows in Figure 20.

5.1. Business process view

Figure 21 depicts an activity diagram that defines the business process view of the on-line shopping application.

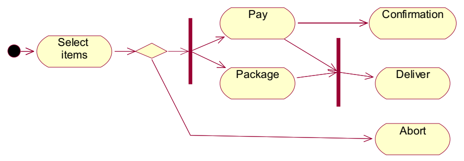


Figure 21. Representing the business view with an activity diagram

As explained in section 3.2, activity diagrams can be used to represent business process concepts, by defining a mapping between the design meta-model of the business process viewpoint and the language meta-model of activity diagrams (see Figure 9).

In this example, the activity diagram defines a business process consisting of a number of business tasks and their ordering. The process starts with the selection of a number of items. Subsequently, either the process is aborted or the items are paid and packaged in parallel. After payment a confirmation is given. After payment and packaging are finished, the items are delivered.

Mapping onto ISDL. The mapping from the business process view onto ISDL can either be performed directly by the business process viewpoint to ISDL mapping, or indirectly by the composition of the business process viewpoint to activity diagrams mapping and the activity diagrams to ISDL mapping. The first mapping is depicted in figure 22, and is consistent with the composition of the latter two mappings, which have been described in section 4.3.

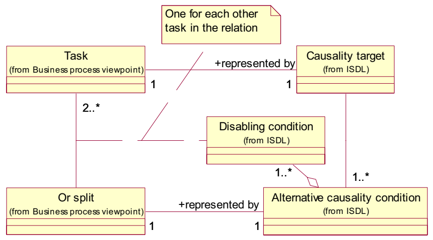

One for each other task in the relation
Task
(from Business process viewpoint)
+represented by
Causality target
(from ISDL)
1
1
2..*
Disabling condition
(from ISDL)
1..*
1..*
Or split
(from Business process viewpoint)
+represented by
Alternative causality condition
(from ISDL)
1
1

Figure 22. Mapping: Business process viewpoint - ISDL

Figure 23 depicts the ISDL diagram that defines the business process view.

For the purpose of this example, we do not consider the embedding of the shopping application in a larger business process, or its relation to other business processes, but delimit the business process to describe only the interactions between the customer and the shopping application. This forms a proper starting point for the implementation of this application in terms of software components.

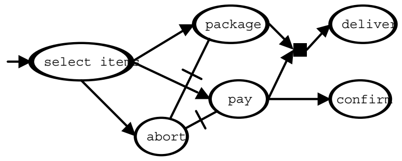


Figure 23. ISDL diagram of the business process view

5.2. Component structure view

Figure 24 depicts the decomposition of the shopping application into four components, and their dependencies:

- Front end (FE): interacts with the customer and coordinates the shopping process;
- Transaction Processor (TP): performs the payment transaction;
- Warehouse (WH): packages the items and forwards them to Parcel Delivery;
- Parcel Delivery (PD): delivers the items to the customer.

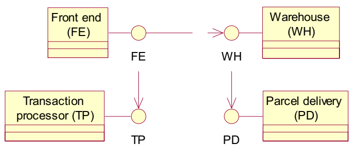


Figure 24. Application decomposition

The component structure view merely identifies the components and how they are interconnected. In the remainder of this example we will focus on the behavior of the individual components and their interactions. The (behavior of the) customer is not explicitly considered. We simply assume he participates in the interactions supported by the application (in particular FE ad PD).

5.3. Component behavior view

Figure 25 depicts the design meta-model of the component behavior viewpoint. In this viewpoint the behavior of a component is considered as an ordered set of two types of actions: external and internal actions. An external action is a contribution of the component to an operation involving another component. This contribution is either a client-side contribution or a server-side contribution, depending on whether the component performs the client or server role in the operation, respectively. The latter contributions can be further decomposed into the sending and receiving of operation request and response messages ('Req', 'Ind', 'Rsp' and 'Cnf', respectively. This will be explained further in section 5.4). An internal action is a complete operation of the component itself, or between internal components.

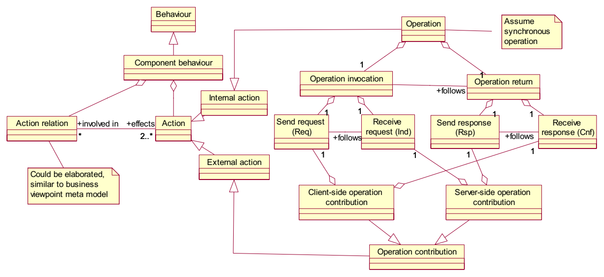


Figure 25. Component behavior viewpoint

In order to define component behaviors using UML statecharts we have to map the concepts of the component behavior viewpoint onto the concepts underlying UML statechart diagrams, which have been explained in section 4.3. Figure 26 depicts part of this mapping. External actions are mapped indirectly onto output and input events, by mapping the receiving of request/response messages onto input events and the sending of these messages onto output events. An internal action is mapped onto a state action, which is an abstraction of the represented internal operation by considering the activity of sending and receiving the involved request and response messages as a single action.

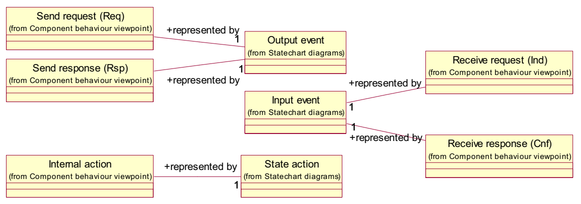

Send request (Req)
(from Component behaviour viewpoint)
+represented by
1
Output event
(from Statechart diagrams)
Send response (Rsp)
(from Component behaviour viewpoint)
1
+represented by
Receive request (Ind)
(from Component behaviour viewpoint)
+represented by
Input event
(from Statechart diagrams)
1
1
+represented by
Receive response (Cnf)
(from Component behaviour viewpoint)
Internal action
(from Component behaviour viewpoint)
+represented by
State action
(from Statechart diagrams)
1

Figure 26. Representing the Component behavior viewpoint with UML statecharts

The behavior of the application Front end component is depicted in figure 27. Based on the representation relation in figure 26, the following interpretation can be given. The first operation that can be invoked is Browse, which returns information about the items that are for sale. After selection of some items (Select), the customer may Checkout to indicate he wants to buy them, thereby providing credit card information. Subsequently, operation Pack is invoked (on WH) to package the items followed by a request (on TP) to take care of the payment (Transaction). If the transaction is accepted, the delivery of the items is Approved (to WH), and the success of the sale is notified (Notify). Alternatively, the sale process may be aborted due to different reasons (canceled by customer or a rejected transaction), which is notified by invoking operation Notify(nosale) (on the customer).

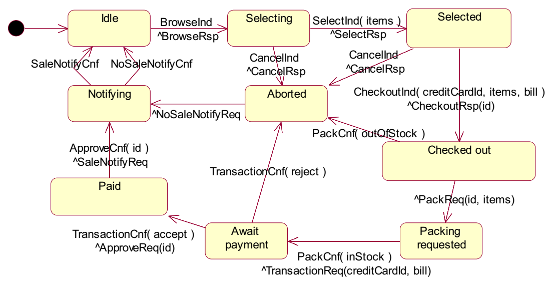


Figure 27. UML statechart: front end behavior

Without further explanation, figure 28 depicts the UML Statechart diagrams of the other components.

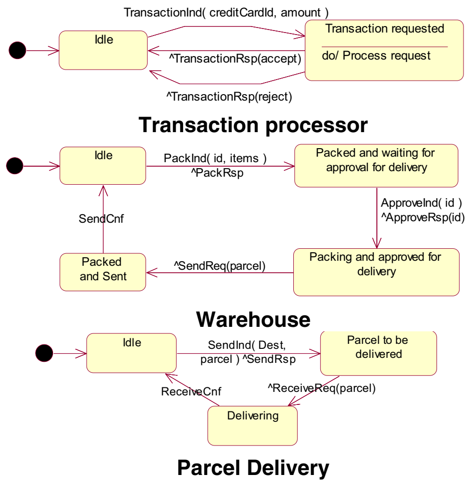


Figure 28. UML statecharts: FE, WH and PD

Mapping onto ISDL. Using the mapping from statecharts to ISDL explained in section 4.3, the statecharts presented above can be translated into ISDL diagrams. Due to space limitations we will not present these diagrams. Instead, section 5.5 will present an ISDL diagram that defines the composite behavior of all components.

5.4. Component interaction view

The component interaction viewpoint relates the client-side and server-side operation contributions modeled in the component behavior viewpoint. UML sequence diagrams can be used for this purpose. We think this is intuitively clear, and therefore omit the presentation of the associated meta-models and the mapping between them. Figure 29 depicts two sequence diagrams, each representing a possible interaction scenario between the shopping application components. For brevity, in most cases only the operation invocations are shown.

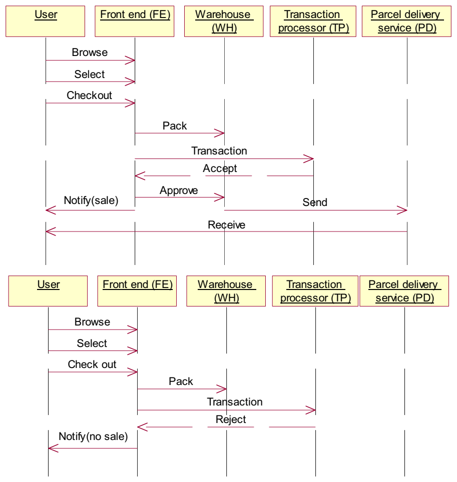


Figure 29. UML sequence diagrams: component interactions

Mapping onto ISDL. Figure 30 (i) illustrates the mapping of an operation (as represented by the meta-model of figure 25) onto ISDL. The sending of an operation request, the receiving of this request, the sending of the operation response, and the receiving of this response, are mapped onto the interaction contributions 'Req', 'Ind', 'Rsp' and 'Cnf'. In addition, the presence of some medium (e.g., middleware supporting remote operations) between the client and server is made explicit.

Figure 30 also shows three possible ISDL abstractions of an operation: (ii) abstracts from individual interaction contributions, (iii) considers the sending and receiving of a request (response) message as a single unit of activity, called Invocation (Return), and (iii) considers the entire operation as a single unit of activity, called Op.

The ordering of operations in UML sequence diagrams can be modeled using the enabling condition of ISDL. However, a sequence diagram generally shows only one specific ordering. An ISDL diagram can be used to represent all possible orderings of operation invocations between components. This is shown in section 5.5.

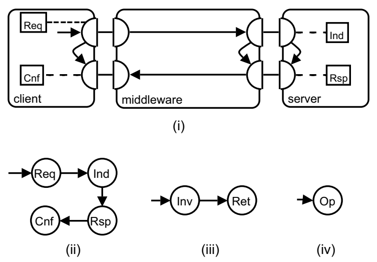


Figure 30. ISDL abstractions of an operation

5.5. Relating component views

The component behavior and component interaction view complement each other. The component behavior view conceives the operation contributions of individual components, and how these contributions are related (for brevity, we have not considered internal component behavior). The component interaction view conceives the relationships between operation contributions from different components.

In principle, the only overlap to be checked between both views is that for each component, the same interactions are identified. However, given the component structure view of section 5.2, and assuming that components interact via some medium as depicted in figure 30, the component behavior view implicitly conceives also the interactions between components, and therefore implicitly conceives the composite behavior of all components. ISDL allows one to define this composite behavior. This behavior is depicted in figure 31 for a single instance of the shopping process, where each operation is represented by a single action, except for 'transaction' that is represented by two actions The disabling relation between actions 'select' and 'cancel' represents that 'cancel' may either disable 'select' or happen after 'select' has happened.

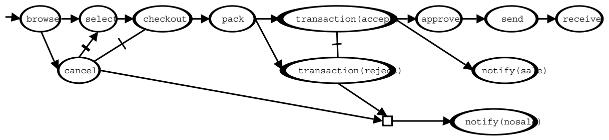


Figure 31. Integrated component view

The above implies that we have to assess whether both views conceive the same possible sequences of interactions. We could do this in two ways:

1. check if the sequence of interactions described by each sequence diagram is a possible execution of the ISDL behavior in figure 31. For the sequences from figure 29, this can easily be established. However, to properly verify this according to the third approach described in section 4.1, a mapping from UML sequence diagrams to ISDL would be necessary. Also a basic relation would be necessary that allows us to verify if the resulting ISDL models are correct sequences of the integrated behavior from figure 31. The benefit of using this approach is that the basic relation and the tool support that implements it may later be re-used for other verifications;
2. map the sequence diagrams to ISDL, integrate them into a single diagram (e.g. using one of the techniques from [18]) and check whether this diagram is equivalent to the diagram in figure 31. In this case a weak equivalence relation may be used, since the sequence diagrams may describe only a subset of all possible interaction sequences.

5.6. Relating enterprise and component views

The component behavior view and component interaction view together (here called the component view) refine the business process view. This means we should assess somehow whether the composition of the UML behavior diagrams that represent the component view correctly implements the UML activity diagram of figure 21. Alternatively, we could assess whether the behavior of figure 31, which represents the composite component behavior, is a correct refinement of the behavior representing the business process view in figure 23 (according to the third approach discussed in section 4.1). The latter option is preferable, because a method for conformance assessment has been developed for ISDL [16,15]. This method is based on the second refinement technique discussed in section 4.2. Because it is beyond the scope of this paper to explain the method, we will appeal to the intuition of the reader to illustrate the conformance between both ISDL diagrams.

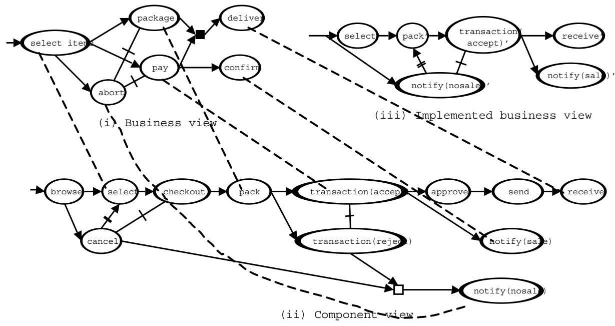


Figure 32. Refinement relation

The relation between figure 32 (i) and (ii) depicts for each original action in the figure 32 (i) a corresponding action, called reference action, in figure 32 (ii), such that the occurrence of the reference action corresponds to the occurrence of the original action. All other actions in figure 32 (ii) are called inserted actions and should be removed, since they are introduced in the refinement step.

Rules have been defined in [16,15] to remove inserted actions in such a way that the relationships between the references actions are preserved.

Figure 32 (iii) depicts figure 32 (ii) after removing the inserted actions. This figure is however not (strictly) equivalent to the original abstract ISDL diagram. One can deduce the following implementation errors: actions 'pay' and 'package' have been implemented in a particular sequence, a 'no sale' notification can be received before items have been selected or after the items have been selected and packaged. However, in certain cases one may be interested in a weaker variant of equivalence, for example, to allow that independent actions are implemented in a particular sequence.

6. Conclusions

This paper introduces an approach to define viewpoints and modeling languages used in an architectural design process. Our approach is generic enough to cater for the different relations that may exist between viewpoints and the modeling languages that are used to represent them. The paper discusses several ways to keep different views (that are constructed according to viewpoints) aligned and consistent.

This paper proposes the use of a basic viewpoint as a basis for defining and relating viewpoints for distributed application design. We claim to have such a basic viewpoint definition for structure and behavior concerns in [15]. This is motivated by the fact that we applied this viewpoint for the design from different viewpoints, such as business process design, distributed application design and protocol design (as explained in [4]). Our basic viewpoint defines some basic relations, such as the refinement relation and the behavior equivalence relation that we can re-use to define relations between other viewpoints.

In the future, we intend to further elaborate on the example introduced in section 5. We intend to build tool support for the basic viewpoint that helps us automatically verify the consistency between views. We will integrate this tool with existing CASE tools for UML modeling, according to the meta-model mappings discussed in this paper. To do this, we must find a more precise way to express these mappings. Finally, we plan to carry out case studies to validate the claims made in this paper.

Acknowledgements

This work is partly supported by the Telematica Instituut in the context of the ArCo project and by the European Commission in the context of the MODA-TEL IST project.

References

[1] J. Aagedal, and Z. Milošević, ODP Enterprise Language: UML Perspective, *Proc. of EDOC 1999*, IEEE, 1999, pp. 60-71.

[2] J-M. Cornily, Specifying distributed object applications using the reference model for Open Distributed Processing and the Unified Modeling Language, *Proc. of EDOC 1999*, IEEE, 1999.

[3] R. Dijkman, D. Quartel, L. Ferreira Pires, and M. van Sinderen, Semantic Verification of Behavior Conformance, *Proc. of the 11th Workshop on Behavioral Semantics*, Seattle, WA, USA, 2002, pp. 43-54.

[4] R. Dijkman, D. Quartel, L. Ferreira Pires, and M. van Sinderen, A Design-for-Change Approach: Developing Distributed Applications from Enterprise Models, *CTIT Technical Report 02-22*, University of Twente, The Netherlands, 2002.

[5] D. D'Souza, and A. Cameron-Wills, *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley, Reading, MA, USA, 1999.

[6] A. Gerber, M. Lawley, K. Raymond, M. Steel, and A. Wood, Transformation: the missing link of MDA, *Proc. of ICGT 2002*, Barcelona, Spain, 2002, pp. 90-105.

[7] R. Hilliard, Views and viewpoints in software systems architecture, *Proc. of IFIP Conf. on Software Architecture*, San Antonio, USA, 1999.

[8] R. Hubert, *Convergent Architecture*, Wiley, New-York, NY, USA, 2002.

[9] ITU-T / ISO, *Open Distributed Processing Reference Model Part 1-4*, ITU-T Specification ITU-T 90x and ISO/IEC Spec. ISO/IEC 10746-1..4, 1995.

[10] A. Naumenko, *Triune Continuum Paradigm: a Paradigm for General System Modeling and its Application for UML and RM-ODP*, Ph.D. Thesis, Swiss Federal Institute of Technology, Switzerland, 2002.

[11] OMG, Clear, Clean, Concise (3C) proposal for UML 2.0, OMG Specification ad/02-09-15, 2002.

[12] OMG, *UML Profile for EDOC Specification*, OMG Spec. ptc/02-02-05, and ad/01-08-20, 2002.

[13] OMG, Meta Object Facility Specification, OMG Spec. formal/2002-04-03, 2002.

[14] OMG, *Model Driven Architecture*, OMG Specification ormsc/02-07-01, 2001.

[15] D. Quartel, *Action Relations – Basic Design Concepts for Behavior Modelling and Refinement*, Ph.D. Thesis, University of Twente, The Netherlands, 1998.

[16] D. Quartel, L. Ferreira Pires, and M. van Sinderen, On architectural support for behavior refinement in distributed systems design, *Journal of Integrated Design and Process Science* ***6***(1), March 2002.

[17] M. van Sinderen, L. Ferreira Pires, and C. Vissers, Protocol Design and Implementation Using Formal Methods, *The Computer Journal* **35**(5), June 1992, pp. 478-491.
[18] S. Uchitel, J. Kramer, and J. Magee, Synthesis of behavioral models from scenarios, *IEEE Transactions on Software Engineering* **29**(2), 2003, pp. 99-115.
[19] C. Vissers, G. Scollo, M. van Sinderen, and H. Brinksma, Specification styles in distributed systems design and verification, *Theoretical Computer Science* **89**, 1991, pp. 179-206.
[20] J. Davies, and J. Woodcock, *Using Z: specification, refinement and proof*, Prentice-Hall, 1996.
[21] A. Wegmann, On the Systemic Enterprise Architecture Methodology (SEAM), *Proc. of ICEIS 2003*, Springer, Berlin, Germany, 2003, pp. 483-490.