A Design-for-Change Approach: Developing Distributed Applications from Enterprise Models

Remco M. Dijkman, Dick A.C. Quartel,
Luís Ferreira Pires, Marten J. van Sinderen
{dijkman, quartel, pires, sinderen}@*cs.utwente.nl*

Centre for Telematics and Information Technology,
University of Twente
PO Box 217, 7500 AE,
Enschede, The Netherlands

Abstract

This paper presents a novel approach to distributed applications design. The proposed approach considers both the enterprise viewpoint and the computational viewpoint of distributed applications during the design process. Two important benefits are thus accomplished: (1) the resulting distributed applications will better match the enterprise's needs, and (2) changes in the enterprise can easily be translated to changes in the distributed application. The approach comes with a formal notation that makes it possible to define a precise relation between enterprise models and models of the distributed applications.

Keywords: distributed application, formal modeling, enterprise model, RM-ODP, evolvability, structuring technique, ISDL.

1 Introduction

The ability to timely adapt to new demands and conditions is one of an enterprise's main ways of getting a competitive advantage. Effective adaptation is often not limited to the enterprise's business processes, but extends to the distributed applications that support the business processes. We call the ability of distributed applications to change together with the enterprise: the evolvability of the applications [11].

When developing evolvable applications, we have to be aware that we are dealing with two knowledge domains: the domain of the business architect, who designs enterprises using enterprise concepts, and the domain of the application architect, who designs applications using application concepts. To be able to develop evolvable software, the concepts from the two domains have to be directly related [2, 8]. Only then can we implement changes in the enterprise design directly in the proper place in the applications. In the method that we propose here, we use a common design notation as the linking pin between the concepts from the two domains. This approach is shown in figure 1. The figure shows that the design notation can be used to express concepts from both domains. In order to relate an enterprise design to an application design precisely, we define a precise technique to relate models that represent designs from the two domains.

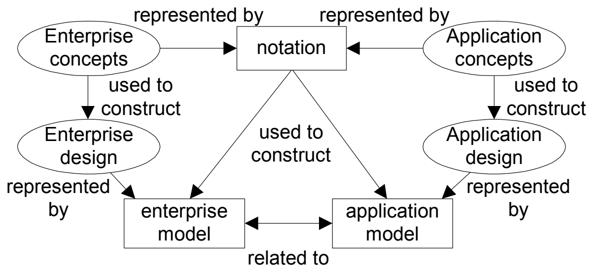

Figure 1 – Relation between Enterprise and Application Designs

We have used the Reference Model for Open Distributed Processing (RM-ODP) [9, 10, 11, 12] as a basis for the development of our approach. We did this because the RM-ODP is the result of a consensus between a large number of companies and academic institutions on the concepts that have to be used for the design of open distributed systems. Another reason for using the RM-ODP is that it provides concepts and abstractions that allow us to relate distributed applications to enterprise designs.

We developed our approach by investigating the relations between the concepts from the RM-ODP enterprise and computational viewpoint. These viewpoints define concepts for enterprise design and distributed application design, respectively. Based on the relation between the concepts, we defined a relation between the two viewpoints. We used a formal notation [18, 21] to define this relation precisely. Using the relations between the concepts we also identified structuring techniques, and we made sure that the structuring techniques in the enterprise viewpoint has counterparts in the computational viewpoint. When using the same structure in both viewpoints, changes to the enterprise design can easily be traced to changes in the computational design.

The remainder of this paper is organized as follows. Section 2 introduces the RM-ODP concepts that we use for enterprise and application design. Also, this section describes the relations between the concepts for enterprise design and the concepts for application design in an informal manner. Section 3 identifies the structuring techniques that are used in enterprise design, and explains how these techniques relate to the structure of distributed applications. Section 4 introduces the formal notation and explains how the notation can be used to define the relation between enterprise and application models. Finally, section 5 presents a case study to illustrate the use of our approach.

2 RM-ODP Concepts

The Reference Model for Open Distributed Processing provides a means for defining standards for open distributed processing, such as standards for components of ODP systems, and standards for modeling ODP systems.

The RM-ODP defines five viewpoints, each of which focuses on different aspects of a system design. Each viewpoint is associated with a viewpoint language that can be used for developing designs from this viewpoint. A viewpoint language consists of a number of concepts that can be used when defining standards from this viewpoint, and rules that define the relations between the concepts.

In this paper we focus on the enterprise and the computational viewpoint. The enterprise viewpoint language can be used to specify the enterprise in which the ODP system is embedded. The computational viewpoint language can be used to specify the logical decomposition of the system into interacting objects.

We use the ODP computational viewpoint to design distributed applications in a platform-independent way. The distributed applications can subsequently be mapped to a specific component technology such as Enterprise Java Beans. This approach is similar to the the Model Driven Architecture approach [15] and the approach used in the EDOC profile for UML [13, 14].

2.1 The ODP enterprise viewpoint

The enterprise viewpoint is defined in [11], and was extended by the concepts and rules from [12] later on. The enterprise language can be used to specify the system in its environment.

The key concept in the enterprise viewpoint is the *community*[1] concept. The definition of a community is: ‘A configuration of *objects* formed to meet an *objective*’ [11]. The objects mentioned in the definition (in the enterprise viewpoint also known as enterprise objects) represent real entities in a company, an example of which could be an employee, or the system that has to be built. Communities can be hierarchically structured. We can do this by viewing a community as a community enterprise object, and making this community enterprise object part of another community.

A community is specified in detail by specifying [11]:

- the *roles* fulfilled by the enterprise objects;
- the *steps* within the *processes* in which the enterprise objects participate;
- the *policies* that apply to the enterprise objects.

These concepts and the rules that apply to them are explained in the sequel.

A role is an identifier for behavior [10]. The behavior it identifies is specified in terms of *actions* and constraints on when these actions can occur. An action is something that happens. Each action in a community is either part of a single role, or part of multiple roles. In case it is part of multiple roles it defines an *interaction* between these roles. An enterprise object can participate in an action by fulfilling the role to which the action is assigned (or, in case of an interaction, one of the roles to which this interaction is assigned). When an enterprise object is said to fulfill a certain role, all actions in this role are associated with this enterprise object. An enterprise object can fulfill several roles in a community. However, at one moment in time, a role can only be fulfilled by one enterprise object. Therefore, if we want to define the behavior of a role once, and associate it to more than one enterprise object, we must define a role *type*, and define multiple *instances* of this role type, each of which can be assigned to a different enterprise object.

A process is also an identifier for behavior. It is defined as: ‘A collection of *steps* taking place in a pre-described manner and leading to an objective’ [12], where a step is defined as: ‘An abstraction of an action, … that may leave unspecified objects that participate in that action’ [12]. While both roles and processes are identifiers for behavior, there is a difference between these concepts: roles focus on the assignment of behavior to enterprise objects, while processes focus on the assignment of behavior to objectives [1]. Processes can be hierarchically structured. This can be done by decomposing a step of a process into a more detailed process [12]. A step must be assigned to an *actor* role. An actor role is a role that participates in performing an action. In contrast we can also have artefact roles that are only referenced in an action. By assigning a step to a role, or multiple roles in

[1] all concepts are in italics when they are first mentioned.

case the step is an abstraction of an interaction, we can eventually assign it to enterprise objects.

'A policy identifies the behavior, or constraints on a behavior, that can be changed during the lifetime of the ODP system …' [12]. Typically, a policy is defined in terms of a set of rules. Each rule can express either an *obligation*, a *permission*, a *prohibition*, or an *authorization*. An obligation is a prescription that a particular behavior is required to occur. A permission is a prescription that a particular behavior is allowed to occur. A prohibition is a prescription that a particular behavior must not occur, and an authorization is a prescription that a particular behavior must not be prevented.

It is hard to pinpoint the exact relation between a policy and the enterprise behavior that is specified by the roles and processes. One view on the relation between policies and enterprise behavior is that policies describe the desired behavior of an enterprise, while the enterprise behavior describes the actual behavior [24]. Another view on the relation between policies and enterprise behavior is that enterprise behavior describes the behavior an object is physically capable of, while policies prescribe a social behavior that objects must commit themselves to when they join a community [16]. Both views on the relation between policies and enterprise behavior describe enterprise behavior as the actual behavior of an enterprise, and policies as constraints on this behavior. In this paper we view policies in this way.

We will not discuss how policies can be expressed in our notation, nor how they can be related to distributed applications. We will merely interpret them as constraints on behavior. A detailed discussion on policies is outside the scope of this paper.

2.2 The ODP computational viewpoint

The computational viewpoint is defined in [11]. At the computational viewpoint we can specify the logical decomposition of a system into objects that interact through their interfaces. The logical decomposition splits up the system into distributed parts.

A computational design is given in terms of a configuration of (computational) objects. Objects can perform actions by themselves, or participate in interactions with other objects. Interactions between objects happen at *interfaces*. Computational objects have a quality of service contract with their environment.

An interface can either be a *signal interface*, *an operation interface*, or a *stream interface*, depending on the type of interactions that can happen at the interface. At a signal interface, *signals* happen, which are defined as atomic shared actions that result in one-way communication from an initiating object to a responding object. At an operation interface *operations* can happen, which can be either *announcements*, or *interrogations*. An announcement is the conveyance of information from a client to a server. An interrogation is a two way communication in which a client requests a function from a server, to which the server subsequently responds by sending the information resulting from the requested function call. At a stream interface, *flows* happen. A flow is an abstraction of a sequence of interactions, in which information flows from a producer object to a consumer object. An interface does not only specify the type of interactions that can happen, but also the allowed sequences in which the interactions can happen. Therefore, an interface completely defines a subset of the behavior of the object.

For two objects to be able to interact on their respective interfaces, a *binding* must be established between these interfaces. Bindings can only be established between two *compatible* interfaces. A precise definition of compatibility between interfaces is given in [11], but we trust on the readers intuitive understanding of compatibility. A binding may be represented by a *binding object*. This has the benefit that the binding can support interactions between more than two interfaces. Another benefit of a binding object is that it can be used to control the binding it represents, by providing *control interfaces*.

Apart from the run-time establishment of bindings between interfaces, objects can instantiate other objects at run-time from *object templates*. An object template corresponds to a class definition in object-oriented programming languages. An object template consists of a number of *interface templates*, which can be used to instantiate the object's interfaces.

2.3 The relation between the viewpoints

A correspondence statement tells how an element of an enterprise design relates to different elements of a computational design. [12] specifies a number of correspondence statements that are mandatory if you want to produce a coherent design, and not just two designs from two viewpoints that are seemingly unrelated. From these correspondence statements we can derive the relation between the enterprise and the computational viewpoint.

The key correspondence statement tells that it is mandatory to specify, for each enterprise object, the computational objects by which it is realized. This correspondence statement does not specify a true refinement relation [4], in the sense that some objects in an enterprise design may not appear in the computational design at all, because they are not automated but rather related to human behavior [17]. From this correspondence statement, we derive that each enterprise object can be realized by multiple computational objects.

From the other correspondence statements, we can derive that an interaction in the enterprise viewpoint is realized in the computational viewpoint by a number of interactions that happen on computational interfaces that have a binding between them. Furthermore, if an enterprise object is constrained by a policy, then the computational objects that realizes this enterprise object must address this policy.

3 Structuring Techniques

In this section we explain how distributed applications can be structured based on an enterprise design. Our goal is to define the structures in such a way that we are able to propagate changes in the enterprise design easily to changes in a computational design.

This section starts by explaining the structuring mechanisms that can be used in the enterprise viewpoint. It then explains how the system can be delimited in the enterprise viewpoint. Finally, it explains how the structure of a design from the computational viewpoint relates to the structure of a design from the enterprise viewpoint.

3.1 Structuring the enterprise viewpoint

We distinguish two approaches to structure enterprise behavior: the role-base and the process-based approach [1, 12]. The role-based approach abstracts from the concepts for process design, which are process and step, and structures the enterprise behavior into roles. The role-based approach focuses on who-does-what in the enterprise [1]. The process-based approach initially abstracts from the role concept, and structures the enterprise behavior into processes. The process-based approach focuses on the actions that must be taken in order to reach a certain goal in the enterprise [1]. While the process-based approach initially abstracts from the roles that are involved in a process, we assign a process' steps to roles before we can assign a process to enterprise objects. Therefore, the process-based approach can not completely abstract from roles.

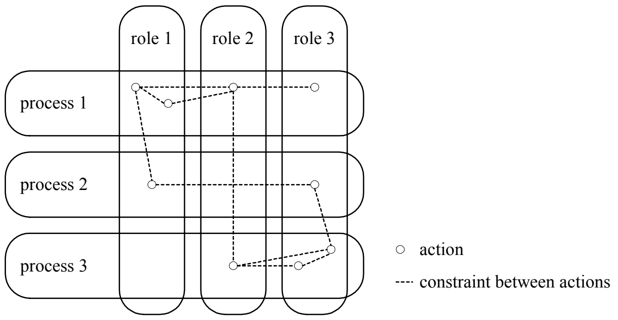

Figure 2 – Relation between Roles and Processes

Although the process-based and role-based approach have a different focus, they are both identifiers for behavior. Hence we consider the role-based and process-based approach as orthogonal, in the sense that each behavior that is defined as a process can also be defined as a collection of roles, and vice versa. The relation between the role-based, and the process-based approach is graphically represented in figure 2. This figure shows actions as dots, and constraints between actions as dotted lines. The actions that are performed in the enterprise, and the constraints on these actions are horizontally identified by processes, and vertically identified by roles. If we ignore the structuring of the actions and constraints into roles and processes, we get the monolithic enterprise behavior [28, 29].

The way in which constraints between actions in different processes and constraints between actions in different roles can be enforced, varies. Constraints between actions in different processes can be enforced directly. However, constraints between actions in different roles can only be enforced if the objects that perform these roles can communicate in some way.

We use interactions to express the communication between objects in different roles. In this case the interaction is the intermediary for a constraint that ranges over different roles. In practice an interaction may be implemented, for example, by sending a memo or a file over the internal mail. In contrast to using interactions to decompose a constraint that ranges over different roles, we use an exit-entry combination to represent a constraint that ranges over different processes.

In case of a constraint between actions that are both in different processes and in different roles, we should use an interaction.

The interactions between roles, and the constraints between processes are shown in figure 3.

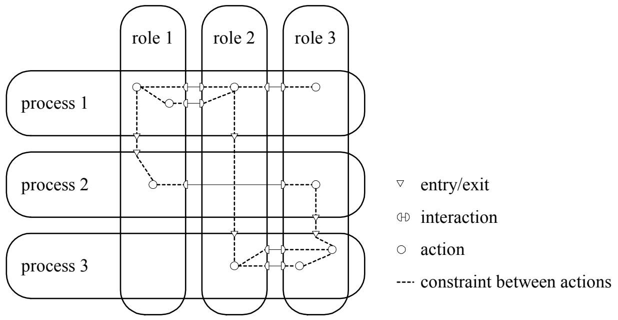

Figure 3 – Communication between Roles and Processes

3.2 Delimiting the system behavior

After defining an enterprise's behavior, we must decide which part of the behavior to automate. We call this the delimitation of system behavior. When delimiting the system behavior, we may decide that a role will partly be fulfilled by the system and partly by an employee. Since, at one point in time, a single role can only be assigned to a single enterprise object, the role then has to be split up.

When we split up a role, it is possible that one or more of its (inter)actions must also be split up, because this (inter)action will partly be performed by the system and partly by an employee.

When delimiting the system behavior, we must make sure that the enterprise behavior after the delimitation conforms to the enterprise behavior before the delimitation, otherwise the processes in the enterprise would be performed incorrectly after the implementation of the system. Therefore, only certain operations (or refinements) on (inter)actions are allowed to delimit the system. The allowed refinements are shown in figure 4 and 5.

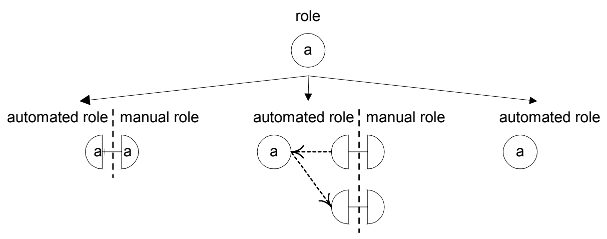

Figure 4 – Allowed Action Refinements

Figure 4 shows the acceptable refinements for splitting up an action. The leftmost refinement represents the case in which an action is supported by the system, in which case it is performed as an interaction between the user of the system and the system itself. An example of an action that is supported by the system is the entry of data into the system.

When an action is fully automated, additional interactions between the system and its user may be necessary. An interaction may, for example, be necessary to express the notification by the user that an action has to be performed. The middle refinement from figure 4 represents this situation. The rightmost refinement represents the situation in which no additional interactions with the user are necessary for the system to perform the action.

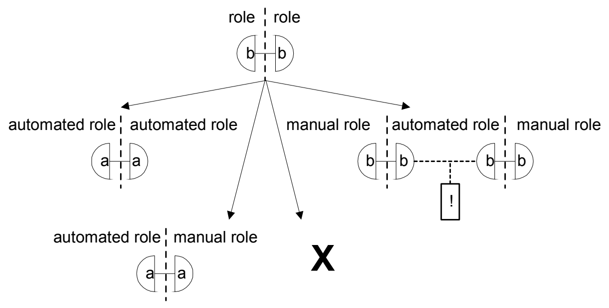

Figure 5 – Allowed Interaction Refinements

Figure 5 shows the acceptable refinements for splitting up an interaction. The top-left refinement represents the situation in which an interaction is replaced by an interaction between two roles of the system (e.g., different system parts). The bottom-left refinement represents the situation in which the interaction is replaced by an interaction between the system and a user. The bottom-right refinement represents the situation in which an interaction is fully implemented by one system part, and therefore removed. The top-right refinement represents the situation in which an interaction between two users is automated (e.g., by an e-mail system). In this situation, we must enforce the constraint that either all interactions by which an interaction is refined happen, or neither of them happens. In figure 5 this constraint is represented by an exclamation mark. The reason for this constraint is that an interaction is atomic, meaning that an interaction happens either at all roles, or not at all. This implies that atomicity must hold for a refinement of an interaction, otherwise a refinement would not be correct.

A constraint can only be automated, if it constrains actions that are in the same automated role, or interactions that have a contribution in the same automated role. Otherwise, the constraint must be enforced manually.

Finally, after the system behavior has been delimited, we assign the roles to enterprise objects. Enterprise objects may be employees, but also distributed applications, and most importantly, the system under development.

In principle the choice on how the system is going to support the enterprise is arbitrary, i.e., the choice on which actions and constraints to automate is arbitrary. However, usually the choice on how the system is going to support the enterprise follows a well-defined pattern. As an example we consider a special form of supporting the enterprise, which is called workflow automation. Other forms of supporting the enterprise are, for example, the support of the enterprise by an information system, a CSCW system, etc.

Workflow automation is defined as 'the automation of a business process, in whole or in part, during which documents, information, or tasks are passed from one participant to another for action, according to a set of procedural rules' [30]. In this type of process automation, not the actions that are performed in the enterprise are automated, but the constraints that impose the order in which the actions are performed. Also, the interactions between the roles that are involved in the process are automated, because they represent the transfer of information from one role to another. Consider, for example, the automation of process 1 from figure 3, which is represented by figure 6.

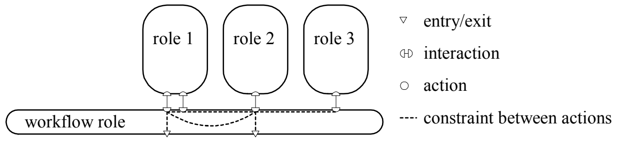

Figure 6 – Workflow Automation of a Process

In figure 6, we can see that the order in which the work is performed (represented by the constraints between the actions) is now the responsibility of the workflow role.

We can also see that the manual actions from figure 3, are replaced by interactions with the system. The part of the interactions from figure 6 that is assigned to roles 1, 2 and 3 represents the actual work that has to be performed in the enterprise (the original actions that were assigned to the roles in figure 3). The part of the interactions that is assigned to the workflow role represents the notification by the workflow system that the action can be performed, and the reception of the notification by the system that the user is done performing the action. Finally, we can see from the figure that the interactions between the roles in figure 3 are removed, because the communication between the roles is now enforced by the system.

3.3 Structuring the computational viewpoint

The computational viewpoint considers the decomposition of enterprise objects into computational objects. Only the enterprise object that represents the system under development is decomposed. The decomposition does not necessarily *have* to match the decomposition of enterprise objects into roles. However, we argue that it is *recommended* to make the first logical decomposition of the enterprise object into computational objects meet the decomposition of the enterprise object into roles (and, indirectly via roles, processes). Such a decomposition makes it easier to trace changes in the enterprise viewpoint to changes in the system.

Consider, for example, figure 6, and assume that the workflow role and role three are automated by the system. Hence, these roles are assigned to the enterprise object that represents the system. In this case we argue that it is preferable to decompose this enterprise object into two computational objects that correspond to the roles that are automated.

It is possible to make the first decomposition of the system already at the enterprise level, by defining the system as a community object, and identifying sub-systems by enterprise objects that are part of this community object [6, 25]. However, one should be careful when using this approach, because the aim of the enterprise viewpoint is to express the relation between the system and its environment, and not the logical decomposition of the system. We only allow a decomposition into sub-systems in the enterprise design when these sub-systems are associated with clear objectives in the enterprise [25].

4 Formal Support to Relate Models

To facilitate precise and unambiguous reasoning about the relationship between an enterprise and corresponding computational model, we use the formal modeling technique ISDL (Interaction System Design Language). For this purpose, we define a mapping from the concepts of the RM-ODP enterprise and computational viewpoints onto ISDL. The use of a single language for the two viewpoints makes it easier to compare models that represent designs from both viewpoints. In particular, it allows us to define rules for refinement and for assessing the conformance between models at different abstraction levels.

In this paper we discuss ISDL in an informal manner. We refer to [21] for the formal semantics of ISDL.

4.1 ISDL

ISDL is a generic modeling language that has been developed for the modeling of various types of distributed systems, such as business processes, telematics applications and communication networks [5, 19, 23]. It is based on minimizing the limitations of existing formal methods [27]. An ISDL model of some system consists of two models: an entity model and a behavior model.

The entity model represents which system parts are considered, and how they are interconnected. Two concepts are used in an entity model: entity and interaction point. An *entity* represents a system (part) that performs some function or behavior, like, for example, a software component or business department. An *interaction point* represents some mechanism through which an entity can interact with other entities, like, for example, an electronic mail client.

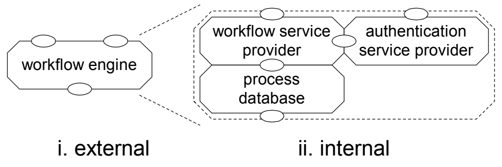

Figure 7 – An Example Entity Model

From an external perspective, a system is modeled by a single entity, having one or more interaction points. Figure 7.i, for example, models a workflow engine that interacts with its user via three interaction points. These interaction points could, for example, represent a workflow client, an interface for administering user information and business process modeling tool. An entity is graphically expressed by a rectangle with cut-off corners. An interaction point is graphically expressed by an ellipsis that overlaps with the entities that share (i.e., communicate via) the interaction point.

From an internal perspective, a system is modeled as a composition of functional parts. For example, these parts may represent sub-components or business units within some department. The internal perspective of the workflow engine is shown in figure 7.ii. This figure shows that the workflow engine is split up into three parts: a generic service provider that notifies users of the tasks that have to be performed, an authentication service provider, and a database for storing the business processes.

The behavior model represents the behavior, or functionality, of each entity in the corresponding entity model. Three concepts are used: action, interaction and causality relation. An *action* represents some unit of activity performed by a single entity. Consider, for example, the action of the workflow service provider that models the calculation of the next task to perform. An *interaction* represents a common activity performed by two (or more) entities. An interaction contribution represents the participation of an individual entity in the common activity. Consider, as an example, the interaction between the workflow service and its user that notifies the user of the next task to perform.

Information, *time* and *location* attributes can be added to an (inter)action, in order to model the result established in some activity, the time moment at which this result is available, and the location where the result is available, respectively. An (inter)action occurrence represents the successful completion of an activity. An (inter)action is atomic in the sense that if an (inter)action occurs, the same result is establish and made available at the same time moment and at the same location for all entities involved in the activity. Otherwise, no result is established and no entity can refer to any intermediate results of the activity.

An action is graphically expressed as a circle (or ellipsis). An interaction contribution is graphically expressed as a segmented circle (or ellipsis), which reflects that multiple entities contribute to the interaction. The information (ι), time (τ) and location (λ) attributes are represented within a text-box attached to the (inter)action[2]. Constraints can be defined on the possible outcomes of the values of ι, τ and λ (expressed after the symbol '|'). In case of an interaction, each interaction contribution defines the constraints of the corresponding entity, such that the values of ι, τ and λ must satisfy the constraints of all involved entities, otherwise the interaction can not happen. In case multiple values are possible for some attribute, a non-deterministic choice between these values is assumed.

A *causality relation* is associated with each (inter)action, modeling the conditions for this (inter)action to happen. Three basic conditions for the occurrence of some action *a* are identified:

- $b \rightarrow a$; action *b* must happen before action *a*;
- $\neg b \rightarrow a$; action *b* must not happen before, nor simultaneously with action *a*;
- $\surd \rightarrow$ a; action a is always enabled.

The and- ($\wedge$) and or-operator ($\vee$) can be used to model more complex causality conditions. For example, $b \vee \neg c \rightarrow a$ represents that action *a* can happen after action *b* has happened or as long as action *c* has not happened yet. Furthermore, a probability attribute can be added to each sufficient condition to model the probability the (inter)action happens when this condition is satisfied.

The causality relation concept allows the modeling of many different relationships between actions. It is often more convenient to express these relationships directly, instead of as a composition of causality relations of the individual actions. Figure 8. depicts the graphical expressions of some common relationships between two or three actions. The $\wedge$- and $\vee$-operator are graphically expressed by the symbols ■ and □, respectively. The $\surd$ condition is expressed by an arrow with no action attached to its shaft.

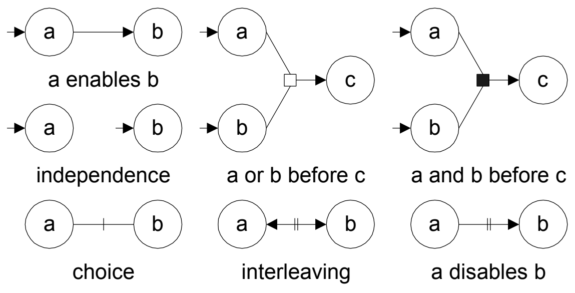

Figure 8 – Common Action Relations

ISDL supports two orthogonal techniques to structure a behavior in terms of a composition of smaller and simpler sub-behaviors: causality-oriented and constraint-oriented structuring.

Causality-oriented structuring is based on the decomposition of a causality relation by means of a syntactical construct, which allows one to define an action and its causality condition in distinct sub-behaviors. This syntactical construct makes use of:

- *entry points*, which are points in a behavior from which actions of that behavior can be enabled by conditions involving actions of other behaviors;
- *exit points*, which represent causality conditions in a behavior that can be used to enable actions of other behaviors.

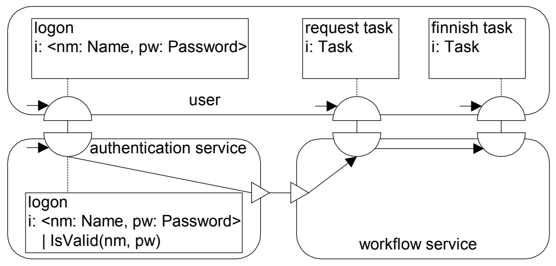

Figure 9 – An Example of Behavior Structuring

Constraint-oriented structuring is based on the decomposition of an action into an interaction, which allows one to define a behavior as a composition of

[2] In this paper we assume that (abstract) data types are used to represent attribute values, types and constraints, but we refrain from defining this representation.

interacting sub-behaviors. This technique can be used to decompose complex conditions and constraints on the execution of an action into simpler sub-conditions and sub-constraints that are assigned to interaction contributions defined in separate sub-behaviors. Furthermore, constraint-oriented structuring is needed to structure a behavior in sub-behaviors that can be assigned to different entities, since entities can only communicate via interactions.

Figure 9 shows an example in which both behavior structuring techniques are used.

A tool environment [26] exists for a dialect of ISDL, called AMBER [5]. The Friends project [7] has adapted and extended this tool environment to support the modeling of software components and their composition into telematics applications.

4.2 Modeling the Enterprise viewpoint in ISDL

In order to represent enterprise models consistently using ISDL, we define how the enterprise concepts explained in section 2.1 are represented in terms of ISDL concepts.

Communities and enterprise objects are represented as entities. Typically, a community is modeled as a composition of entities, since it is defined as a configuration of objects and can be hierarchically structured. However, from an external perspective (i.e., when abstracting from the internal functioning), a community can be represented as a single entity.

A role is represented by a behavior definition. Actions and interactions are represented directly in ISDL. The causality relation concept is used to represent constraints on the occurrence of (inter)actions. In order to represent that an enterprise object fulfills a certain role, the corresponding behavior definition is assigned to the entity that models the enterprise object in ISDL.

A process is also represented by a behavior definition. A process step can be represented as a single action, or alternatively as a behavior consisting of multiple actions in case one wants to decompose a step into a process. The relationships between roles and processes has been explained in section 3.1. These relationships can be represented directly using the constraint-oriented and causality-oriented structuring techniques.

We interpret a policy as a constraint on existing behavior (see section 4.2). A constraint can be modeled as a constraint on the attributes of an (inter)action, or as an additional action relation. We do not discuss the incorporation of policies into behavior models here. This topic will be addressed in a forthcoming paper.

4.3 Modeling the Computational viewpoint in ISDL

Analogous to the previous section, we define how the Computational concepts explained in section 2.2 are represented in terms of ISDL concepts.

A computational object is represented by an entity having one or more interaction points. An interaction point that is shared between two entities, represents a binding between these entities. A binding object can itself be represented as an entity. An interface is represented by a behavior definition, which defines possible interactions between some computational object and its environment. One may define a distinct interaction point for each interface in the entity, or alternatively assign multiple interfaces to the same interaction point.

Signals, operations and streams can be represented in ISDL as interactions and their causality relations. Figure 10, for example, illustrates the modeling of an interrogation, followed by an announcement.

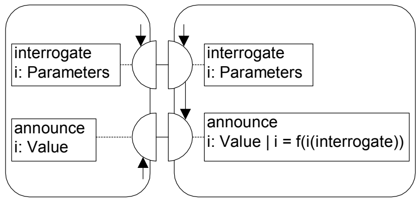

Figure 10 – Interrogation and Announcement

4.4 Behavior refinement

The enterprise model is more abstract in the sense that it prescribes *what* should be implemented by a supporting system, whereas the computational model is more concrete in the sense that it prescribes *how* the system should be implemented.

A technique called behavior refinement [18, 20, 21] has been developed for ISDL to enforce the correct replacement of an abstract behavior by a more concrete behavior. Since this technique applies to arbitrary ISDL behaviors, it can also be applied for the refinement of enterprise models into computational models. We focus on behavior, since it comprises most of the complexity of a design.

In general, an abstract behavior can be replaced by many alternative concrete behaviors. Depending on the choice of a concrete behavior, different concrete actions and their causality relations are added to the abstract behavior. Since this choice is determined by specific design objectives, behavior refinement can not be automated in its totality.

When abstracting from certain concrete actions and their causality relations, the abstraction of this concrete behavior is completely determined by the remaining concrete actions and their causality relations. In these circumstances, the abstraction of a concrete behavior is unique. Rules have been defined to calculate this abstraction [18, 21]. These rules can, in principle, be automated.

The uniqueness of an abstraction allows one to assess the conformance between an abstract behavior and a concrete behavior, by comparing the abstraction of the concrete behavior with the original abstract behavior. Therefore, the following design activities are distinguished in an instance of behavior refinement (see figure 11):

1. *delimitation of the abstract behavior*: we only consider the refinement of behaviors that are influenced by a finite number of abstract actions. For example, in case of recursive behaviors one should identify the finite behavior parts that are (infinitely) repeated;
2. *refinement of the abstract behavior into a concrete behavior*: in this activity we determine how the abstract behavior is implemented by the concrete behavior;
3. *determination of the abstraction of the concrete behavior*: a method to perform this activity is presented in [18, 21];
4. *comparison of the abstraction of the concrete behavior with the original abstract behavior*: both behaviors should comply to a certain correctness relation, e.g., an equivalence relation. If this is not the case, the concrete behavior is not considered a correct implementation of the abstract behavior. In this case the designer must return to design activity 2.

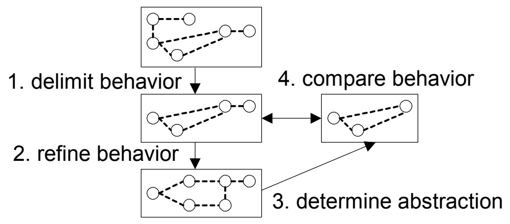

Figure 11 – Refinement Steps

Although one cannot automate the refinement of a behavior in general, one can define specific and frequently used refinements. This facilitates and shortens the design process, since in principle the conformance of such refinements has to be checked only once.

An example of a frequently used refinement is the delimitation of the system behavior to a workflow management system, discussed in section 3.2.

5 Case Study

In this section we illustrate our approach by means of a simplified case study.

After introducing the case study informally, we show how it can be modeled from the enterprise and the computational viewpoint. We also show how the models from the different viewpoints relate for this particular case study, by considering both their structure and their formal relation.

5.1 Informal description

A leading Dutch bank uses the AMBER dialect [5] of the modeling technique we introduced in section 4 to model its business processes. At this bank, we investigated the business processes of the mortgage department.

One of the policies of the bank is that it obliges itself to send an answer to a mortgage application of a client within two weeks. This answer is either an acceptance or a rejection. Furthermore, the mortgage sales process ensures that before the bank accepts or rejects the mortgage application, it asks for additional papers to assure the financial status of the client. Examples of these papers are a pay slip or a credit status of the client's bank account. After the bank has accepted the mortgage application, it pays out the mortgage, and starts collecting the monthly payment of the mortgage and interest.

The actions that are performed by the mortgage department can be split up into two processes: the sales processes that has selling mortgages as its goal, and the payment process that has collecting the client's payment for the mortgage as its goal.

Internally, the mortgage department can be split up into two units, a sales unit, and an administrative unit. The sales unit sells mortgages, while the administrative unit performs the administrative actions on the mortgages after it has been sold, such as the monthly collection of the client's payments. The sales unit is split up into a front-office that has contact with the clients, and a back-office that performs the internal actions at the bank.

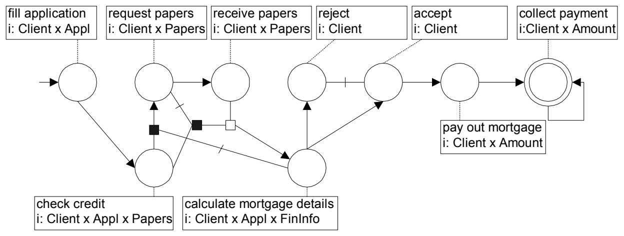

Figure 12 – Monolithic Enterprise Behavior

5.2 A model from different viewpoints

We first consider the monolithic enterprise behavior of the mortgage department. Figure 12 shows this behavior. The monolithic enterprise behavior only models the actions that take place in the department, without considering the decomposition of the actions into roles or processes. The actions with a double outline, like 'collect payment', can be performed more than once. We only consider one mortgage, therefore the entire behavior is carried out only once. If we want to consider more than one mortgage, the entire behavior must be instantiated more often.

Figure 12 models that after a mortgage application has been received, the creditworthiness of the client is checked. After this, either the mortgage details are calculated directly, or additional papers are requested and received from the client first. After the mortgage details are calculated, the application is either rejected or accepted. Once the mortgage is accepted it is paid out. After it has been paid out, the monthly action of collecting the mortgage payment is enabled. Figure 12 shows the information that is used in each action. The information attributes refer to elements of the static schema that can be defined in the ODP information viewpoint, which we do not take into account in this paper. Optionally, the relation between the information in each action may be shown. We can, for example, express that the amount that is paid out in the action 'pay out mortgage' is the same as the amount that is calculated and written down on the application form in the action 'calculate mortgage details'. The policy of the company to reply to an application within two weeks can be expressed as a constraint on the time attributes of the 'fill application', and the 'accept' and 'reject' actions.

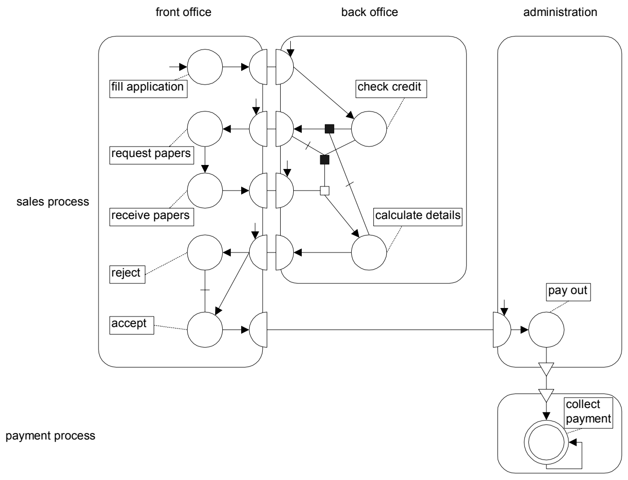

Figure 13 – Decomposed Enterprise Behavior

Figure 13 refines figure 12 by introducing roles, and replacing a relation that reaches over multiple roles by two relations that are connected by an interaction that represents the interchange of information between these roles. Consider, for example, the enabling relation between the action 'fill application' and the action 'check credit' from figure 12. In figure 13, this relation is replaced by an enabling relation from action 'fill application' to an interaction, and an enabling relation between this interaction and the 'check credit' action. The interaction that is introduced represents the transfer of the application form and the client information from the front office to the back office.

Figure 14 shows the enterprise model that results from identifying the system boundaries. To come to this model, we decided to automate both the sales, and the payment process with a workflow management system. Also we decided to automate the payment actions that are performed by the administration. The dotted line delimits the roles that are fulfilled by the enterprise object that represents the system.

Figure 14 shows that all actions that are performed by the roles that were assigned to 'employee objects' have become actions that are supported by the system. Also, the interactions that existed between the 'employee roles' from figure 13 have been removed, because the system now takes care of the transfer of information. The order in which the actions can occur is mostly controlled by the process roles. Only some minor ordering control is enforced by the front office role. The front office can, for example, make the decision whether a mortgage is accepted or rejected.

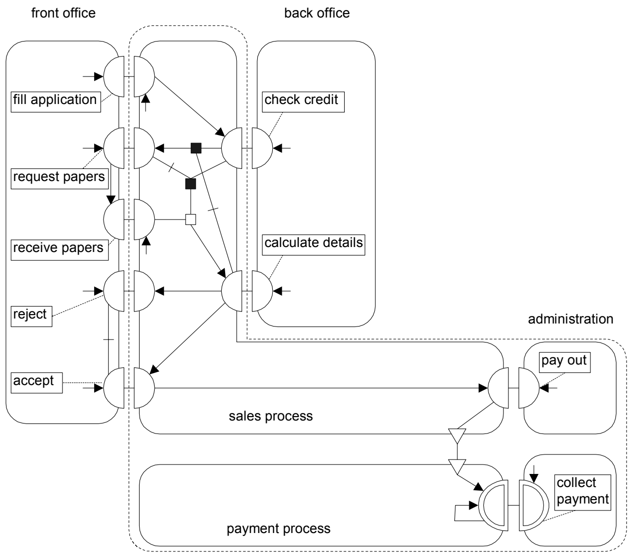

Figure 14 – Enterprise Behavior with System Boundaries

The decomposition shown in figure 14, can also be used as a first level logical decomposition of the system in the computational viewpoint. Subsequently we can decompose the system further. A possible logical decomposition of the system is shown in figure 15. For this decomposition we decided to build the system using a workflow service, and an administration object that can perform the payment actions. The workflow service makes use of an authentication service, and of a service that stores the business processes that the workflow system supports.

The behavior of the sales process from figure 14 is implemented in figure 15 by the joint behavior of the generic workflow service provider object, the authentication service provider object, and the sales process object. Whether or not the enterprise behavior is correctly implemented by these objects still has to be proven using the technique from section 4.4. However, to be able to construct this proof, we first have to define the generic behavior of the workflow service object, the authentication service object, and the container that contains the sales process object. The definition of these behaviors is not given in this paper because of space limitations.

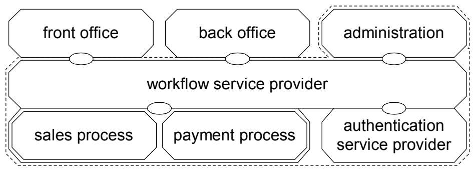

Figure 15 – Logical Decomposition at the Computational Level

The behavior that is specific to the enterprise's sales process, is implemented by the sales process computational object. Therefore, changes to the sales process in the enterprise, can easily be traced to changes in the sales process object in the computational model.

The behavior of both administration roles from figure 14 is implemented in figure 15 by the administration object. Therefore, if we change the behavior of either one of the enterprise roles, this traces to changes in the administration object. We may consider the decomposition of the administration object into a generic payment object, and two objects that reflect the specific behavior of the two enterprise roles. This makes a change in either one of the roles even easier to trace to computational objects.

6 Conclusions and Future Work

This paper proposes an approach for developing distributed applications based on an enterprise design. We define the approach according to the rules of the RM-ODP. Our approach is based on a modeling technique that allows us to precisely define the relation between the enterprise and the system that supports it. Further, the approach proposes structuring techniques for enterprise designs, and system designs such that the system design reflects the structure of the enterprise design. Thereby the relation between a part of the enterprise and a part of the system is clearly defined.

The benefit of our approach is that the relation between the enterprise and the distributed applications that support it, is defined precisely and intuitively clear, because of the formal modeling technique and the structuring techniques. This promotes the evolvability of the system, because changes in the enterprise can easily be propagated to the distributed applications. We have shown this briefly in a case study.

Our approach mainly focuses on the behavioral aspect of system design. We can also specify the relation to the information that is used in the enterprise, but complementary notations have to be used to define a complete model of this information.

The work in this paper is strongly related to the work on the relation between viewpoints from [3, 4, 25].

Much work still has to be done on the approach presented here. This work will be performed along three tracks. First, notational extensions will be defined to model actions for the creation and deletion of objects and bindings between objects. Second, notational extensions will be defined to model specific classes of behavior models, such as workflow descriptions. These extensions are expected to be comparable to UML profiles. Third, as explained in section 4.4, frequently used refinements will be defined. An example of such a frequently used refinement is the refinement from section 3.2 to transform a business process model into a business process model that is automated with a workflow engine.

References

[1] J. Aagedal, Z. Milošević. ODP Enterprise Language: UML Perspective, in C. Atkinson (ed.): *proceedings of EDOC 1999*, IEEE Press, pp. 60-71, 1999.

[2] L. Andrade, J. Fiadeiro. Coordination Technologies for Managing Information System Evolution, in K. Dittrich, A. Geppert, M. Norrie (eds.): *Proc. CaiSE 2001*, LNCS 2068, Springer Verlag, Berlin, pp. 374-387, 2001.

[3] H. Bowman, E. Boiten, J. Derrick, M. Steen. Viewpoint Consistency in ODP, a General Interpretation, in E. Najm, J.-B. Stefani (eds.): P*roceedings of Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall, pp. 189-204, 1996.

[4] H. Bowman, J. Derrick, M. Steen. Some Results on Cross Viewpoint Consistency Checking, in K. Raymond, L. Armstrong (eds.): proceedings of IFIP TC6 International Conference on Open Distributed Processing, Chapman and Hall, pp. 399-412, 1995.

[5] H. Eertink, W. Janssen, P. Oude Luttighuis, W. Teeuw, C. Vissers. A business process design language, in: *Proceedings of the World Congress on Formal Methods*, 1999.

[6] European Organisation for the Safety of Air Navigation, *ECHO Final Report*, 1.0 edition, 1997.

[7] Friends Project. http://www.telin.nl/Middleware/FRIENDS /ENindex.htm.

[8] P. Herzum, O. Sims. *Business Component Factory – A Comprehensive Overview of Component-Based Development for the Enterprise*, Wiley, New-York, 2000.

[9] ITU-T / ISO. *Open Distributed Processing Reference Model. Part 1 – Overview*, ITU-T X.901 | ISO/IEC 10746-1, 1995.

[10] ITU-T / ISO. *Open Distributed Processing Reference Model. Part 2 – Foundations*, ITU-T X.902 | ISO/IEC 10746-2, 1994.

[11] ITU-T / ISO. *Open Distributed Processing Reference Model. Part 3 – Architecture*, ITU-T X.903 | ISO/IEC 10746-3, 1995.

[12] ITU-T / ISO. *Information Technology - Open Distributed Processing Reference Model – Enterprise Language*, ITU-T X.911 | ISO/IEC 15414.

[13] OMG. *UML Profile for Enterprise Distributed Object Computing Specification*, ptc/02-02-05, February 2002.

[14] OMG. *A UML Profile for Enterprise Distributed Object Computing – Part II Supporting Annexes*, ad/01-08-20, August 2001.

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

[16] P. Linington, Z. Milošević, K. Raymond. Policies in Communities: Extending the Enterprise Viewpoint, in *proceedings of EDOC 1998*, pp. 11-21, 1998.

[17] J. Putman. *Architecting with RM-ODP*, Prentice Hall, Upper Saddle River, 2001.

[18] D. Quartel, L. Ferreira Pires, M. van Sinderen. On Architectural Support for Behavior Refinement in Distributed Systems Design, accepted in: *Journal of Integrated Design and Process Science*.

[19] D. Quartel, M. van Sinderen, L. Ferreira Pires. A model-based approach to service creation, in: *Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE Press, pp. 102-110, 1999.

[20] D. Quartel, L. Ferreira Pires, H. Franken, C. Vissers. An engineering approach towards action refinement, in: *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE Press, pp. 266-273, 1995.

[21] D. Quartel. *Action Relations- Basic Design Concepts for Behaviour Modelling and Refinement*, Ph.D. Thesis, University of Twente, Enschede, 1994.

[22] D. Rowe, J. Leaney, D.Lowe. Defining systems evolvability – a taxonomy of change, in: *Proceeding of the IEEE Conference on Computer Based Systems*, IEEE Press, pp. 45-52, 1998.

[23] M. van Sinderen, L. Ferreira Pires, C. Vissers, J.-P. Katoen. A design model for open distributed processing systems. *Computer Networks and ISDN Systems*, 27, pp. 1263-1285, 1995.

[24] M. Steen, J. Derrick. ODP Enterprise Viewpoint Specification, *Computer Standards and Interfaces*, vol. 22, pp. 165-189, 2000.

[25] C. Taylor, E. Boiten, J. Derrick. Interpreting ODP Viewpoint Specification: Observations from a Case Study, in B. Jakobs, A. Rensink (eds.): *Proceeding of Formal Methods for Open Object-Based Distributed Systems*, Kluwer, pp. 61-76, 2002.

[26] Testbed Project. http://www.telin.nl/testbed.

[27] C. Vissers, M. van Sinderen, L. Ferreira Pires. What Makes Industries Believe in Formal Methods, in A. Danthine, G. Leduc, P. Wolper (eds.): *Proceedings of the 13th International Symposium on Protocol Specification, Testing and Verification*, Elsevier, pp. 3-26, 1993.

[28] C. Vissers, G. Scollo, M. van Sinderen, E. Brinksma. Specification Styles in Distributed Systems Design and Verification, *Theoretical Computer Science*, vol. 89, Elsevier, pp. 179-206, 1991.

[29] M. de Weger. *Structuring of Business Processes*, Ph.D. Thesis, University of Twente, Enschede, 1998.

[30] WfMC. *Terminology & Glossary*, WFMC-TC-1011, 1999.