Consistency in Multi-Viewpoint Architectural Design of Enterprise Information Systems

Remco M. Dijkman [1,2], Dick A.C. Quartel [2], Marten J. van Sinderen [2]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
R.M.Dijkman@tue.nl

[2] University of Twente, Enschede, The Netherlands
{D.A.C.Quartel,M.J.vanSinderen}@utwente.nl

Abstract. Different stakeholders in the design of an enterprise information system have their own view on that design. To help produce a coherent design this paper presents a framework that aids in specifying relations between such views. To help produce a consistent design the framework also aids in specifying consistency rules that apply to the view relations and in checking the consistency according to those rules. The framework focuses on the higher levels of abstraction in a design, we refer to design at those levels of abstraction as architectural design. The highest level of abstraction that we consider is that of business process design and the lowest level is that of software component design. The contribution of our framework is that it provides a collection of basic concepts that is common to viewpoints in the area of enterprise information systems. These basic concepts aid in relating viewpoints by providing: (i) a common terminology that helps stakeholders to understand each others concepts; and (ii) a basis for defining re-usable consistency rules. In particular we define re-usable rules to check consistency between behavioural views that overlap or are a refinement of each other. We also present an architecture for a tool suite that supports our framework. We show that our framework can be applied, by performing a case study in which we specify the relations and consistency rules between the RM-ODP enterprise, computational and information viewpoints.

Keywords: Multi-Viewpoint Design; Enterprise Information Systems; Viewpoint Consistency; Conceptual Modelling; View Integration

1. Introduction

In any large-scale design, different people with different interests are involved. These people, or *stakeholders* as we call them, have their own way of looking at a system, for which they use their own modelling languages, techniques and tools. Informally, we call the way in which a stakeholder looks at a system the *viewpoint* of that stakeholder. From his viewpoint, each stakeholder constructs his own design part, or *view*. However, because views are parts of the same multi-viewpoint design, we must preserve the coherence and consistency between the different views.

In this paper we propose a framework that aids in preserving the consistency in a multi-viewpoint design of Enterprise Information Systems. To this end the framework provides:

- a collection of basic concepts that is common to all viewpoints;
- a means to specify relations between different views;
- a means to specify consistency rules that apply to these relations;
- re-usable relations; and
- re-usable consistency rules.

The framework focuses on the architectural design of Enterprise Information Systems, which focuses on higher levels of abstraction in the design process. The highest level of abstraction that we consider is the level at which the system is described in its enterprise environment (e.g. by means of a business process in which the system is used). The lowest level of abstraction that we consider is the level at which the system parts correspond to parts that can be deployed on some middleware system (e.g. J2EE or Web Services).

The problems of coherency and consistency in multi-viewpoint design are well-known and several frameworks are proposed to address these problems [14,13,5,24,10,12,11,2,1,25,16,8,17]. This paper contributes to this work by providing a common collection of basic concepts to specify consistency rules and by providing re-usable viewpoint relations and consistency rules. The benefit of using common, basic concepts are that these concepts:

- provide a common terminology to all stakeholders, helping them to understand each others concepts more easily; and
- provide a basis for specifying re-usable relations and rules, something that has, to the best of our knowledge, not been attempted before.

By providing these techniques, we claim that our framework reduces the time and effort needed to specify and check relations and consistency between viewpoints.

We derived the elements of our framework in two steps. Firstly, we analyse existing frameworks for multi-viewpoint design and sets of concepts for design from the viewpoints. From these frameworks and concepts, we generalize to develop a common collection of basic concepts. We also use these frameworks and concepts to derive frequently occurring inter-viewpoint relations for re-use. Secondly, we apply the basic concepts and relations in a case study to evaluate them.

This paper is further structured as follows. Section 2 presents related work and, based on an analysis of the related work, motivates the contribution of our framework. Section 3 presents the framework. Also, it explains and justifies the re-usable relations and consistency rules that we define further on. Section 4 presents the common collection of basic concepts that supports the specification of (re-usable) relations and consistency rules between views. Section 5 formally defines the re-usable consistency rules, such that they can be checked. Section 6 presents an example in which the framework is used. Section 7concludes.

2. Related Work

In Figure 1 we plotted the support that existing frameworks in the area of architectural design provide for defining view relations and checking consistency in multi-viewpoint design. We compared the frameworks with respect to two aspects of viewpoint relations: (i) the expressiveness of the viewpoint relations; and (ii) the conceptual support to represent the viewpoint relations.

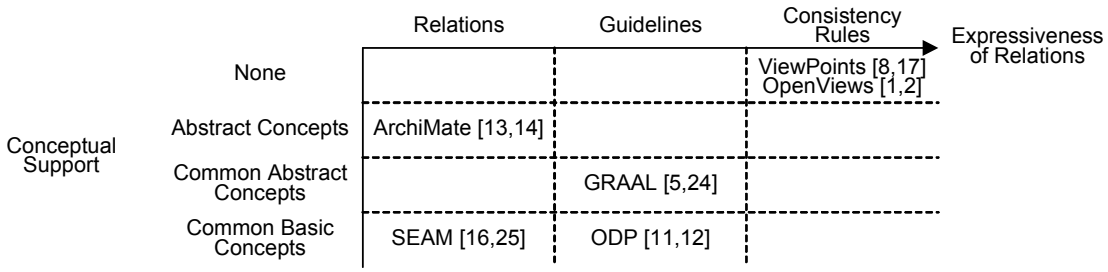


Figure 1. Existing Frameworks and their Support for Consistency Checks

We distinguish three levels of expressiveness of viewpoint relations. At the lowest level, a framework supports the definition of relations between views, but not the (consistency) rules that apply to these relations. At the next level a framework supports the definition of consistency guidelines that each stakeholder in a multi-viewpoint design must follow. These guidelines are defined informally and no automated support is available to check them. At the highest level, a framework supports the definition of consistency rules and their automated checking.

A framework can provide conceptual support to represent relations between the viewpoints, by defining a set of concepts that crosses the boundaries between the viewpoints and relations between these concepts. For example, consider a set of concepts that includes an 'Action' concept and an 'Information Item' concept and a relation that relates an 'Action' to the 'Information Items'. This set crosses the boundaries between a viewpoint that focuses on behavioural aspects and a viewpoint that focuses on information aspects, allowing a designer to relate those viewpoints. We discovered three different forms of conceptual support in literature. *Abstract concepts* provide abstractions of concepts that are used in the viewpoints covered by the framework. They have relations with each other, which allow a designer to represent relations between views from the viewpoints in the framework. Abstract concepts are developed with the sole purpose of representing the relations between views and cannot be used to represent the views themselves in detail. *Common abstract concepts* have the additional property that they are shared between the views, where regular abstract concepts are different for each of the views. Like abstract concepts, *(Common) basic concepts* have relations that allow a designer to represent relations between views. However, unlike abstract concepts, basic concepts can

represent some aspects from the views in detail. In theory this makes it possible to design some (part of the) views with basic concepts rather than viewpoint concepts. But typically a composition of basic concepts or a specialization of a basic concept is necessary to represent a single viewpoint concept. This makes a view designed with basic concepts harder to develop and understand than a view designed with viewpoint concepts. For that reason viewpoint concepts are more frequently used for viewpoint design.

Figure 1 illustrates the potential for a more advanced framework that combines the highest level of expressiveness with conceptual support for representing viewpoint relations. Moreover, we show below that this combination allows for the definition of re-usable rules to check consistency. To the best of our knowledge this has not yet been attempted.

3. A Framework for Preserving Consistency in Multi-Viewpoint Design

In this section we outline our framework for preserving consistency in a multi-viewpoint design. Firstly, we present the elements of multi-viewpoint design. Secondly, we present a means to specify relations between viewpoints. Thirdly, we explain the roles that basic concepts play when specifying viewpoint relations and when checking consistency in a multi-viewpoint design.

3.1. Multi-Viewpoint Design

Figure 2 illustrates the elements of multi-viewpoint design. Multi-viewpoint design is based on the observation that multiple stakeholders contribute to a design.

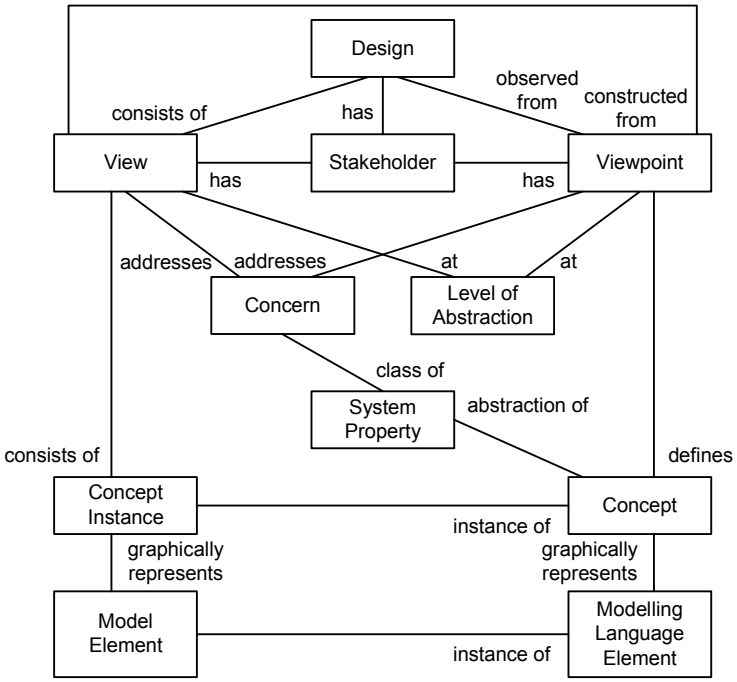


Figure 2. The Elements of Multi-Viewpoint Design

Each of these stakeholders focuses on a part of the design, which we call the *view* of that stakeholder. More specifically, we say that a stakeholder focuses on certain design concerns and considers these concerns at a certain level of abstraction. This observation is shared by most of the frameworks that we considered in section 2. A *design concern* is a class of system properties. For example, the behaviour concern is the class of properties that address the behaviour of a system, such as the activities that can occur in the system and the relations between these activities. A *level of abstraction*, also called a level of detail, is a relative position in the design process that prescribes what design information is considered essential at that position in the design process.

A *viewpoint* is a prescription of the concerns at a certain level of abstraction that a stakeholder must address and the concepts that he uses to do so. A *design concept* is an abstraction of some common and essential property of distributed systems. A view is constructed using instances of the concepts from the corresponding viewpoint. To communicate a view it can be represented graphically (or textually) by a model. To this end each concept instance is represented by one or more model elements. Correspondingly the viewpoint defines the modelling language used to construct the

models, such that each concept defined by the viewpoint is represented by one or more modelling language elements. In this way, viewpoints and views, concepts and concept instances and modelling language elements and model elements have a template/instance relation, such that one provides a template to construct the other.

The concepts shown in Figure 2 are in line with the concepts defined in the IEEE 1471 recommended practice [10].

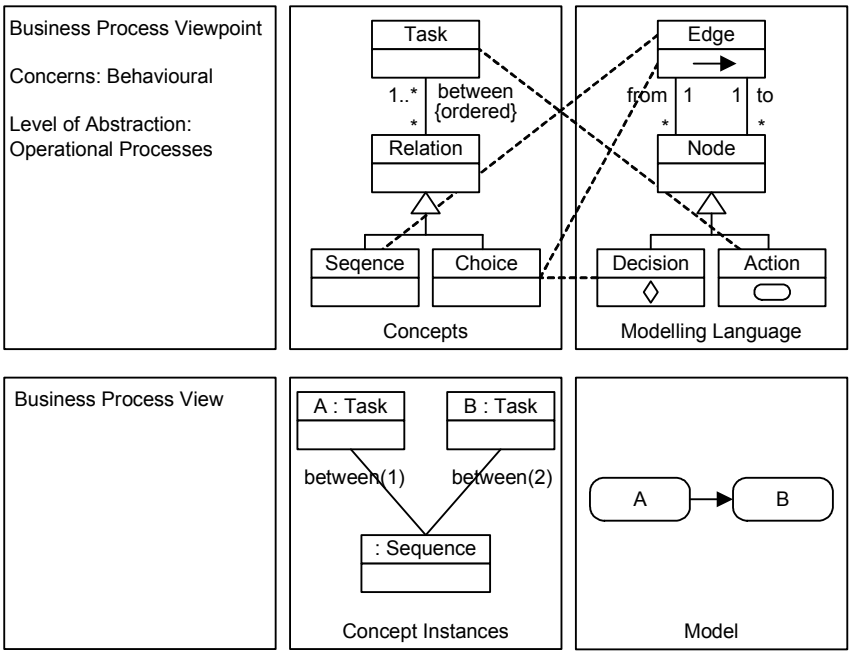


Figure 3. Example of a Viewpoint and a View

Figure 3 shows an example of a viewpoint. The viewpoint is that of an operational manager, who focuses on the behavioural concern at the level of abstraction of operational processes. He uses concepts such as 'Task' and 'Task Relation' to construct views from his viewpoint, which he graphically represents using modelling language elements such as 'Node' and 'Edge'. The dashed line informally depicts the ‘representation relation’ between concepts and the modelling elements that represent them. These modelling elements are associated with a graphical representation. The figure also shows an example of a view that is constructed according to the viewpoint. The view shows a process that consists of a sequence of two tasks, *A* and *B*.

3.2. Relations and Consistency in Multi-Viewpoint Design

To construct a coherent and consistent multi-viewpoint design, relations between the viewpoints (and views constructed according to them) must be specified explicitly. We represent relations between two viewpoints by relating concepts from these viewpoints. The semantics of these relations must be specified separately. This can partly be done by means of consistency rules. A *consistency rule* is a rule that represents a requirement on the relation between concepts from different viewpoints. In a consistent design all consistency rules must evaluate to 'true'. In the remainder of this paper we use the UML Meta Object Facility (MOF) [19] and the Object Constraint Language (OCL) [18] to specify viewpoint concepts and their relations as well as consistency rules that apply to these relations. We assume that the reader has basic knowledge of both MOF and OCL.

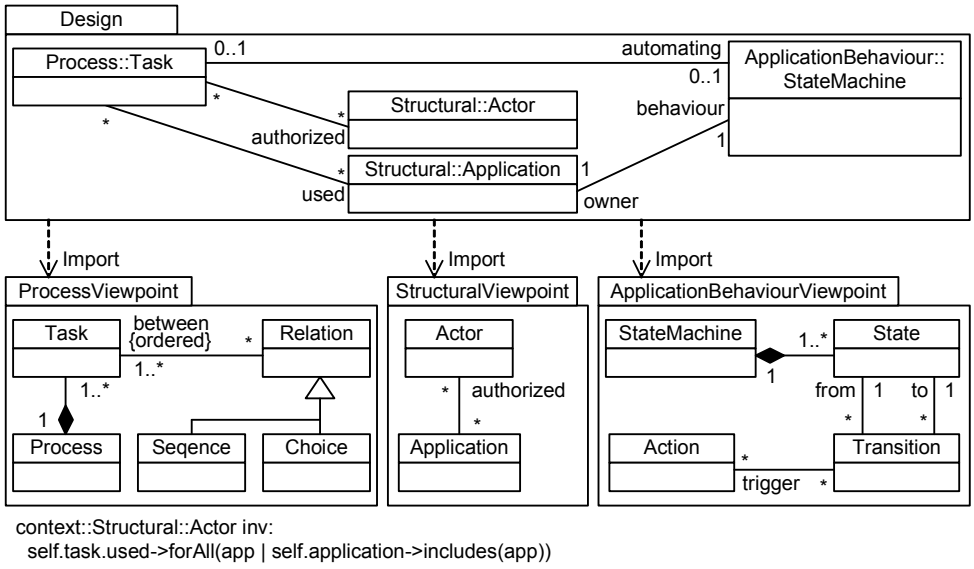


Figure 4. Example of Related Viewpoints

Figure 4 shows an example of three related viewpoints. One viewpoint is the behavioural viewpoint from Figure 3 and the others show the structural aspect of the enterprise and the behaviour of individual applications. The structural viewpoint can be used to represent the actors and applications that exist in an enterprise and to represent which actor is authorized to use which application. The application behaviour viewpoint can be used to represent the behaviour of an applications by

means of a statemachine. The statemachine represents the states that an application can be in and how actions can trigger a transition to another state. Each viewpoint is represented as a MOF package. The overall design is also represented as a MOF package. The design imports each of the used viewpoints, such that all of the concepts (and relations) from the viewpoints are available to it. It also specifies the relations between the viewpoints. These relations represent that actors can be authorized to perform tasks and that applications are used to perform tasks. The relations also relate applications to their behaviour and they represent that a process can be fully automated by (the behaviour of) an application. One consistency rule is specified that applies to the relation between the viewpoints. This rule specifies that an actor must be authorized for all applications used in tasks for which he is authorized.

Since viewpoints consider certain concerns at a certain level of abstraction, we can position the viewpoints of a design relative to each other. The same goes for views, because they are constructed as instances of viewpoints. Frequently occurring relations between viewpoints, and therefore views, can be inferred from the relative position that viewpoints can have with respect to each other.

Figure 5 illustrates the relative position of some viewpoints in a table. The columns of the table represent the different concerns of stakeholders in the design, while the rows represent the levels of abstraction at which these concerns are considered. The rows in the table are ordered, such that the level of abstraction decreases (and therefore the level of detail increases) as we get to lower rows in the table.

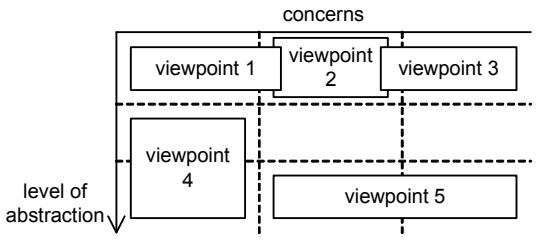


Figure 5. Example of Viewpoints in a Design

Because each viewpoint considers certain design concerns at a certain level of abstraction, viewpoints can be related by considering related levels of abstraction and

by considering related concerns. This gives rise to two frequently occurring viewpoint relations: the refinement relation and the overlap relation. Two viewpoints have a *refinement relation*, if they (partly) consider the same concerns at different levels of abstraction. In case of a refinement relation between two viewpoints, the more concrete viewpoint is a refinement of the more abstract viewpoint. If two viewpoints have partly overlapping concerns we say that they have an overlap relation. Two viewpoints have an *overlap relation* if they (partly) consider the same concerns at the same level of abstraction. For example, viewpoints 1 and 2 from Figure 5 have an overlap relation, while viewpoints 1 and 4 have a refinement relation in which viewpoint 1 is the more abstract and viewpoint 4 is the more concrete viewpoint.

3.3. Using Basic Concepts in Multi-Viewpoint Design

A common collection of basic concepts can be used in a multi-viewpoint design for the following purposes:

1. As a common frame of reference that helps stakeholders, or a stakeholder that is responsible for maintaining relations between the viewpoints, to understand the concepts of the other stakeholders. Because, if a stakeholders knows what his concepts represent in terms of basic concepts, he can explain them to other stakeholders in those terms.
2. As a source of re-usable concepts, relations, consistency rules and notation. Because, if a viewpoint is defined in terms of the basic concepts, all concepts, relations, consistency rules and modelling language elements that are defined on the basic concepts are automatically inherited by the viewpoint.

In our framework we exploit (2) by defining re-usable relations and consistency rules on the basic concepts. In particular we define a *refinement* and an *overlap* relation and corresponding consistency rules on the basic concepts, because we have shown a need for these relations in section 3.2.

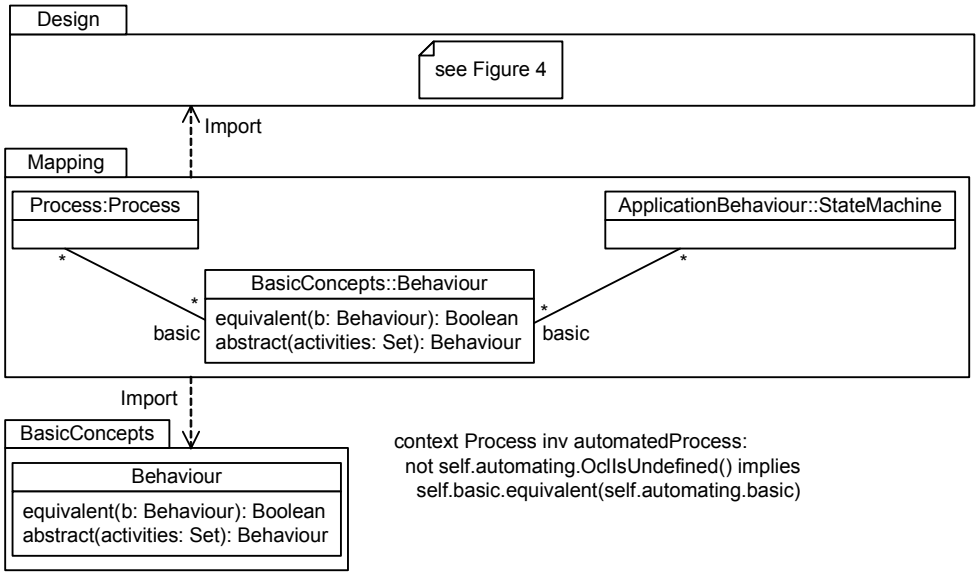


Figure 6. Relating Views via Basic Concepts

Figure 6 illustrates how the basic concepts can be used to specify and check a particular consistency in the design from Figure 4. This design allows a designer to relate a process to the (behaviour of) an application that automates this process. Figure 6 specifies a part of the mapping between the viewpoint concepts and the basic concepts. This mapping maps processes and statemachines to basic behaviours. It is the responsibility of the stakeholder to create a basic behaviour for each process and statemachine and to relate that basic behaviour to the corresponding process or statemachine. In the next section we explain how this can be facilitated by tools. Once a basic behaviour exists for each process and statemachine, we can check the consistency rule. This consistency rule represents that, if a process is automated by a statemachine, the behaviour of the process must be equivalent to the behaviour of the statemachine. The consistency rule uses the basic consistency rule 'equivalent' that is defined on the behaviour basic concept.

Checking consistency in this way is very similar to checking consistency by means of a formalism; checking consistency by means of a formalism involves mapping viewpoint concepts to that formalism, while our approach involves mapping viewpoint concepts to basic concepts. However, we can observe that formalisms are often aimed towards use for particular design concerns. Therefore, as a foundation for verifying consistency between views, a collection of formalisms may be required that

each addresses its own concerns. This presents us with the additional challenge of maintaining the consistency between formalisms. Furthermore, formalisms are mainly aimed towards mathematical rigour, rather than ease of use and understanding, which are important qualities when trying to understand viewpoint concepts and how they are used and related. Therefore, we claim that using basic concepts helps the designer, because he does not have to understand the formalisms involved in checking consistency, nor how they are related. Moreover, through the basic concepts we *do* use formalisms to check consistency, but these formalisms are invisible to the designer. They provide a formal basis of the concepts that is only used by the tool to perform the consistency checks, but invisible to the user of the tool. The user only sees the basic concepts.

3.4. A Tool Architecture to Support the Framework

Figure 7 shows the a tool architecture that supports the framework. The architecture shows the software components and illustrates the data that they can contain and exchange.

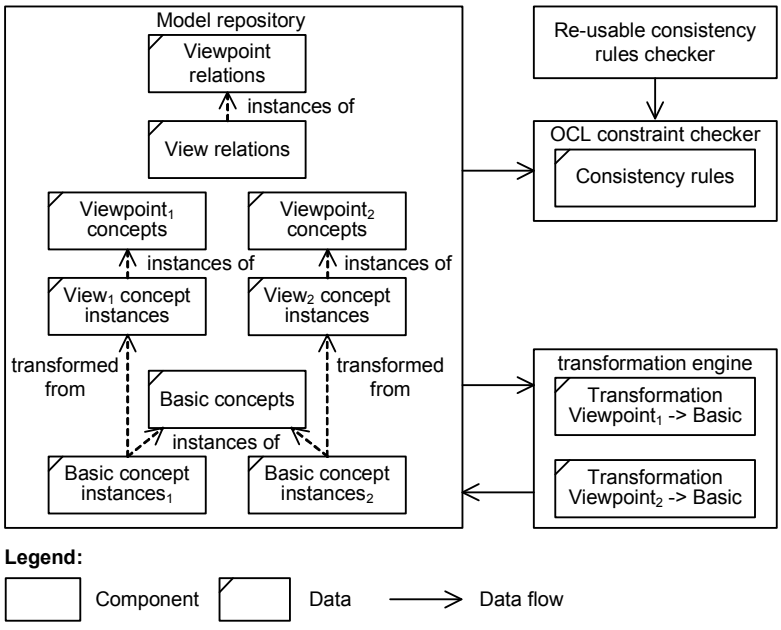


Figure 7. Tool Architecture

The architecture contains a model repository in which the designer stores the concepts that the viewpoints define, as well as the concept instances that represent the views. The designer also stores the views and viewpoint relations in the repository. The model repository is MOF compliant. We store our basic concepts and their relations, as we define them in section 4, in the repository.

The architecture contains a transformation engine that can transform views into compositions of basic concept instances. Such transformations are necessary to re-use the consistency rules that are defined on the basic concepts. To use these consistency rules between two viewpoints, both viewpoints are transformed into compositions of basic concepts, after which the consistency rules that are defined on the basic concepts can be used. The transformations are defined by the stakeholders of the viewpoints, to reflect the mapping of viewpoint concepts onto basic concepts.

The OCL constraint checker can verify consistency rules that the designer prescribes in the form of OCL constraints. The OCL constraint checker can invoke the checker for the re-usable consistency rules that are defined on the basic concepts.

We developed an implementation of the tool architecture [4]. However, that implementation does not yet implement the relation between the OCL constraint checker and the re-usable consistency rules checker. A MOF M2 model of the basic concepts that can be inserted into the model repository is also available.

4. Basic Concepts for Multi-Viewpoint Architectural Design

In this section we present our basic concepts. The concepts are developed through careful analysis of the domain of architectural design of distributed systems and case studies in this area [21,6]. Earlier versions of the concepts are presented in [22,23,6]. We distinguish between structural, behavioural and information concepts. In this paper we explain the basic concepts only briefly. We refer to [3] for a more detailed explanation and for the MOF M2 models that allow a stakeholder to define an (automated) mapping from his viewpoint concepts to the basic concepts.

4.1. Structural Concern

The structure of a system is the aggregate of the system's parts and their relationships to each other. We consider two kinds of relationships between system parts. The first kind is the *connection relationship* that exists between parts that interact via some communication mechanism. The second kind is the *part-whole relationship* that exists between a system and its parts. We can consider each part as a (sub-)system. Therefore, this relation can also exist between a part (sub-system) and its sub-parts. The structure of a system can change over time. However, in this paper we focus on the structure of a system at a certain moment in its lifecycle, which we call a *structural snapshot*.

Figure 8 graphically represents the concepts that we use to represent structural snapshots. An *entity* represents a logical or physical part of a system that carries behaviour. An entity can be drawn inside another entity, to represent that it is part of the other entity. An *interaction point* represents a shared communication mechanism that two or more entities can use to interact. An *interaction point part* represents an entity's (potential) participation in a shared communication mechanism.

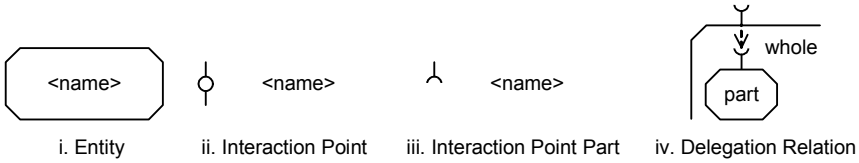


Figure 8. Graphical Representation of Structural Concepts

4.2. Behavioural Concern

The behaviour of a system consists of the activities that can be performed by the system and the relations between these activities. The behaviour of a system can be structured into sub-behaviours to improve modularity of the behaviour. An activity can be performed either by a single system part or by some system parts in collaboration. It produces a tangible or intangible result that is available to all parts that engage in the activity. This result is available to the parts at some logical or

physical location. An activity takes time to be performed. Hence, it starts and finishes at particular time moments. Two activities are related if the occurrence of one depends on the (non-)occurrence of the other.

Figure 9 graphically represents the concepts that we use to represent behaviour. We use a *behaviour block* to represent a behaviour. An *action* represents the successful completion of an activity that is performed by a single entity. An *interaction* represents the successful completion of an activity that is performed by some entities in collaboration. An *interaction contribution* represents the participation of an entity in an interaction. Actions and interaction contributions are assigned to a behaviour, by drawing actions inside the behaviour to which they are assigned and interaction contributions on the border of the behaviour to which they are assigned. The name of an action or interaction contribution is drawn inside a box that is attached to it by a dashed line. By convention, interaction contributions of the same interaction must have the same name. Attributes represent the possible result of an action or interaction. If an action of interaction occurs, its attributes are given a value. Together, these values represent the result of the action or interaction. Attributes have a name and a type. The type identifies the possible values and the information structure of those values. They are defined in the information concern. Constraints may further constrain the possible values of an attribute. Two special kinds of attributes exist: the time and the location attributes, which represent the time and the location at which an action or interaction occurs, respectively.

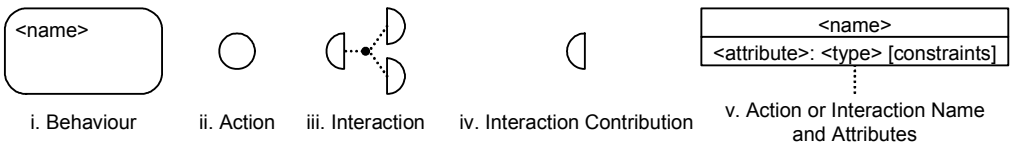


Figure 9. Graphical Representation of Behavioural Concepts

Each action has a condition for its occurrence. This condition represents the dependency of the action to other actions or interactions. Similarly, each interaction contribution has a condition for its occurrence. By giving an interaction contribution, rather than an interaction, a condition, each entity can specify its own conditions for the occurrence of the corresponding interaction. In this paper we do not present a

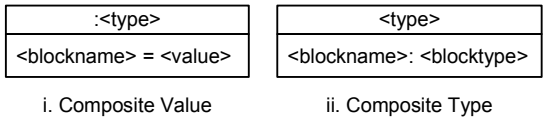
detailed language for specifying the conditions of actions and interactions. Rather, we explain in section 5 how these conditions can be formalized by means of Petri nets, which is sufficient for the reader to understand how consistency can be checked by means of the basic concepts. For more details on representing conditions and on behaviour structuring, we refer to [3].

The relation between the behavioural and the structural concepts is as follows. Behaviours are assigned to the entity of which they represent the behaviour. Interactions occur between entities and must therefore be drawn between the behaviours of those entities. The value of a location attribute of an interaction must represent an interaction point between the entities that participate in the interaction.

4.3. Information Concern

Information concepts can be used to represent the possible values and the structure of an attribute type. It is not our goal to define detailed information concepts. Therefore, we only define minimum requirements that those concepts should meet to be usable in our framework. A designer must define a *binding* between a language's concepts and our concepts to show that the language meets these requirements. This binding also defines how the language's concepts can be used in our framework.

Figure 10 shows our information concepts. An *information value* represents the result of an action or interaction. Each information value has an *information type* that represents the structure of information values of that type and a set of allowable information values. An information type can either be *primitive* (unstructured) or *composite* (structured). In case it is composite, it consists of information blocks of specified types. An information value of a composite type consist of information values that are governed by the information types identified by the blocks of the composite type. Information types with a special status are the 'Time' and 'Location' types. A value of type 'Time' represent the time at which the result of an action or interaction is available. A value of type 'Location' represents an interaction point in the structural concern.



i. Composite Value ii. Composite Type

Figure 10. Graphical Representation of Information Concepts

We defined a binding of UML to our basic concepts [3]. We bind the UML instance specification concept to the information value concept. We bind the UML data type concept to the primitive type concept. This means that the primitive types that are available when using the UML binding are: the UML Boolean, Integer, String, Unlimited Natural and Enumeration types. We bind the UML class concept to the composite type concept and the UML property concept to the information block concept. We bind OCL constraints to conditions and constraints in the behavioural concern. In this way we can represent information values and information types in UML.

5. Re-usable Basic Consistency Rules for the Behavioural Aspect

This section explores the notions of overlap and refinement from section 3.2 in more detail and motivates some frequently occurring refinement and overlap relations, with the corresponding consistency rules, which it defines as re-usable relations on the basic concepts. We focus on defining these relations for the behavioural aspect. For which we first explain how we formalize it using Petri nets.

5.1. Formalizing Behaviour Using Petri Nets

We use a labelled Petri net to formalize a basic behaviour. To this end a Petri net represents when an action or interaction in a behaviour can occur. A labelled Petri net is a four-tuple (P, T, F, l), such that:

- P is the set of *places*.
- T is the set of *transitions*.
- $F \subseteq (T \times P) \cup (P \times T)$ is the *flow* relation that connects places and transitions. A flow has a direction, such that each place or transition has a set of *incoming* places or transitions and a set of *outgoing* places or transitions. Incoming

places or transitions of x, denoted $\bullet x$, are the places or transitions with a flow going from it to x (formally $\bullet x = \{y \mid (y, x) \in F\}$). Outgoing places or transitions of x, denoted $x\bullet$, are the places or transitions with a flow pointing to it from x (formally $x\bullet = \{y \mid (x, y) \in F\}$).

- $l : T \rightarrow (L \cup \{\tau\})$ is the labelling function that labels transitions with a meaningful label or with the 'silent' label τ that represents that nothing observable happens. We label each transition with the name of the action or interaction contribution that it represents (or with the silent label), such that we can represent when this action or interaction contribution can occur.

As a notational convention we denote the elements of a Petri net by subscripting these elements with the identifier of the Petri net. For example, we denote the places of a Petri net N as P_N.

$M : P \rightarrow Nat$ (in which *Nat* is the set of natural numbers) represents the *marking* of a Petri net. A marking relates each place to a number of *tokens*. M^i is a special marking, called the *initial marking*. We use the initial marking to represent the start situation of a behaviour. We express a Petri net N with marking M as: (N, M).

We graphically represent a place by a circle, a transition by a vertical bar and a flow by an arrow that is attached to the places and/or transitions that it connects. We represent a label by drawing the label close to its transition and a marking by drawing black dots, which represent tokens, on places.

A transition t is *enabled* in (N, M), if there is at least one token on each of its incoming places. We express that t is *enabled* in (N, M) as: $(N, M)\ [t\rangle$. Formally, $(N, M)\ [t\rangle$ if and only if for each $p \in t\bullet : M(p) \geq 1$. If a transition is enabled, it can *fire*. We use firing of a transition to represent that the corresponding action or interaction contribution occurs. If a transition fires, it removes one token from each of its incoming places and puts one token on each of its outgoing places, causing the marking to change. We denote firing of transition t in a Petri net N with marking M, causing it to change into a marking M' as: $(N, M)\ [t\rangle\ (N, M')$. Formally, if $(N, M)\ [t\rangle$ then $(N, M)\ [t\rangle\ (N, M')$, such that:
$M'(p) = M(p) - 1$, if and only if $p \in \bullet t$ and $p \notin t\bullet$

$M'(p) = M(p) + 1$, if and only if $p \notin \bullet t$ and $p \in t\bullet$

$M'(p) = M(p)$, otherwise

We denote a sequence of transitions as t^*; if all transitions are labelled τ, we write τ^*. We denote the successive firing of a sequence of transitions t^*, causing a Petri net N with marking M to change into a marking M' as: $(N, M)\ [t^*>\ (N, M')$.

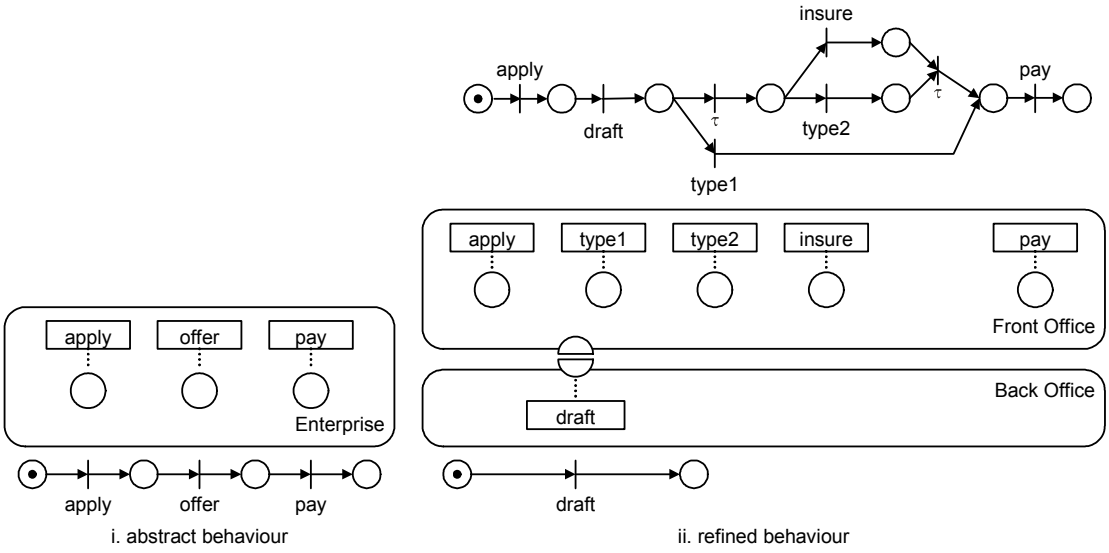


Figure 11. A Behaviour and its Refinement

Figure 11 illustrates our approach to formalizing behaviours by showing two examples. Figure 11.i shows a behaviour, specified using the basic concepts from section 4, that represents a business process of an enterprise. A Petri net represents when an action of the behaviour can occur. To this end a transition is related to an action or interaction contribution with the same name. For Figure 11.i this means that the actions in the business process can occur in sequence. Figure 11.ii shows a refinement of the enterprise from Figure 11.i. The company is refined into two interacting behaviours that represent the front office and the back office. The figure represents that, initially, a client can apply for a loan. Next, the front office interacts with the back office to make a first draft of the loan. After a first draft has been made, the front office can propose two different loans to the client, one of which includes an additional life insurance. After either one of the loans was accepted by the client, the loan is paid out.

5.2. Frequently Occurring Overlap Relations

Two views overlap if they (partly) consider the same properties. If two views overlap they must be equivalent with respect to their overlapping properties. This motivates the definition of a re-usable 'equivalence' relation and the corresponding consistency rule.

We define a re-usable equivalence relation for behaviours. We define this relation using the notion of (weak) bi-similarity [15]. Informally, two processes are bi-similar if, in any state, one process can take the same transitions as the other and can take silent transitions independently. Two transitions are 'the same' if their labels are the same.

Formally, a Petri net K is bi-similar with a Petri net N, denoted $K \sim N$, if and only if there exists a symmetric relation R that relates markings of N to markings of K, in which K can take the same transitions as N and vice versa. Hence R must satisfy that:

- it relates the initial markings of K and N: $(M_N^i, M_K^i) \in R$
  (and, because of symmetry of R: $(M_K^i, M_N^i) \in R$);
- if $(M_X, M_Y) \in R$ and X can take some transition $t \in T_X$, $(X, M_X)[t>(X, M_X')$, then either:
  - it is a silent transition, $l_X(t) = \tau$, and $(M_X', M_Y) \in R$; or
  - Y can take 'the same' transition, $s \in T_Y$, $l_X(t) = l_Y(s)$, possibly preceded and/or succeeded by silent transitions, $(Y, M_Y)[\tau^*>(Y, M_Y^1)$, $(Y, M_Y^1)[s>(Y, M_Y^2)$, $(Y, M_Y^2)\ [\tau^*>(Y, M_Y')$, and $(M_X', M_Y') \in R$.

Figure 12 shows a behaviour, we name it A here, that is equivalent, by bi-similarity, to the behaviour from Figure 11.i, we name it B here. Intuitively, this can easily be seen, because, like Figure 11.i, Figure 12 performs the actions apply, offer and pay in sequence, even though it performs silent transitions in between.

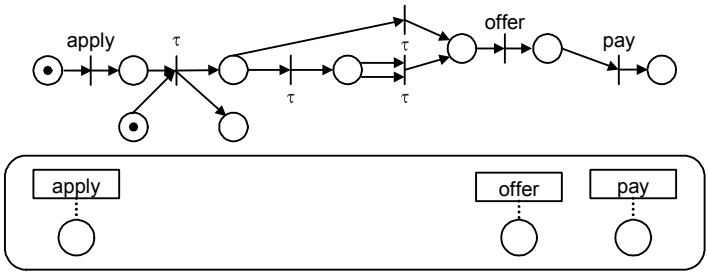


Figure 12. A Behaviour that is Equivalent to Figure 11.i

We use the notion of bi-similarity, rather than the stronger notion of *branching* bi-similarity, which can detect non-equivalence of behaviours with respect to the moment at which a choice is made in a behaviour. We use the weaker notion of bi-similarity, because we use a causality-based technique for specifying behaviour, in which we represent a causality condition for the occurrence of each action or interaction contribution. In such a technique the moment of choice is not a concept. Unlike, for example, UML activity diagrams in which a decision node represents a moment of choice. The drawback of using the bi-similarity notion is that it does not check causal equivalence, which would be desirable when using a causality-based technique. However, to the best of our knowledge, there is no equivalence notion in Petri nets that checks causal equivalence.

5.3. Frequently Occurring Refinement Relations

A view is a refinement of another view, if it contains more detailed information about how the system under design will be implemented. Hence, the refined view must be equivalent to the more abstract view, after we insert the details into the more abstract view; or, vice versa, after we remove the details from the refined view. This motivates the definition of a re-usable refinement relation and corresponding consistency rules.

Focusing on behaviour, we first explain how we allow a behavioural view to be refined. Second, based on these possible refinements, we explain how we can check the consistency of a behavioural view and its refinement.

A behaviour can be refined in one, or a combination, of the following ways:

1. By describing one of its behaviours as a composition of multiple interacting behaviours. As a result of this form of decomposition, activities that were performed by the (single) original behaviour may be performed by the (multiple) refining behaviours. Hence, they are transformed from actions into interactions.
2. By describing a relation between two of its activities as a composition of multiple more fine-grained relations, introducing activities to connect these relations. We refer to these activities as *inserted* activities.
3. By describing one of its activities as a composition of multiple more fine-grained activities. When an activity is refined in this way, some of its refining activities correspond to its completion. We refer to these activities as *final* activities. The other refining activities are inserted activities.

To check consistency we introduce techniques to remove the details that are inserted during refinement. Subsequently, we can check equivalence of the abstract behaviour with the behaviour from which we removed the details. In particular we introduce techniques to: (i) compose multiple interacting behaviours into a single behaviour, composing interactions between those behaviours into actions at the same time; (ii) abstract from inserted actions; and (iii) integrate final actions into a single action. Note that we only defined the abstraction and integration techniques for actions and not for interactions. We can still apply the techniques to interactions, by first composing them into actions. This has the limitation that we cannot check refinement with respect to assigning activities to behaviours. For example, the refinement check would approve of refining an interaction (assigned to multiple behaviours) by an action (assigned to a single behaviour), although we did not define such a refinement as correct in the refinement rules above. Further investigation of this problem and a possible solution is left for future work.

Composing Behaviours. Two behaviours can be composed by integrating their interactions into actions. The resulting behaviour must be equivalent to the original behaviour, with the difference that the interactions are transformed into actions. We achieve this with our formalism, by creating a single transition for each interaction between the composed behaviours. This transition represents the integrated action. The transition can be labelled with the name of one of the interaction contributions

from which it was derived (the choice is arbitrary, because all interaction contributions must have the same name by convention). Incoming flow relations to (transitions representing) contributions of the interaction, become incoming flow relations of the newly created transition. Similarly, outgoing flow relations from contributions of the interaction, become incoming flow relations from the newly created transition. Finally, we can remove the transitions that represent the interaction contributions. As an example, Figure 13 shows the result of composing the behaviours from Figure 11.ii.

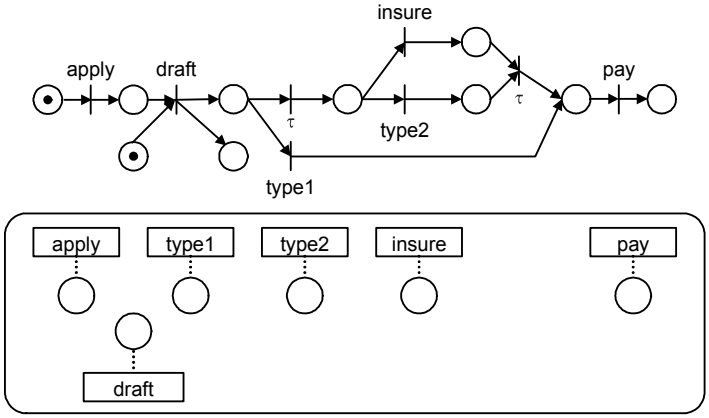


Figure 13. Composition of Figure 11.ii

Abstracting from Inserted Actions. We can abstract from an inserted action by labelling it with the silent label, τ. This means that, in an equivalence check, although the action can still occur, it is not observed; it can occur independently of what happens in the equivalent behaviour. In our case it is not observed, because it exists at a lower level of detail.

Integrating Final Actions. To integrate final actions into an *integrated action*, we need to know how the completion of final actions corresponds to the completion of the original action. For example, the completion of *all* final actions can correspond to the completion of the abstract action, or the completion of *any* of the final actions can correspond to the completion of the abstract action. Therefore, we require that the designer specifies a *completion condition* that represents which of the final actions must have completed, for the integrated action to complete. A completion condition

can use a conjunction, represented by $\wedge$, to represent that all actions in the conjunction must have completed for the abstract action to complete. It can use a disjunction, represented by $\vee$, to represent that any of the actions in the disjunction must have completed for the abstract action to complete. And, it can use combinations of conjunctions and disjunctions. We assume that a completion condition is specified in the disjunctive normal form. An example of a completion condition is: $a_1 \vee (a_2 \wedge a_3)$. This condition represents that the completion of some abstract action corresponds to the completion of final action a_1 or the completion of final actions a_2 and a_3.

To integrate final actions into an integrated action, we create the integrated action and a corresponding transition. We compute the flow relations that represents when the integrated action can occur, by transforming the completion condition into a Petri net as follows. We transform each of the conjunctions into a silent transition. Flow relations to and from transitions in a conjunction become flow relations of that silent transition. In this way each of the silent transitions is enabled when *all* transitions in the conjunction are enabled (corresponding to the semantics of conjunction). Figure 14.i illustrates the transformation of $a_2 \wedge a_3$. Subsequently, we create a place with flows to it from each of the silent transitions that represent the conjunctions. A flow leaves from it to the integrated action. In this way the integrated action is enabled after *any* of the conjunctions is satisfied (corresponding to the semantics of disjunction). Figure 14.ii illustrates the transformation of $a_1 \vee (a_2 \wedge a_3)$ for integrated action a.

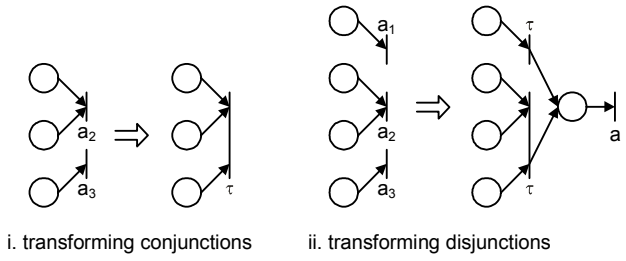


Figure 14. Incoming Flows of an Integrated Action

If an action depends on final actions (there is a path of flows from the final actions to it), then we must replace this dependency by a dependency on the integrated action,

because we will remove the final actions. We can make the replacements by observing that the incoming flows of an action determine a condition for its occurrence. This can easily be seen if we consider that a transition is enabled if there are tokens on *all* its incoming places (conjunctive condition). There are tokens on an incoming place if *any* of the incoming transitions of that place has fired (disjunctive condition). This relation between a Petri net and a causal condition assumes that a token represents that an action has occurred. Therefore, we can only compute the condition for the occurrence of an action, if each transition that is part of the condition has exactly one outgoing flow in the context of that condition. Because, under this assumption an action occurs more than once if its transition produces more than one token. Formally, we can transform the incoming flows of a transition t into a condition, by applying the following function f to a place x in a Petri net:

$f(x) = \wedge\{ f(y) \mid y \in \bullet x \}$, if $x \in T$ (x is a transition)

$= \vee\{ g(y) \mid y \in \bullet x \}$, if $x \in P$ (x is a place)

$g(t) = f(t)$, if $l(t) = \tau$

$= l(t)$, if $l(t) \neq \tau$

Figure 15 shows an example in which we compute the condition for an action b.

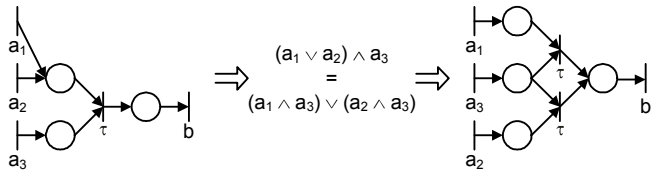


Figure 15. Outgoing Flows of an Integrated Action

We can replace the final actions in this condition as follows:

1. We can replace a *conjunction* of final actions by the integrated action, if the completion condition is also a conjunction of final actions. Because, in case of a conjunction, the condition of the action represents that it can occur if *all* final actions have occurred, while the completion condition represents that the occurrence of the integrated action corresponds to the occurrence of *all* final actions.
2. We can replace a *disjunction* of final actions by the integrated action, if the completion condition is also a disjunction of final actions. Because, in case of a

disjunction, the condition of the action represents that it can occur if *any* of the final actions have occurred, while the completion condition represents that the occurrence of the integrated action corresponds to the occurrence of *any* of the final actions.

3. We can replace a *combination* of conjunctions and disjunctions by applying a combination of rules 1 and 2.

Basically this means that we can replace the part of a condition that is equivalent to the completion condition. To make this replacement easy, we rewrite the condition for the action into the form $A \vee C$, where A is a condition on the final actions in the disjunctive normal form and C is a condition on other actions. If we cannot rewrite a condition into this form, the final actions cannot be replaced. This is the consequence of an refinement that does not conform to our refinement rules, in which a condition that depends on an integrated action is refined by a condition that only depends on a part of that integrated action (or rather: only some of the integrated action's final actions). We refer to [22,3] for more details on replacing final actions in a condition.

After we made the replacement, we can integrate the condition for a transition t back into the Petri net, as follows:

1. remove all places, silent transitions and flows that belong to the original condition of t, excluding t itself and excluding the (transitions that represent) actions in the original condition
2. create a place p
3. create a flow from (p, t)
4. for each conjunction $a_1 \wedge a_2 \wedge \ldots \wedge a_n$:
    - 4.1. create a transition t' labelled τ
    - 4.2. create a flow (t', p)
    - 4.3. for each conjunctive element a that corresponds to a transition t"
        - 4.3.1. find a place p' that represents that a has occurred,
        - 4.3.2. if it does not yet exist:
            - 4.3.2.1. create a place p' that represents that a has occurred
            - 4.3.2.2. create a flow (t", p')
        - 4.3.3. create a flow (p', t)

Figure 15 shows how we transform a condition the condition for the occurrence of *b* back into a Petri net, after we computed the disjunctive normal form of the condition. It remains to be proven that the Petri net after transforming it into the 'disjunctive normal form' is bi-similar to the original Petri net.

After an integrated action and its relation to other actions have been defined, we can remove the final actions, the corresponding transitions and incoming flow relations.

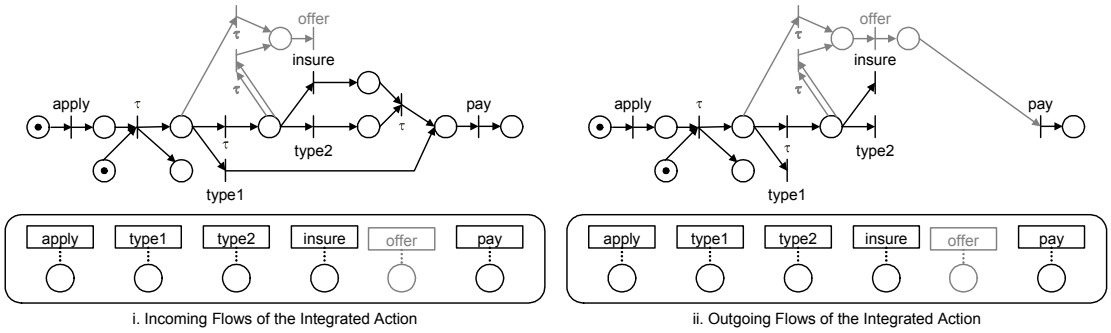


Figure 16. Creation of Integrated Action *offer*

Figure 16 shows how an integrated action, *offer*, can be created from the completion condition $type_1 \vee (type_2 \wedge insure)$. Figure 16.i shows in grey how the integrated action and its incoming flows can be created from the completion condition. Figure 16.i has a silent transition that represents the conjunction $type_1$ and a silent transition that represents the conjunction $type_2 \wedge insure$. Flows leave from these transitions to the place that represents the disjunction $type_1 \vee (type_2 \wedge insure)$. Figure 16.ii shows in grey how the outgoing flows of the final actions can subsequently be replaced by outgoing flows of the integrated action. Only the action *pay* depends on the final actions according to the condition $type_1 \vee (type_2 \wedge insure)$, which is already in the disjunctive normal form and (syntactically) equivalent to the completion condition. Hence, the dependency of *pay* on the final actions can be replaced by the dependency of *pay* on the integrated action *offer*. Finally, removing the final actions from Figure 16.ii yields Figure 12.

Example. Figure 12, Figure 13 and Figure 16 illustrate the process of checking consistency between Figure 11.i and its refinement, Figure 11.ii, if we consider that Figure 11.i is refined by:

1. refining *offer* into *draft*, $type_1$, $type_2$ and *insure*, such that $type_1 \vee (type_2 \wedge insure)$ is the completion condition for *offer*.
2. decomposing *enterprise* into *front office* and *back office*.

To verify consistency, we must reverse the refinement and check equivalence. Hence, we must:

1. compose *front office* and *back office*, resulting in Figure 13;
2. abstract from the inserted action, *draft*;
3. integrate action *offer* from its final actions, by computing the incoming flows of *offer*, resulting in Figure 16.i, computing the outgoing flows, resulting in Figure 16.ii and finally, removing the final actions, resulting in Figure 12.

Figure 12 is equivalent to Figure 16, proving that the refinement is consistent.

6. Case Study

To validate the framework, we applied it to specify consistency rules in the Reference Model for Open Distributed Processing (RM-ODP) [12,11]. We specified consistency rules between the RM-ODP enterprise, computational and information viewpoint. In this paper, we provide an overview of the case study for illustration purposes, focusing on the enterprise and computational viewpoint. The full case study is presented in [3].

Figure 17 shows some of the concepts from the RM-ODP enterprise and computational viewpoints. It also shows how (the concepts of) these viewpoints are related and how they are related to the basic concepts.

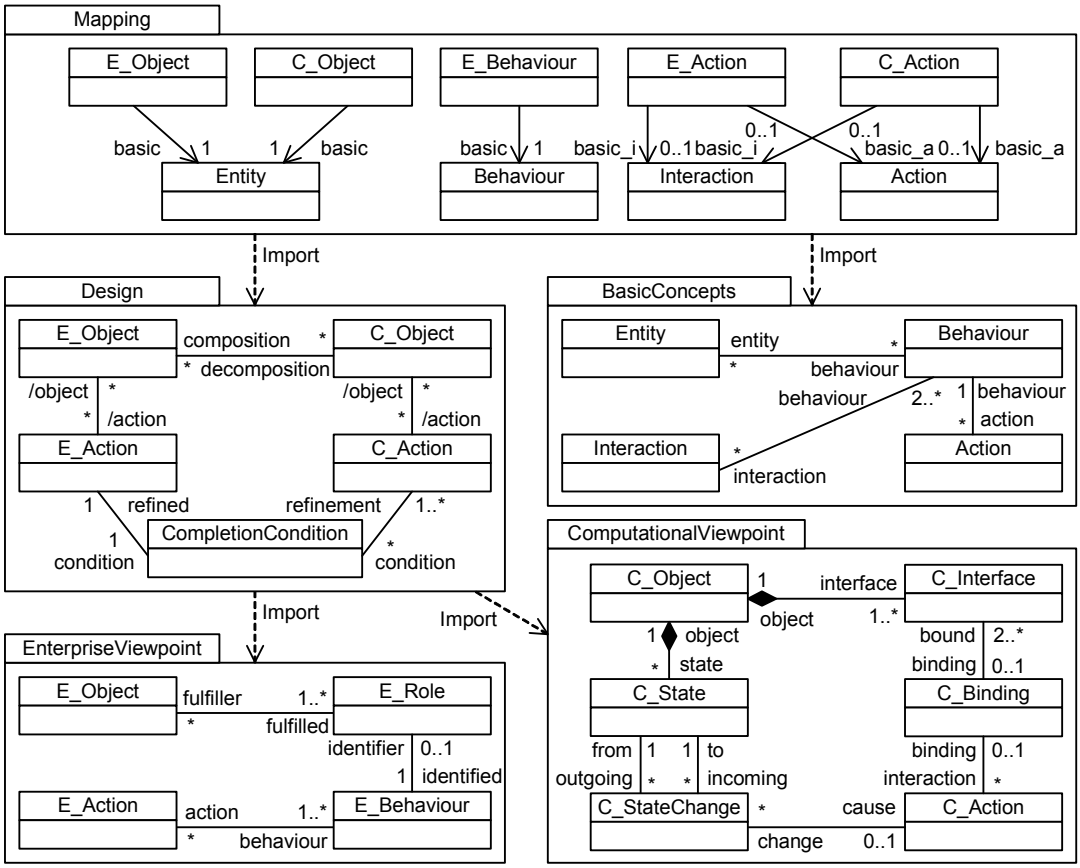


Figure 17. RM-ODP Enterprise and Computational Viewpoints and their Relations

Enterprise Viewpoint. The enterprise viewpoint can be used to specify a system in its enterprise environment. It consists of enterprise objects, which can represent either human actors or applications in the enterprise. The behaviour of the enterprise can be specified using the role-based or the process-based approach or a combination of both approaches. In the process-based approach we specify the enterprise behaviour by means of business processes. In this paper we focus on the role-based approach, in which we specify enterprise behaviour by means of several behaviours identified by roles. An enterprise object can fulfil roles, representing that it has the behaviour identified by those roles. A behaviour contains actions. Each action can be assigned to a single behaviour, representing that it is performed only by (the object that fulfils) that behaviour, or to multiple behaviours, representing that it is an interaction between (the objects that fulfil) those behaviours.

Each enterprise object is mapped onto a basic entity and each enterprise behaviour is mapped onto a basic behaviour. An enterprise action is mapped onto a basic action if it is assigned to a single behaviour, it is mapped onto a basic interaction if it is assigned to multiple behaviours. The mapping keeps track of these relations between enterprise viewpoint and basic concepts.

Computational Viewpoint. The computational viewpoints can be used to specify a decomposition of the system. It consists of computational objects that expose their functionality at interfaces. Interfaces of computational objects can be bound, representing that the objects interact with each other through those interfaces. An object has states. A state represents the condition of an object at a given time that determines the possible sequences of actions that it can perform. If an object performs an action that action causes the state of the object to change. An action can be assigned to interfaces, representing that it is an interaction through those interfaces.

Each computational object is mapped onto a basic entity. The states and state changes of an object constitute a state machine that can be mapped onto a basic behaviour. A computational action is mapped onto a basic interaction if it is assigned to interfaces. The mapping keeps track of the relation between computational objects and basic entities and of the relation between computational actions and basic actions or interactions.

Relations and Consistency Rules. Since the enterprise viewpoint represents the system in its environment and the computational viewpoint represents a decomposition of the system, the relation between the two is that: the computational viewpoint *refines* the part of the enterprise viewpoint that represents the system. Hence, each enterprise object (that represents a system part) and its behaviour can be decomposed into multiple computational objects. Also, each enterprise action can be refined by one or more computational actions. To be able to specify and check these relations, they are part of the design. The design maintains the relation between an enterprise object and the computational objects in which it is decomposed (if an enterprise object is refined by a single computational object we relate it to a single computational object by the decomposition relation). The design also maintains the relation between an enterprise action and its final actions via a completion condition.

We consider all computational actions that are not related to an enterprise action as inserted actions.

We can specify consistency rules that apply to the inter-viewpoint relations, using the mapping onto the basic concepts and the consistency rules that are defined on the basic concepts. We use these consistency rules to express that the computational viewpoint must be a correct refinement of the part of the enterprise viewpoint to which it is related. More precisely, the behaviour of the enterprise objects that are decomposed into computational objects must be equivalent to the behaviour of those computational objects, after:

1. we compose the behaviours of those computational objects;
2. abstract from (inserted) computational actions that *are not* related to enterprise actions; and
3. integrate (final) computational actions that *are* related to enterprise actions via a completion condition.

This procedure to check consistency follows the procedure to check refinement explained in section 5.

7. Conclusions

This paper proposes a framework to help maintain consistency in designs that incorporate viewpoints from different stakeholders. The framework focuses on viewpoints that address behavioural, structural and information concerns. Our framework is based on the hypothesis that the use of a common set of basic design concepts aids in defining relations between viewpoints and rules to check the consistency between views. A common and basic set of concepts represents properties that all stakeholders consider relevant (common) and that are elementary (basic), as opposed to composite properties that can be represented by a composition of elementary properties.

In a case study we show that our framework and our set of basic concepts can be applied to check consistency between views. We show this by applying the frame-

work to define consistency rules between the RM-ODP enterprise, computational and information viewpoints.

Currently we only use the basic concepts to define consistency rules that apply to behavioural aspects. Hence, one could argue that using a behavioural formalism instead, is just as effective and less cumbersome. However, it can easily be seen that our approach can be expanded to integrate formalisms and re-usable consistency rules for other aspects as well. The benefit of using the basic concepts is that they relate the various formalisms and hide them from the designer, such that the designer does not have to know the details of the formalisms. Hence, a direction for future work is to define consistency rules with respect to other aspects than behaviour.

The framework can only aid in specifying consistency rules on concepts that can be mapped onto the basic concepts. For that reason it is important to evaluate the basic concepts with respect to their ability to represent existing viewpoints and adapt them to accommodate those viewpoints.

In section 5.3 we made the correctness of our operations to check refinement intuitively clear. We also tested them by applying them to typical cases of refinement. For some operations a formal proof of its correctness would strengthen our case. For example, we could produce a formal proof that the behaviour of two interacting behaviours is equivalent (bi-similar) to the behaviour of their composition. We could also produce a formal proof that a Petri net after pre-processing is equivalent to the Petri net before pre-processing. These proofs remain for future work.

Acknowledgements

The authors thank Paul Grefen for his comments on an earlier version of this paper.

References

1. Boiten, E. A., Bowman, H., Derrick, J., Linington, P. F., & Steen, M. W. A. (2000). Viewpoint consistency in ODP. Computer Networks, 34(3), 503-537.

2. Boiten, E., Derrick, J., Bowman, H., & Steen, M. W. A. (1999). Constructive consistency checking for partial specification in Z. Science of Computer Programming, 35(1), 29-75.
3. Dijkman, R.M. (2006). Consistency in Multi-Viewpoint Architectural Design. Ph.D. Thesis. University of Twente, Enschede, The Netherlands.
4. Dijkman, R.M. (2005). YATL4MDR: A Model Transformation Engine and OCL Checker for the NetBeans Meta-data Repository. Available at: wwwhome.cs.utwente.nl/~dijkman/YATL4MDR.html.
5. Eck, P. A. T. van, Blanken, H. M., & Wieringa, R. J. (2004). Project GRAAL: Towards operational architecture alignment. International Journal of Cooperative Information Systems, 13(3), 235-255.
6. Eertink, H., Janssen, W., Oude Luttighuis, P., Teeuw, W., & Vissers, C.A. (1999). A Business Process Design Language. In: Proceedings of the World Congress on Formal Methods (FM) (Vol. 1709 in Lecture Notes in Computer Science, pp. 76-95).
7. Ferreira Pires, L. (1994). A framework for distributed systems development. Ph.D. Thesis. University of Twente, Enschede, The Netherlands.
8. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency handling in multi-perspective specifications. IEEE Transactions on Software Engineering, 20(8), 569-578.
9. Glabbeek, R.J. van, & Weijland, W.P. (1996). Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM, 43(3), 555-600.
10. IEEE. (2000). IEEE recommended practice for architectural description of software-intensive systems (IEEE Std No. 1471-2000).
11. ITU-T, & ISO/IEC. (1999). Information technology - open distributed processing reference model - enterprise language (ITU-T Specification 911 and ISO/IEC Specification 16414).
12. ITU-T, & ISO/IEC. (1995). Open distributed processing reference model (ODP-RM) (ITU-T Specification 901.4 and ISO/IEC Sprecification 10746-1.4).
13. Jonkers, H., Lankhorst, M. M., Buuren, R. van, Hoppenbrouwers, S., Bonsangue, M., & Torre, L. van der (2004). Concepts for modelling enterprise architectures. International Journal of Cooperative Information Systems, 13(3), 257-287.
14. Lankhorst, M. M. (2005). Enterprise architecture at work: Modelling, communication and analysis: Springer.

15. Milner, R. (1989). Communication and Concurrency, Prentice Hall, London, United Kingdom.
16. Naumenko, A. (2002). Triune continuum paradigm: A paradigm for general system modeling and its applications for UML and RM-ODP. Ph.D. Thesis. École Politechnique Fédérale de Lausanne, Switzerland.
17. Nuseibeh, B., Kramer, J., & Finkelstein, A. (1994). A framework for expressing the relationships between multiple views in requirements specification. IEEE Transactions on Software Engineering, 20(10), 760-773.
18. Object Management Group. (2003). UML 2.0 OCL specification (Final Adopted Specification No. ptc/03-10-14).
19. Object Management Group. (2002). Meta object facility (MOF) specification (Available Specification No. formal/02-04-03).
20. Quartel, D., Ferreira Pires, L., & Sinderen, M. van (2002). On architectural support for behavior refinement in distributed systems design. Journal of Integrated Design and Process Science, 6(1).
21. Quartel, D. (1998). Action relations - basic design concepts for behaviour modelling and refinement. Ph.D. Thesis. University of Twente, Enschede, The Netherlands.
22. Quartel, D., Ferreira Pires, L., Sinderen, M. J. van, Franken, H. M., & Vissers, C. A. (1997). On the role of basic design concepts in behaviour structuring. Computer Networks and ISDN Systems, 29(4), 413-436.
23. Sinderen, M. J. van (1995). On the design of application protocols. Ph.D. Thesis, University of Twente, Enschede, The Netherlands.
24. Wieringa, R. J., Blanken, H. M., Fokkinga, M. M., & Grefen, P. W. P. J. (2003). Aligning application architecture to the business context. In J. Eder & M. Missikoff (Eds.), Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE) (Vol. 2681 in Lecture Notes in Computer Science, pp. 209-225).
25. Wegmann, A. (2003). On the systemic enterprise architecture methodology (seam), Proceedings of the 5th International Conference on Enterprise Information Systems - Volume III (pp. 483-490).