

A Rigorous Approach to Relate Enterprise and Computational Viewpoints

Remco M. Dijkman Dick A.C. Quartel Luís Ferreira Pires Marten J. van Sinderen
University of Twente, Centre for Telematics and Information Technology
{r.m.dijkman|d.a.c.quartel|l.ferreirapires|m.j.vansinderen}@utwente.nl

Abstract

Multi-viewpoint approaches allow stakeholders to design a system from stakeholder-specific viewpoints. By this, a separation of concerns is achieved, which makes designs more manageable. However, to construct a consistent multi-viewpoint design, the relations between viewpoints must be defined precisely, so that the consistency of designs from these viewpoints can be verified. The goal of this paper is to make the consistency rules between (a slightly adapted version of) the RM-ODP enterprise and computational viewpoints more precise and to make checking the consistency between these viewpoints practically applicable. To achieve this goal, we apply a generic framework for relating viewpoints that includes reusable consistency rules. We implemented the consistency rules in a tool to show their applicability.

1. Introduction

Multi-viewpoint approaches are often used to cope with the complexity of distributed systems design. In such approaches different stakeholders design the system from their own perspective, or *viewpoint*. By doing this, they achieve separation of concerns and break up the overall design into smaller and more manageable parts. The implementation of the distributed system must be consistent with each of the viewpoint designs. However, to be able to build a system that is consistent with each of the viewpoint designs, the viewpoint designs must also be mutually consistent.

The Reference Model for Open Distributed Processing (RM-ODP) [13, 14] prescribes a multi-viewpoint approach. Specifically, it prescribes the use of five viewpoints: enterprise, information, computational, engineering and technology. RM-ODP prescribes an abstract design language for each of the five viewpoints. It also prescribes consistency rules to maintain the consistency between viewpoint designs. In this paper we focus on the enterprise and computational viewpoints and the consistency rules between these two. We slightly

adapt the original RM-ODP viewpoints, to make a strict distinction between behaviour and structure and to make some concepts more concrete. However, we clearly state where our enterprise and computational viewpoints deviate from the corresponding RM-ODP viewpoints.

The goal of this paper is twofold. Firstly, we aim to make the consistency rules between the enterprise and computational viewpoints precise and to make checking consistency between designs from these viewpoints practically applicable. Secondly, we aim to evaluate the generic framework for relating viewpoints that we proposed in [6]. We evaluate it, by applying it to relate our enterprise and computational viewpoints. To show that our way to relate viewpoints is practically applicable, we implemented it in a prototype tool. The main contribution of our work is that we describe the relation between the enterprise and computational viewpoints precisely (section 6) and that we developed a tool to enforce this relation [22].

The remainder of this paper is organized as follows. Section 2 explains the generic framework from [6], which we use to relate the enterprise and computational viewpoints. Section 3 briefly describes a slightly adapted version of the RM-ODP basic modelling concepts, which we use to define the enterprise and computational viewpoints. Section 4 explains basic (consistency) relations that can exist between viewpoints. Section 5 introduces our enterprise and computational viewpoints and shows how these viewpoints are defined in terms of the basic modelling concepts. Section 6 explains the (consistency) relation between the enterprise and computational viewpoints. These relations are based on the basic viewpoint relations from section 4. Section 7 gives an example of an enterprise and computational viewpoint design. Also, it illustrates how the consistency between these views can be assessed. Finally, section 8 presents related work and section 9 the conclusions.

2. Generic framework for relating viewpoints

According to the generic framework from [6] a *viewpoint* prescribes the means to construct a design that addresses the concerns of a particular stakeholder at a

particular level of detail. We call the design from a viewpoint a *view*. A viewpoint prescribes a set of concepts that a designer can use to construct a design. A *concept* is an abstraction of some common and essential properties of a system, such as: remote procedure call, distributable object and interface.

Since all viewpoints deal with the same system, they are related to each other. In [6] we distinguish two basic viewpoint relations: the *complement relation* and the *refinement relation*. The complement relation exists between viewpoints that deal with different, partly overlapping, concerns at the same level of detail. The refinement relation exists between viewpoints that deal with the same concerns but regard these concerns at different levels of detail. Two views must be consistent with each other according to the relation that their viewpoints have. To verify consistency between two views, we defined *consistency rules* that match the relation between these views. If the consistency rules hold between the views we say that the views are consistent with each other. For example, if a viewpoint *A* has a refinement relation to another viewpoint *B* and two views *a* and *b* exist that are constructed from viewpoint *A* and *B* respectively, then we apply refinement consistency rules to verify if view *a* is a correct refinement of view *b*. In this paper we focus on the refinement relation and the refinement consistency rules, because we found that this relation applies to the enterprise and computational viewpoints. Section 4 explains our notion of refinement and the basic forms of refinement that we distinguish. A refinement relation between two viewpoints can consist of one or more basic forms of refinement. Consequently, the consistency between two views can be verified using one or more refinement consistency rules. Section 6 explains the refinement relation that exists between the enterprise and computational viewpoints.

One way to verify the consistency between two views is via a basic viewpoint on which the basic viewpoint relations are defined as well as the consistency rules. Figure 1 illustrates this approach. The approach requires that mappings exist from each viewpoint to the basic viewpoint. If such mappings exist, we can verify the consistency between two views, *a* and *b*, constructed from viewpoint *A* and *B* respectively, by mapping them to view *a'* and *b'*, which are constructed from the basic viewpoint. The consistency rules, which are defined in the basic viewpoint, can then be used to verify the consistency between *a'* and *b'*. If *a'* and *b'* are consistent, we infer that *a* and *b* are also consistent. We define a meta-model for each of the viewpoints and meta-model transformations between the viewpoints. We do that to make it possible to define mappings precisely and to facilitate the development of tools to automate the mappings.

The benefit of the approach is that all viewpoints that have a mapping to the basic viewpoint can reuse the viewpoint relations and consistency rules from the basic viewpoint. However, our approach only pays off when the benefit of reuse outweighs the cost of defining the mappings.

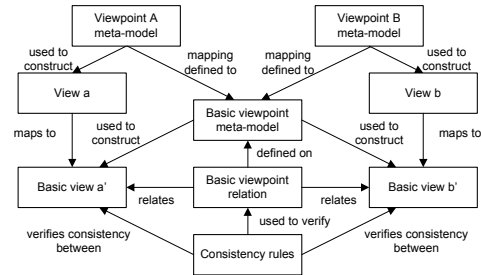


Figure 1. Relate views via a basic viewpoint

We claim that the basic viewpoint approach to relate viewpoints is particularly suitable for RM-ODP, because RM-ODP already prescribes a set of basic modelling concepts [14 (part 2, clause 8)] and defines its viewpoint concepts in terms of these basic modelling concepts. Therefore, we can use the basic modelling concepts as a basic viewpoint. The mappings between the RM-ODP specific viewpoints and the basic viewpoint can be based on the definition of the viewpoint concepts in terms of the basic modelling concepts.

3. Basic modelling concepts

We make a distinction between structural concepts and behavioural concepts. We can use structural concepts to describe the things that exist and where these things exist in space and time, while we can use behavioural concepts to describe how things behave.

We focus on the design of a system at a particular moment in time. This is also called a snapshot of a system. Hence, we do not consider dynamic changes in the structure of a system over time, while the RM-ODP concepts do address this concern. We leave this as future work.

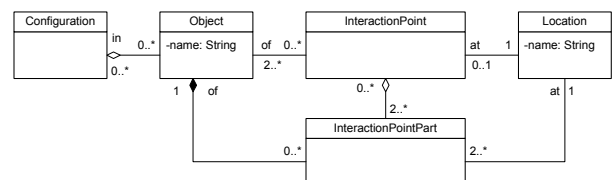


Figure 2. Meta-model of the basic structural concepts

3.1. Basic structural concepts

Figure 2 presents the meta-model for our basic structural concepts. We further explain the concepts from the meta-model in the remainder of this subsection.

An object represents a concrete or abstract thing of interest, such as a ‘CORBA object’ or a ‘server’. We associate an object with a name that identifies it uniquely. Objects can be grouped in a configuration.

An interaction point represents a shared logical or physical mechanism through which two or more objects can interact, such as a ‘programming interface’ or a ‘computer network’. An interaction point does not only exist *between* objects, but is also *part of* the objects. A ‘computer network’ is a good example of an interaction point, since it consists of a ‘network cable’ and ‘network cards’, which are part of the interacting ‘computers’.

We introduce the interaction point part concept to represent the (potential) participation of an object in an interaction point. Each interaction point is partitioned into the interaction point parts of its participating objects. Moreover, an object/interaction point pair uniquely identifies an interaction point part.

RM-ODP prescribes that an interaction point exists during a time interval and at a location in (logical) space. In our design approach, the structural design that contains an interaction point is valid for a particular interval in time, because this design is a snapshot of the system structure. Hence, an interaction point that is used in a structural design is implicitly associated with a time interval. Therefore, we only associate an interaction point with a location. To represent interaction points that change over time, we have to draw different snapshots that represent different time intervals. The drawback of this approach is that it is hard to represent the relation between a time interval and the availability or location of an interaction point. The concern that addresses the dynamism of the structure of the system should address this issue.

Our notion of interaction point differs slightly from the notion that RM-ODP uses, because RM-ODP does not distinguish between interaction points and interaction point parts. Rather than combining interaction point parts into interaction points to allow objects to interact, RM-ODP states that objects interact at (bound) interfaces. However, we reserve the term interface for representing behavioural aspects (consistently with its definition in RM-ODP) and introduce the interaction point part concept as its structural counterpart. Therefore, we also say that objects interact at their interaction points or interaction point parts rather than at their interfaces.

Figure 3 shows our notation for the basic structural concepts. We assume that interaction points are uniquely identified by the location at which they exist and that

interaction point parts are uniquely identified by the location of the interaction point to which they belong and the name of the object of which they are a part.

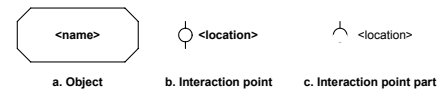


Figure 3. A notation for basic structural concepts

3.2. Basic behavioural concepts

Figure 4 presents the meta-model for our basic behavioural concepts. We further explain the concepts from the meta-model in the remainder of this subsection.

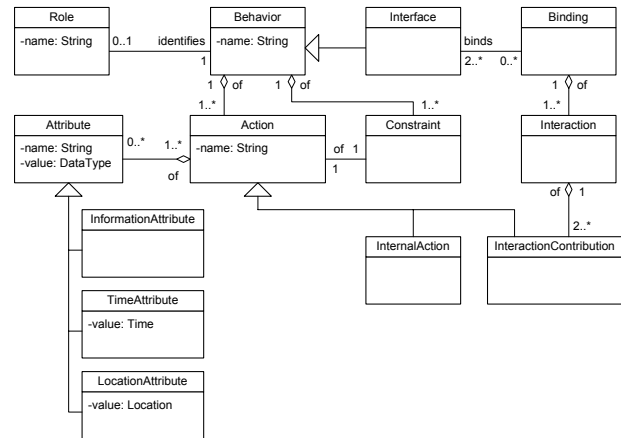


Figure 4. Meta-model of the basic behavioural concepts

A behaviour consists of a collection of actions and constraints on when these actions may occur. An action can either be assigned to a single behaviour, in which case we refer to it as an internal action, or it can be part of an interaction that is shared between behaviours, in which case we refer to it as an interaction contribution.

To associate a behavioural semantics with the basic modelling concepts, we have to specify the form that constraints take. We express constraints by associating each action with a condition (or constraint) for its occurrence. In this way each constraint affects the occurrence of exactly one action, much like a precondition. We specify the constraint of an action’s occurrence as a condition on the (non-)occurrence of other actions. For example, if action *a* is allowed to occur after *b*, we say that the constraint for *a* is *the occurrence of b*. If there is a choice between *a* and *b*, we say that the constraint for *a* is *the non-occurrence of b* and the constraint for *b* is *the non-occurrence of a*. By associating

each action with a constraint in this way, we get a *causal automaton* as explained in [11]. We defined a textual notation to express constraints. For the occurrence of an action we use its name, for the non-occurrence we prefix the name with the negation (\neg) symbol. For the conjunction and disjunction of (non-)occurrences, we use the corresponding logical symbols (\wedge and \vee). For example, if the constraint for *a* is *the occurrence of b and the non-occurrence of c*, we express this as: $b \wedge \neg c \rightarrow a$. A special constraint ‘always’ indicates that an action can always occur. [19, 21, 22] give a more detailed account of behaviour specification with this technique, along with a graphical notation. It also explains how the constraints can be extended with constraints on the time at which an action occurs, the information that is established in an action and the location at which an action occurs.

This way of specifying relations between actions allows for a fully concurrent time model. Actions that are not causally related are independent and can therefore happen concurrently. If an action is enabled by another action it must happen after that action. If an action is disabled by another action it can happen before, but not after (or at the same time as) that action. Hence, at run-time mechanisms must be implemented that make actions aware of the (non-)occurrence of actions to which they are causally related.

Interactions are partitioned into interaction contributions in the same way as interaction points are partitioned into interaction point parts. The constraints on an interaction are specified as constraints on its interaction contributions. In this way, each behaviour that contributes to an interaction can specify its own constraints on when the interaction can occur.

An interface is a particular type of behaviour. It represents a subset of the behaviour of an object that is intended to be performed at a particular interaction point (part). Interfaces can be bound to allow objects to interact. Therefore, we define interactions in the context of a binding. A binding maps onto a single interaction point and vice-versa and an interface maps onto a single interaction point part and vice versa.

A role is an identifier for a behaviour. Hence, it is not a behaviour itself, but it refers to one.

An action occurs at a time moment and a location. In addition to this, RM-ODP states that interactions convey information. However, we generalize this definition and say that interactions *establish* information. The difference between establishing information and conveying information is the following. In case of conveying information one interacting partner has already picked a value that must be accepted by the other partner. In case information is established, neither of the partners may have picked a value yet. Instead they establish a value in a negotiation (interaction), while abstracting from how this

negotiation takes place. We say that internal actions also establish information, while RM-ODP does not prescribe this. To represent these properties of actions, we assign an information, a time and a location attribute to each internal action and interaction contribution. The information attribute represents the information values that can be established in the action, the time attribute represents the time at which the action can occur and the location attribute represents the locations at which the action can occur. When an action occurs, its attributes are associated with values that represent the information established and the time and location at which the action has occurred. All attributes are associated with a type that defines the range of values that an attribute may get. The type of the information attribute can be freely defined by the designer. The type of the time attribute is fixed and the type of the location attribute is the location concept from the structural meta-model.

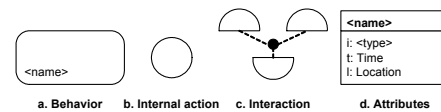


Figure 5. A notation for basic behavioural concepts

Figure 5 shows our notation for the basic behavioural concepts. An interaction contribution is represented by a circle segment of an interaction. In case an interaction is formed by only two interaction contributions, the black dot in the middle may be left out. The attributes and the name of an action are associated with this action using a comment box, such as the one shown in figure 5.d.

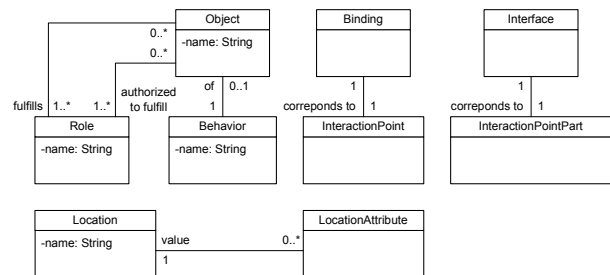


Figure 6. The relation between the structural and behavioural meta-model

3.3. Relation between structure and behaviour

Figure 6 shows the relations between the structural and behavioural meta-models from the previous subsections.

Each object is associated with its behaviour, while a behaviour may or may not be associated with an object.

An object can be authorized to fulfil one or more roles. At a moment in time it can fulfil one or more of the roles for which it is authorized. If a role is assigned to an object, that object must observe the behaviour of the role. Hence, the behaviour of an object must satisfy all roles that the object fulfils. The location attribute of an action in a behavioural design refers to a location in the corresponding structural design. This location corresponds to an interaction point or an interaction point part. Finally, each interface maps onto a single interaction point part and vice versa. Similarly, each binding maps onto a single interaction point and vice versa.

4. Basic viewpoint relation: refinement

In line with the distinction between structural and behavioural concepts, we distinguish between structural and behavioural refinement. Figure 7 shows the different forms of structural and behavioural refinement.

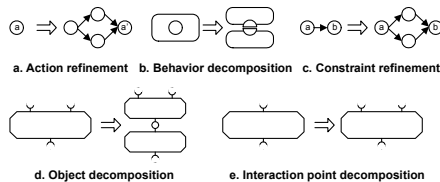


Figure 7. Different forms of refinement

We distinguish three basic forms of behavioural refinement: action refinement, behaviour decomposition and constraint refinement. In case of *action refinement* an action is refined into multiple actions. These actions achieve the same result, but represent a more detailed account of how that result is achieved. Figure 8.a shows an example of action refinement. When an action is refined, some of the actions from the refinement coincide with the completion of the original action. We call these actions *reference actions*. In the example, ‘*send welcome letter*’ and ‘*verify client’s status*’ are both reference actions for the original action ‘*process new client*’, because the original action completes if both these actions complete. Actions from the refinement that do not coincide with the completion of the original action are *inserted actions*. In the example, ‘*enter client details*’ is an inserted action. In case of *behaviour decomposition* a behaviour is split up into two or more communicating behaviours. As a consequence internal actions of the original behaviour can be refined into interactions of the communicating behaviours. Figure 8.b shows an example of behaviour decomposition. In the example, actions ‘*request*’ and ‘*respond*’ are refined into interactions. In case of *constraint refinement* a constraint is split up into two or more constraints that are related by actions that they have in common. These actions are again inserted

actions, because they do not coincide with the completion of any of the original actions. Figure 8.c shows an example of constraint refinement. In the example, the action ‘*send data to subsystem*’ is inserted as the result of refining the constraint of the ‘*process data*’ action.

We distinguish two basic forms of structural refinement: object decomposition and interaction point decomposition. In case of *object decomposition* an object is split up into two or more objects. As a consequence, the behaviour of the object must also be decomposed into communicating behaviours of its component objects. In case of *interaction point decomposition* an object is refined by decomposing its interaction points or interaction point parts into two or more interaction points or interaction point parts. As a consequence, the interactions that occurred at the original interaction point either have to be assigned to one of the interaction points from the decomposition, or have to be refined, after which the resulting interactions can be assigned to different interaction points from the decomposition.

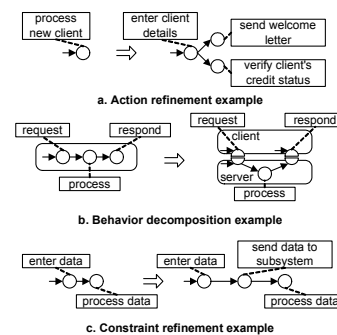


Figure 8. Examples of refinement

A viewpoint *A* can have a refinement relation with another viewpoint *B* using one or a combination of the forms of refinement mentioned above. A view *a* from viewpoint *A* then is consistent with a view *b* from viewpoint *B* if and only if *a* can be reached from *b* by applying the rules that relate *A* to *B*. For example, if *A* relates to *B* through behavioural decomposition, then *a* is consistent with *b* if *a* can be obtained by decomposing *b*.

To verify consistency in this way, we have to know exactly which rules to apply and in which way and which order we have to apply them. However, designers do not like to be bound by rules when refining a design. They rather allow themselves complete freedom when constructing a ‘refined’ design. Hence, we do not know how the rules were applied to refine the design. Since there are many ways in which they *can* be applied, it is not feasible to verify consistency if we do not have this information. However, each time we apply a refinement rule, we add information to the design and there is only one way to remove this information again. Therefore, we

use abstraction instead of refinement to verify the consistency between a design and its refinement. Abstraction rules define how to remove the added information from a design. After this has been done, we can verify if the resulting design is equivalent to the design before refinement.

We say that two behaviours are equivalent if each action from one appears in the other and the conditions of corresponding actions are equivalent. We call this strong equivalence. The benefit of this notion of equivalence is that it preserves causal relations between actions. The drawback is that it very strict. In contrast, other, less strict, notions of equivalence exist [10]. Some of these notions can be used via the semantics of our basic behavioural concepts in terms of Petri-nets [15] and partial orders [19]. The suitability of the different notions of equivalence in the context of our basic concepts requires further study.

The abstraction rules are the inverse of the refinement rules. For example, if viewpoint A relates to viewpoint B through behavioural decomposition, then a is consistent with b if we can apply the inverse of decomposition, namely composition, to a and end up with a design a' that is equivalent to b . This approach to consistency verification requires us to define abstraction rules, as the inverse of the refinement rules above. We claim that, if we enforce the rule that abstraction can only be applied to actions or behaviours that exist, the order in which the abstraction rules are applied, is automatically the right order (i.e. the same order in which the refinement rules were applied). Hence, the designer does not have to keep track of the order in which he applies the refinement rules. To prove this claim, we must show that, if two refinement rules are applied in a particular order, either the corresponding abstraction rules must necessarily be applied in reverse order, or the refinement rules could have been applied in another order as well. A detailed proof is left out due to space limitations.

We define the *behaviour composition* rule as the inverse of the behaviour decomposition rule. The composition rule composes the specified behaviours into a single behaviour and composes the interactions between these behaviours into internal actions. The condition of a composed internal action is the conjunction of the conditions of the interaction contributions of which it is composed. For example, because the conditions of the two contributions to the 'respond' interaction from figure 8.b are *always* and *'process'*, the condition for this action in the composition is $always \wedge 'process'$, which equals *'process'*.

We use the *action abstraction* rule as the inverse of the constraint refinement rule. The action abstraction rule removes the specified inserted actions ($a1, a2, \dots$) from a design. Also, it changes the conditions of actions that

depend on $a1, a2, \dots$, such that where $a1, a2, \dots$ appears in a condition it is replaced by *the condition of $a1$, the condition of $a2, \dots$* . For example, in figure 8.c *'send data to subsystem'* is an inserted action. The condition of *'process data'* is *'send data to subsystem'*, and the condition of *'send data to subsystem'* is *'enter data'*. Therefore, when we remove the inserted action *'send data to subsystem'*, the condition of *'process data'* becomes the condition of *'send data to subsystem'*, which is *'enter data'*.

We use two rules as the inverse of action refinement: the action abstraction and *action integration* rule. Action abstraction abstracts from the specified inserted actions in an action refinement. Action integration integrates the specified reference actions from the action refinement into a single action. The condition of an integrated action depends on the way in which its reference actions correspond to the completion of the original action. Either the completion of all reference actions corresponds to the completion of the original action (*conjunctive reference actions*) or the completion of any of them (*disjunctive reference actions*). A combination of conjunctive and disjunctive reference actions is also possible. The designer must specify how the completion of reference actions corresponds to the completion of the original action in a *completion condition*. The condition of the integrated action then corresponds to the completion condition, where the reference actions are replaced by their conditions. As an example, consider the reference actions *'send welcome letter'* and *'verify client's credit status'* from figure 8.a. The completion of both of these actions coincides with the completion of the original action *'process new client'*. Therefore, we say that the completion condition is $'send welcome letter' \wedge 'verify client's credit status'$. If we integrate these actions into a single action, *'integrated'*, the condition of *'integrated'* is $\langle \text{the condition of 'send welcome letter'} \rangle \wedge \langle \text{the condition of 'verify client's credit status'} \rangle$. As the condition of both actions is *'enter client details'*, the condition of *'integrated'* is $'enter client details' \wedge 'enter client details'$, which equals *'enter client details'*. As another example, suppose that an action *'deliver package'* was refined into the actions *'deliver by airmail'* and *'deliver by sea mail'*. Judging by the names of the actions, the occurrence of either one of them corresponds to the completion of the original action, such that the completion condition is $'deliver by airmail' \vee 'deliver by sea mail'$. Hence, the condition of the integrated action becomes $\langle \text{the condition of 'deliver by airmail'} \rangle \vee \langle \text{the condition of 'deliver by sea mail'} \rangle$.

We defined the abstraction rules as operators that can be applied to basic viewpoint designs. Hence, if we focus on behaviour, we have the following design operators.

abstract: $\wp Action \times Behaviour \rightarrow Behaviour$, where $abstract(\{a1, a2, \dots\}, b)$ returns the behaviour in which the inserted actions $a1, a2, \dots$ are abstracted from.

integrate: $\wp (Condition \times Action) \times Behaviour \rightarrow Behaviour$, where $integrate(\{(c1, a1), (c2, a2), \dots\}, b)$ returns the behaviour in the reference actions from the completion conditions $c1, c2, \dots$ are integrated and named $a1, a2, \dots$ respectively.

compose: $Behaviour \times Behaviour \rightarrow Behaviour$, where $compose(b1, b2)$ returns a single behaviour in which interactions between $b1$ and $b2$ are composed into internal actions.

\sim : $Behaviour \times Behaviour \rightarrow Boolean$, where $b1 \sim b2$ returns true if and only if $b1$ and $b2$ are behaviourally equivalent.

The theory that underlies our notion of refinement and the operators explained above is further explained in [19, 20].

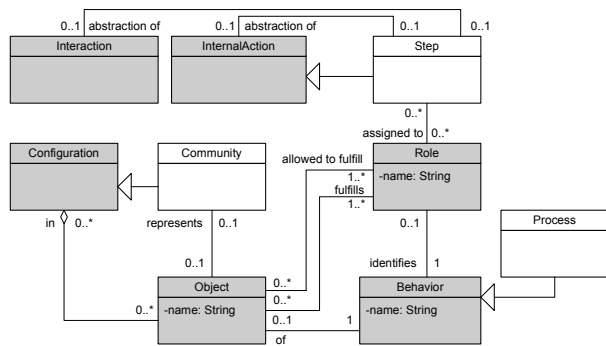


Figure 9. Meta-model of enterprise viewpoint concepts

5. Enterprise and computational viewpoint concepts

We define the enterprise and computational viewpoints as extensions of the basic concepts from section 3.

5.1. Enterprise viewpoint concepts

The enterprise viewpoint is used to design the relation of a system to its environment. The *system* and its *environment* form a *community*, which is a configuration of objects that is formed to meet an objective. A system is a special object, while the environment of the system consists of all objects that are not part of the system. A system can participate in more than one community.

Each community is further defined in terms of an *enterprise contract* that constrains how the objects in that community collaborate. To this end an enterprise contract states the objective of the community, the intended structure and behaviour of the community and policies

that govern the structure and behaviour of the community. In this paper we focus on the intended structure and behaviour of the community.

There are two complementary approaches to specify the intended behaviour of a community: the *role-based approach* and the *process-based approach*. These approaches may both be used in the design of a community's behaviour. In the role-based approach several behaviours are defined in the community, each of which is identified as a role. Objects can participate in the community by fulfilling one or more of the roles. If an object fulfils a role, it must satisfy the behaviour that the role identifies. In the process-based approach, the behaviour of a community is defined in terms of processes. A process is a collection of steps taking place in a pre-described manner and leading to an objective. A step is an abstraction of an action or interaction that may leave the objects that participate in it unspecified. We interpret the process and step concept in terms of basic concepts by considering a process as a (special case of) behaviour that only consists of internal actions. These internal actions are the steps of the process. We can assign a step to the role that performs it or the roles that perform it in collaboration. Subsequently, we can associate the role to an object, such that the object performs the step. Figure 9 shows the meta-model that is consistent with the observations above. We did not include the enterprise contract concept in the meta-model, because an enterprise contract is completely defined by its parts (structure, behaviour, policies and objective). Figure 9 includes some of the basic modelling concepts. These concepts are shown in grey. The figure only shows the basic modelling concepts insofar as they are needed for the understanding of the enterprise viewpoint concepts. However, all basic modelling concepts from section 3 can be used to construct a design from the enterprise viewpoint.

The meta-model shows that the enterprise concepts that we introduced can all be interpreted as basic concepts at a more generic level: a process is a behaviour, a step in a process is an internal action and a community is a configuration. Hence, an enterprise view can be transformed into a basic view by interpreting enterprise specific concepts as their generic counterparts. After transformation, the resulting basic enterprise design can be used for comparison with other (transformed) basic views.

Since the transformation of an enterprise view into a basic view relies on generalization, certain information is lost in the transformation. This is because the generic concepts do not support all information of the specific concepts. For example, the relation between steps and the roles that perform them is lost, because the generic

counterpart of the step concept, the internal action concept, has no relation with the role concept.

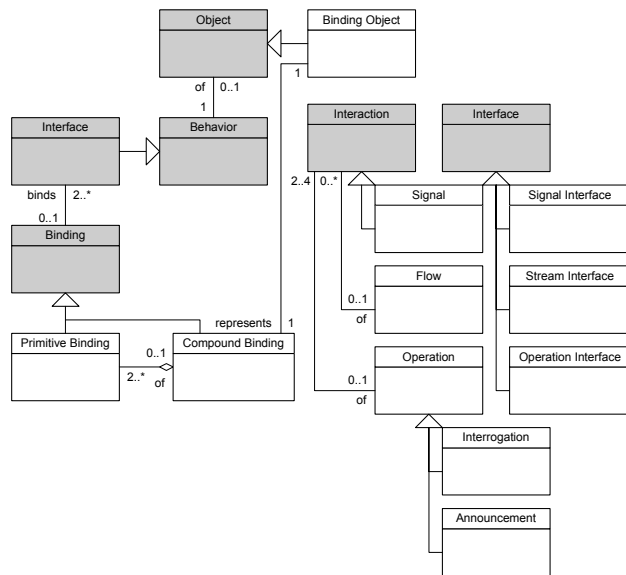


Figure 10. Meta-model of computational viewpoint concepts

5.2. Computational viewpoint concepts

The computational viewpoint is used to specify the logical and functional decomposition of a system into interacting objects. The computational objects that comprise the system are represented in a configuration.

In the RM-ODP computational viewpoint, objects are connected, or bound, via their interfaces. This suggests that interfaces and bindings represent the structural aspect of connections between objects, although interfaces are defined as purely behavioural. Indeed, specific computational specification languages, such as [1, 4, 18] use interfaces and bindings in a structural design to represent connections between objects. However, we want to maintain the strict distinction between structural and behavioural aspects that we introduced in the presentation of the basic concepts. Therefore, we reserve the interaction point concept to represent connections between objects and the interface and binding concepts to represent the behavioural aspects of this connection.

The computational viewpoint distinguishes between *primitive* and *compound* bindings. A primitive binding directly connects two objects (via an interaction point), while a compound binding connects two or more objects via an intermediate object, which is called the binding object. A compound binding can be defined in terms of primitive bindings between the objects that participate in the compound binding and the binding object.

Finally, the computational viewpoint defines three specific kinds of interactions, *signals*, *flows* and *operations*, as well as the corresponding kinds of interfaces *signal interfaces*, *stream interfaces* and *operation interfaces*, respectively. A signal is an atomic interaction between two objects. A flow is an abstraction of a sequence of interactions between two objects. An operation is a request/response mechanism, where a request, called an *invocation*, is sent from one object to another and an optional response, called a *termination*, to this request is sent in the opposite direction as a result. If an operation consists of only an invocation, we call it an *announcement*. If it consists of both an invocation and a termination, we call it an *interrogation*.

Figure 10 shows the meta-model that describes the computational viewpoint concepts. Again, the figure shows the basic modelling concepts in grey insofar as they are needed for the understanding of the computational viewpoint concepts.

As with the enterprise concepts, computational concepts can be interpreted as basic concepts by generalization: binding objects specialize objects, compound and primitive bindings specialize bindings, signal, stream and operation interfaces specialize interfaces and signals specialize interactions.

Flows and operations can not be interpreted as interactions, because their semantics differs from that of an (atomic) interaction. Moreover, RM-ODP states that flows and operations can be interpreted as a composition of signals [14 (part 3, clause 7.2.2.5)] and therefore of basic interactions. An operation can be interpreted as a composition of basic interactions, by considering both an invocation and a termination as composed of two interactions, one for each communicating partner. Since the termination of an operation is optional, an operation is composed of either two or four basic interactions. A flow can be interpreted as a sequence of basic interactions. Hence, to interpret flows and operations in terms of basic modelling concepts we need a more sophisticated relation than generalization. This also means that we need more sophisticated tooling mechanisms to map flows and operations to basic interactions. We intend to use model transformations for this purpose, but we leave this for future work.

Note that, when we transform a computational view into a basic view according to these interpretation rules, we lose all information about the constituents of the compound bindings. As a result, binding objects are interpreted as regular objects with a regular behaviour and primitive bindings with other objects. We do not consider this loss of information a problem for comparing a computational view to the enterprise viewpoint, because from an enterprise perspective it is only relevant to know the behaviour of a (binding) object. Whether this

behaviour relates to a connection between objects or to application logic is unimportant.

6. The relation between the enterprise and computational viewpoints

An enterprise view represents the relation between a system under design and its environment, while a computational view represents the functional decomposition of that system. Therefore, we claim that the computational viewpoint is related to the enterprise viewpoint by delimitation and refinement. A computational view is delimited with respect to an enterprise view, because an enterprise view describes the system *and* its environment, while the computational view only describes the system. Therefore, actions can exist in the enterprise view, which are not considered in the computational view. These are the actions that are performed by the environment of the system. Furthermore, a computational view is a refinement of an enterprise view, because an enterprise view describes the system as a whole, while the computational view describes the functional decomposition of the system. A computational view may also describe the actions that the system performs in more detail than the enterprise view does. These observations are supported by the consistency rules that RM-ODP prescribes between the enterprise and computational viewpoints [13].

If a combination of the role-based and the process-based approach is used in the enterprise viewpoint, it is likely that some enterprise actions are only considered in the process-based part while others are only considered in the role-based part.

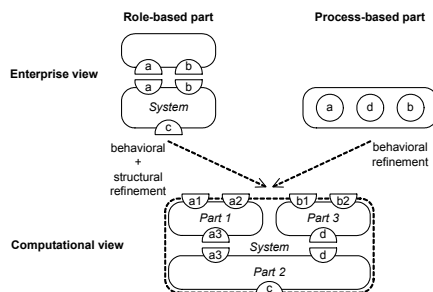


Figure 11. Example of relations between computational and enterprise models

Figure 11 illustrates the relation between the computational viewpoint and the enterprise viewpoint. It shows an enterprise view and a computational view. The computational view is a structural decomposition of the enterprise view, because the enterprise view represents the system as a whole, while the computational view represents the system as a composition of three parts. The

computational view also is a delimitation of the enterprise view, because the computational view does not show the enterprise object that is part of the environment of the system. Moreover, the process-based part of the enterprise view does not make a distinction between the actions that are performed by the system and the actions that are performed by the environment of the system. Finally, the computational view is a behavioural refinement of the enterprise view, because the computational view describes the actions that the system performs in more detail. Specifically, it decomposes action *a* into actions *a1*, *a2* and *a3* and action *b* into actions *b1* and *b2*. The figure shows action *d*, which is only considered in the process-based part of the enterprise view, and action *c*, which is only considered in the role-based part of the enterprise view.

Based on these observations, we define rules to verify the consistency between an enterprise and a computational view. These rules lead to a formula for verifying the consistency between a computational view and an enterprise view, which we define in terms of the operators from section 4. We verify the consistency between the computational view and the process-based part separately from the consistency between the computational view and the role-based part. We do this, because the process-based part considers actions that the role-based part does not consider and vice versa. To verify the consistency, the designer must first specify the relations that exist between elements from the computational view and elements from the enterprise view. These relations are used as input when verifying the consistency.

Since the behaviour of the system in the computational view is a decomposition of the behaviour of the system in the enterprise view, the designer must specify which behaviours in the computational view represent parts of the system. To verify consistency, these behaviours must be composed into a single system behaviour. Hence, for computational behaviours cb_1, cb_2, cb_3 that represent the behaviours of the system parts:

$$cb = compose(compose(cb_1, cb_2), cb_3)$$

When we verify the consistency between the computational view and the role-based part, the designer must specify which actions are not considered in the role-based part. To verify the consistency between the computational view and the role-based part, we must abstract from these actions. Hence, for a computational behaviour cb and a set of actions *unconsidered* that are not considered in the role based behaviour:

$$cb' = abstract(cb, unconsidered)$$

Similarly, when verifying the consistency with the process-based part, we must abstract from actions that are only considered in the role-based part.

Since the computational view only considers the system, the designer must specify which actions in the enterprise view are performed by the environment of the system. To verify the consistency, we must abstract from these actions in the enterprise behaviour. Hence, for an enterprise behaviour eb (that can be either a role-based or a process-based behaviour) and a set of actions $environmentactions$ from the environment of the system:

$$eb' = abstract(eb, environmentactions)$$

Since the computational view also considers the actions in the enterprise view on a more detailed level, the designer must specify the relation between actions in the computational and enterprise view. Some computational actions can be inserted with respect to enterprise actions. Since these actions represent design information that is added between the enterprise and the computational viewpoint, they must be removed in the abstraction step. We say that these actions are abstracted from. Other computational actions can be reference actions for enterprise actions. For reference actions, the designer has to specify the completion condition and the enterprise action to which they correspond. The reference actions have to be integrated. Hence, for a computational behaviour cb , a set of inserted actions $inserted$, and a set of completion conditions c_1, c_2, \dots on reference actions that specify the completion of the original actions a_1, a_2, \dots :

$$cb' = integrate(abstract(cb', inserted), \{(c_1, a_1), (c_2, a_2), \dots\})$$

Finally, we have to compare the resulting enterprise and computational behaviour, using the equivalence operator.

As an example consider figure 11, with a role-based behaviour rb , a process-based behaviour pb and a computational behaviour that consists of the behaviours cb_1, cb_2 and cb_3 of the system parts. Further, we say that the completion of a_1 and a_2 corresponds to the completion of a and the completion of either b_1 or b_2 corresponds to the completion of b . Hence, action a_3 is an inserted action. Now, to assess whether the computational behaviour is consistent with rb , we use the formulae:

$$cb''' \sim rb'$$

where:

$$rb' = abstract(rb, environmentactions)$$

$$cb''' = integrate(cb'', reference)$$

$$cb'' = abstract(cb', inserted)$$

$$cb' = abstract(cb, unconsidered)$$

$$cb = compose(compose(cb_1, cb_2), cb_3)$$

$$reference = \{(a_1 \wedge a_2, a), (b_1 \vee b_2, b), (c, c)\}$$

$$inserted = \{a_3\}$$

$$unconsidered = \{d\}$$

$$environmentactions = \emptyset$$

Figure 12 illustrates how these formulae affect the computational behaviour.

To assess whether cb is consistent with pb , we use the same formulae, but with rb replaced by pb and:

$$unconsidered = \{c\}$$

$$reference = \{(a_1 \wedge a_2, a), (b_1 \vee b_2, b), (d, d)\}.$$

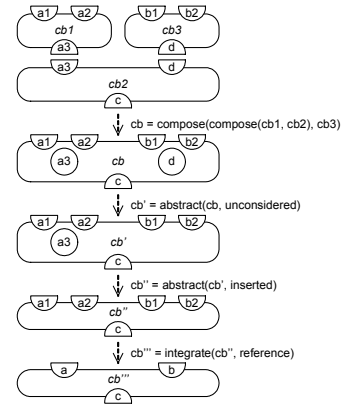


Figure 12. Relations between abstractions of computational behaviour

From the above, we can deduce that the designer must make considerable effort to specify the relation between an enterprise and a computational behaviour. Such effort may cause the designer to ignore the consistency check. However, the consistency check can be simplified. Observe that all actions, other than the ones that appear in the $reference$ set, are abstracted from. Hence, the designer only has to specify the $reference$ set. Then, a tool can automatically abstract from all other actions.

Process-based and role-based parts can partly deal with the same actions. In figure 11 this is the case for actions a and b . Because the role-based and the process-based parts of the enterprise viewpoint partly deal with the same actions, we also say that they are complementing viewpoint designs. Therefore, the consistency between the role-based and the process-based part should be verified to construct a consistent enterprise view. However, we do not do this here, because our goal is to verify the consistency between an enterprise and a computational view.

7. Example

As an example, we relate the enterprise view from figure 13 to the computational view from figure 14. We represented both views with UML as a concrete syntax. The benefit of using UML as a concrete syntax is that existing UML tools can be reused for enterprise and computational viewpoint design. The drawback of using UML is that it uses different concepts than the ones proposed for enterprise and computational design in section 5. Hence, before we can analyze or compare the

designs as enterprise or computational designs, we have to transform the UML models into designs that use the viewpoint concepts from section 5. To make this feasible, meta-model transformation techniques are proposed, specifically for transforming a concrete syntax into an abstract syntax [1, 2].

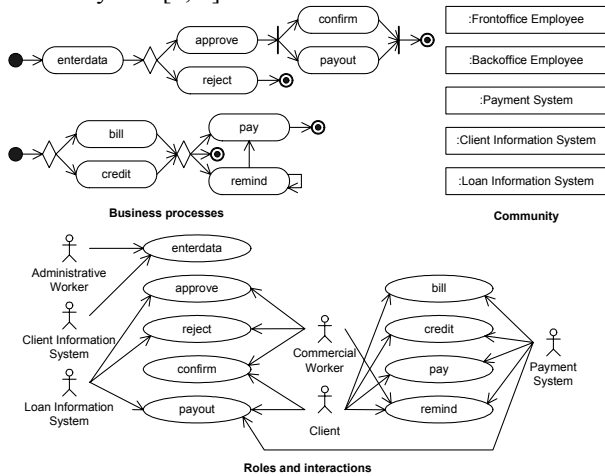


Figure 13. A representation of an enterprise view with UML

Figure 13 represents the enterprise view. An activity diagrams represents the business processes. A use case diagram represents the contribution of roles to steps from the business processes. An object diagram represents the objects that are part of the community. The enterprise view shows two business processes that describe the behaviour of the community to which the system under design belongs. The first business process describes how a loan application is processed. First the details of the application are entered into a computer system, then the loan is either approved or rejected and, if the loan is approved, a confirmation is sent and the loan is paid out. The second business process describes how monthly payments for the loan are cashed. Depending on the kind of contract of the client, the client's account is credited directly or a bill is sent to the client. If the client does not pay the bill or the client's account could not be credited, he receives a reminder. The roles that are involved in the business process are indicated as actors in a use case diagram. Each of the steps in the business processes is an abstraction of an interaction between several roles, as indicated by the use case diagram.

The system under design belongs to a community that consists of five objects: two employees and three software systems. These objects fulfil roles in the enterprise, although we did not model this with UML. The front office employee fulfils both the role of administrative worker and of commercial worker, the back office employee fulfils only the role of administrative worker

and the software systems fulfil the roles with the same names. The client role is not fulfilled by any object from the enterprise community.

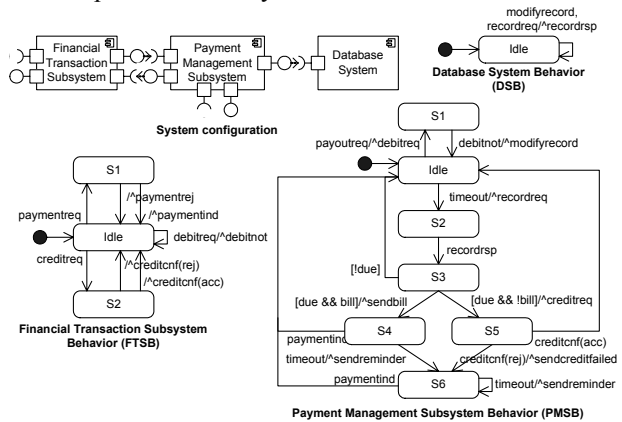


Figure 14. A representation of a computational view with UML

Figure 14 represents the computational view. A component diagram represents the objects and their connections. state machines represent the behaviour of the objects. The computational view shows the realization of the payment system that was identified in the enterprise view. The view shows that the system is composed of three interacting parts. The parts communicate via interaction points. In addition to this, the financial transaction subsystem and the payment management subsystem have interaction point parts that can form interaction points with objects outside of the payment system. The state machines show that, in the database, we can either modify a record or request a record by some SQL query. The financial transaction subsystem allows the bank to debit the accounts of its clients. Also, it allows an account to be credited, either by the bank or by the owner of the account. The bank can request an account to be credited with the *creditreq* interaction and the client can request this with the *paymentreq* interaction. Crediting an account fails if there is not enough money in the account. The payment management subsystem manages the payment of the loan to the client and the monthly payment of the fees associated with the loan. The financial transaction subsystem performs the payment of the loan on request (of the payment management subsystem). Upon payment, the payment management subsystem stores information about the payment in the database. Each month, upon a timeout, the payment subsystem checks in the database if the client has to pay. If a payment is due and the preferred method of payment is sending a bill, then the bank sends a bill to the client and waits for the client to pay. If the preferred method of payment is by directly crediting the client's account, then the financial transaction subsystem is told

to credit the client's account. If crediting the account fails or the client does not pay his bill within a certain period, the client is sent reminders until he pays.

To verify the consistency of the enterprise and computational view, we must transform them to the abstract syntax of their viewpoints as explained above. Subsequently, we must transform them to the abstract syntax of the basic viewpoint from section 3. This is trivial because the relations between concepts from the enterprise and computational viewpoints and the basic viewpoint are mostly generalization relations. Therefore, without presenting the exact transformations, we claim that the enterprise view corresponds to the basic viewpoints design from figure 15 and the computational view to the design from figure 16.

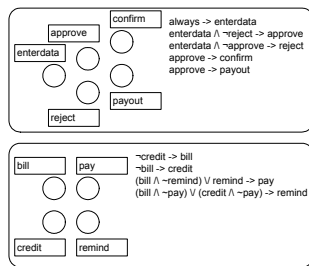


Figure 15. A basic enterprise view

Finally, we must compare the resulting basic viewpoint designs by comparing the computational behaviour to each of the business processes from the enterprise view. To compare the views, we must specify the precise correspondence between their actions. Also, we must specify which behaviours from the computational view are a decomposition of a behaviour from the enterprise view.

The correspondence between behaviours is such that all behaviours from the computational view are a decomposition of the behaviour of the payment system from the enterprise view. The enterprise system engages in the actions: *payout*, *bill*, *credit*, *pay* and *remind*. The correspondences between the actions from the computational behaviour and the first business process are as follows. The action of storing details about the loan payment in the database corresponds to the completion of the *payout* step in the business process. The *payoutreq*, *debitreq* and *debitnot* actions are inserted actions and the other actions and steps are not considered in the comparison. Figure 17 illustrates this relation graphically. It shows how the computational view could have been reached from the enterprise view by behaviour decomposition and action refinement. The correspondence between the actions from the computational behaviour and the second business process are as follows. The *sendbill* action corresponds to the *bill*

step and *paymentind* corresponds to *pay*. The occurrence of either *credit(acc)* or *credit(rej)* corresponds to the *credit* step, because the credit step is indifferent about whether crediting the account succeeded or not. That decision is only made after the attempt to credit the account was made. Similarly, the occurrence of either *sendreminder* or *sendcreditfailed* correspond to the *remind* step. The other actions and steps are either inserted or not considered in the other view.

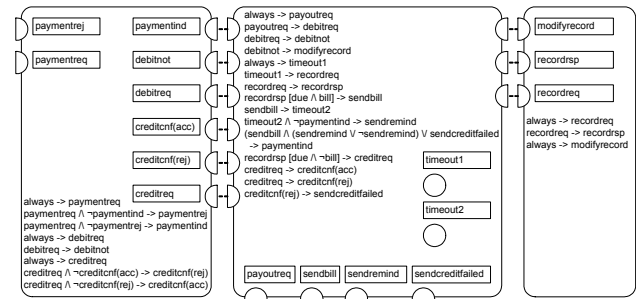


Figure 16. A basic computational view

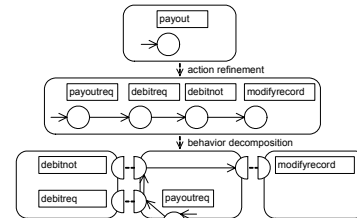


Figure 17. Refinement relations

We can assess straightforwardly that the computational behaviour is consistent with the first business process. To verify the consistency between the computational behaviour and the second business process, pb_2 , we use the following formulae:

$$cb'' \sim pb_2'$$

where:

$$pb_2' = \text{abstract}(pb, \text{environmentactions})$$

$$cb'' = \text{integrate}(cb'', \text{reference})$$

$$cb'' = \text{abstract}(cb', \text{inserted})$$

$$cb' = \text{abstract}(cb, \text{unconsidered})$$

$$cb = \text{compose}(\text{compose}(FTSB, PMSB), DSB)$$

$$\text{reference} = \{(\text{sendbill}, \text{bill}), (\text{paymentind}, \text{pay}), (\text{credit}(\text{acc}) \vee \text{credit}(\text{rej}), \text{credit}), (\text{sendreminder} \vee \text{sendcreditfailed}, \text{remind})\}$$

$$\text{inserted} = \{\text{timeout1}, \text{timeout2}, \text{recordreq}, \text{recordrsp}, \text{paymentreq}, \text{paymentrej}, \text{creditreq}\}$$

$$\text{environmentactions} = \emptyset$$

$$\text{unconsidered} = \{\text{payoutreq}, \text{debitreq}, \text{debitnot}, \text{modifyrecord}\}$$

The actions that can be abstracted from are all actions that do not appear as reference actions. This allows us to

simplify the formulae, because we do not have to define each set of actions that we have to abstract from explicitly. We can simply abstract from all actions in the design that are not reference actions. We invite the reader to verify the conformance. For this purpose a tool and a manual for using that tool is available on [22].

8. Related work

The work closest to ours is the Systemic Enterprise Architecture Methodology (SEAM) [23]. SEAM also proposes the use of basic modelling concepts as a basis for relating different viewpoints in enterprise application design. It makes these concepts more precise using both an ontological semantics [16] and a behavioural semantics [3]. However, while SEAM focuses on defining the basic concepts and their semantics precisely, our work focuses on defining the relations between viewpoints precisely.

Other work on relating RM-ODP viewpoints includes that described in [5]. This work differs from ours in that it relates viewpoints without using the RM-ODP basic modelling concepts. Using the basic concepts has the benefit that the semantics and operations defined on the basic concepts can be reused.

Other frameworks for viewpoint consistency verification are discussed in [7], [8] and [9]. These frameworks rely on the designer to specify (Boolean) constraints that define the relations between viewpoints. These constraints can be evaluated to verify the consistency between concrete views. These frameworks are more generic than our framework. However, our framework includes reusable operators, which make it more powerful for relating behavioural views.

9. Conclusions

In this paper we describe an approach to precisely define the relations between an RM-ODP based enterprise and computational viewpoints. We show how the approach can be used to verify the consistency between an enterprise and a computational viewpoint design and illustrate the approach with an example. So far the approach focuses on consistency between behavioural concerns of viewpoint designs.

Another goal of this paper has been to evaluate our framework for relating viewpoints [6] by applying it. We conclude that the framework can be successfully applied to relate the behavioural aspects of our enterprise and computational viewpoints. However, we expect that the framework is generic enough to be applied to other viewpoints as well. A point of attention, when applying the framework to relate other viewpoints, is that the relation between those viewpoints and the basic viewpoint may not be as straightforward as in RM-ODP.

This means that the designer may have to spend some time on defining this relation, which may annul the time saved with reusing the basic viewpoint relations.

The generic framework is also expected to be useful in the context of Model Driven Architecture (MDA) [17] and IEEE 1471 [12] compliant design trajectories. These trajectories acknowledge the existence of different viewpoints and the importance of specifying the relations between these viewpoints. In MDA the relations between viewpoints take the form of (automated) model transformations. However, we claim that it is not always feasible to relate viewpoints by means of transformations. Therefore, our approach to consistency verification between viewpoints can complement the MDA approach.

To illustrate the practical applicability of the framework we implemented it in a prototype tool [22]. The tool implements the design operators that we introduced in section 4. Section 6 explains how these design operators can be applied to verify the consistency between an enterprise and a computational view. The tool only uses RM-ODP basic modelling concepts.

Currently, we are adding support for meta-model transformation to the tool, such that it can interact with other tools, such as Poseidon or Rational Rose, which implement other (viewpoint specific) notations. In addition to this we plan to add support to the framework and the tool for verifying consistency with respect to other design concerns, more specifically: information and structure. We also plan to add support for verifying consistency between viewpoints that have a complement relation rather than a refinement relation.

To fully support consistency verification in RM-ODP based design approaches, we need to address other concerns than the behavioural concerns. Policies and dynamism of the structure of a system are examples of such concerns. Expressing policies is already possible to the extent of obligations, which represent that an object must perform a certain behaviour, and permissions, which represent that an object is allowed to perform a certain behaviour. These policies can be represented at a basic level by *must* and *may* conditions [19, 20], respectively.

Acknowledgement

This work is partly supported by the Dutch Freeband programme in the context of the A-MUSE project.

References

- [1] D. Akehurst, J. Derrick, and A. Waters. Addressing computational viewpoint design. In: *Proc. of the 7th IEEE Conf. on Enterprise Distributed Object Computing*, Brisbane, Australia, September 2003.

- [2] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In: *Proc. of the 5th Intl. Conf. on the Unified Modeling Language*, Dresden, Germany, 2002.
- [3] P. Balabko and A. Wegmann. From RM-ODP to the Formal Behavior Representation. In: *Practical Foundations of Business and System Specifications*, Kluwer Academic Publishers, September 2003.
- [4] G. Blair and J. Stefani. *Open Distributed Processing and Multimedia*, Addison-Wesley, 1997.
- [5] H. Bowman, E. Boiten, J. Derrick, and M. Steen. Viewpoint consistency in ODP, a general interpretation. In: *Proc. of Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall, 1996.
- [6] R. Dijkman, D. Quartel, L. Ferreira Pires, and M. van Sinderen. An approach to relate viewpoints and modeling languages. In: *Proc. of the 7th IEEE Conf. on Enterprise Distributed Object Computing*, Brisbane, Australia, September 2003.
- [7] A. Egyed. *Heterogeneous View Integration and its Automation*. PhD thesis, University of Southern California, Los Angeles CA, USA, 2000.
- [8] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseihbeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.
- [9] P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In: *Proc. of the 7th European engineering conference*, Springer Verlag, 1999.
- [10] R. van Glabbeek. The linear time – branching time spectrum I: the semantics of concrete sequential processes. In: *Handbook of Process Algebra*, pages 3– 99. Elsevier Science, 2001.
- [11] J. Gunawardena. Causal automata. *Theoretical Computer Science*, 101(2):265–288, 1992.
- [12] IEEE Architecture Working Group. *IEEE recommended practice for architectural description of software-intensive systems*, IEEE-Std 1471-2000, 2000.
- [13] ITU-T / ISO. *Information Technology - Open Distributed Processing Reference Model – Enterprise Language*, ITU-T Spec. ITU-T 911, and ISO/IEC Spec. ISO/IEC 16414, 1999.
- [14] ITU-T / ISO. *Open Distributed Processing Reference Model Part 1-4*, ITU-T Spec. ITU-T 901..4 and ISO/IEC Spec. ISO/IEC 10746-1..4, 1995.
- [15] J.-P. Katoen. Causal Behaviours and Nets. In: *Proc. of the 16th Intl. Conf. on Application and Theory of Petri Nets*, Torino, Italy, 1995.
- [16] A. Naumenko. *Triune Continuum Paradigm: a Paradigm for General System Modeling and its Application for UML and RM-ODP*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2002.
- [17] OMG. *Model Driven Architecture*, OMG Specification ormsc/02-07-01, 2001.
- [18] J. Putman. *Architecting with RM-ODP*, Prentice Hall, 2001.
- [19] D. Quartel. *Action Relations - Basic Design Concepts for Behaviour Modelling and Refinement*. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
- [20] 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.
- [21] D. Quartel, L. Ferreira Pires, M. van Sinderen, H. Franken, and C. Vissers. On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems*, 29(4):413–436, 1997.
- [22] D. Quartel. *ISDL home*: <http://isdl.ctit.utwente.nl/>, n.d.
- [23] A. Wegmann. On the systemic enterprise architecture methodology (SEAM). In: *Proc. of the Intl. Conf. on Enterprise Information Systems*, Berlin, Germany, 2003.