Automated management of inter-organisational applications

Lea Kutvonen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 UNIVERSITY OF HELSINKI
Lea.Kutvonen@cs.Helsinki.FI

Abstract

Inter-organisational applications require improved support from middleware services. This paper analyses the management requirements of multidomain applications, covering both technological domains and organisations (administrative business domains). The analysis leads to a suggested middleware solution for these needs, presenting a group of cooperative, pervasive management services that use a common, abstract management language suitable for expressing platform-independent model of cooperation between sovereign components and contracts about selected technologies. The approach is compatible with OMG MDA (Model Driven Architecture) work.

1 Introduction

Global systems and inter-organisational cooperation are essential requirements for current IT systems. Current middleware solutions already provide reasonable support for managing the technological heterogeneity of operating systems and network solutions and for adapting to dynamic changes in available resources. However, management aspects related to inter-organisational cooperation schemes still require improved support.

Because the operating environment is a distributed and heterogeneous environment covering multiple technology domains and multiple administrative domains, the middleware should address two levels of management services. Locally, for administering each technology domain, there is a need for component and service management, addressing issues such as lifecycle services and surveillance. In addition to this, new cooperative management services are needed for inter-organisational applications. The cooperative management of applications should include common application management facilities such as lifecycle, component location and (re)configuration. An essential issue to be solved is the control and maintenance of interoperability.

In a multi-organisational environment, the challenges stem from the sovereignty of systems. In a sovereign system, decisions – for example on application-level operational policies, platform architecture, object models, authorization policies and communication protocols – can be done independently from other systems. Further difficulties encountered by inter-organisational applications include the need to adapt to the constant change in potential partners and the independently driven development of services in each of these systems.

These requirements lead us to a solution, where application federations are established by demand and are dynamically configurable. We consider a federated system – that provides a service or represents an application – to be composed of sovereign services [16, 17]. A service is sovereign when it is provided by an autonomous organisation and supported by an autonomously administered IT system. Therefore, this approach does not build on administrative, platform-level decisions or heavy bridges that acquire an integrated, homogeneous computing environment. Focusing on services alone makes the approach in many respects similar to component-based approaches, although there is no strong requirement for component support from the platform on which service providers are running.

The inter-organisational application management is based on explicitly stated, platform-independent contracts – federation contracts – that define the forms of cooperation between performing components. The application structure is defined as a business architecture model, in terms of roles, interactions between the roles and policies governing alternative interaction sequences. Roles are populated even as late as at application runtime.

The preparation of a federation contract focuses on capturing a shared understanding of business logic and semantics of the services exchanged. The selection of members of a federation is further restricted by the technical requirements of each member, in respect to their needs for mutual communication.

The cooperative management approach suggested in this paper provides facilities for coherent application composition methodology ranging from design time to runtime.

For inter-organisational applications, the automatic discovery of new service components and new organisations is essential. In addition to discovery, application programmers need automatic tools for ensuring that interoperability facilities exist and business processes are respected, and for controlling the deployment, activation and accessibility of the supported services, without authority for direct management actions at distinct domains.

The approach has aspects of software engineering methodologies as well as runtime environment support in form of middleware services. This paper does not discuss the software engineering aspects at any deeper level, but one structural expectation deserves to be mentioned: We expect that component-based software engineering provides explicit and separate connector (binding) elements in addition to application components. This allows us to separate the issues of service types describing aspects of business logic and the issues of binding types describing structures within communication subsystems.

The cooperative management services are expected to be pervasive and based on a common, abstract management language. A pervasive service is available at each platform, regardless of the platform technology. Examples of current pervasive services include process management, naming services, and directory services. New management services include application composition, cross-domain configuration of operational policies, management of complex and dynamic bindings, and technology-independent interfaces for component life-cycle services. Because contracts are expressed across heterogeneous technology domains, the management language must be technology independent. At each domain an agent maps the contract expressions to technology-dependent solutions for that domain.

This paper focuses on cooperative, pervasive management services in response to OMG CORBA Management RFI [22]. However, the model is directly applicable to any other platform as well; there are no CORBA-specific requirements. The model presented is in line with the OMG model driven architecture (MDA) [21, 25, 27, 23] scheme, and RM-ODP (Reference Model of Open Distributed Processing) [10, 11] view of middleware services. This paper presents ideas under study and prototyping in Pilarcos project at the University of Helsinki [17].

The MDA is a vision statement on how OMG standards evolve in a coordinated fashion for improved interoperability and integration. The MDA discusses the needs of software engineering processes; interoperability and integration of applications and services so that portability of design and implementation are supported, and interoperability of implementations based on same design on different platforms.

The RM-ODP standard by ISO and ITU defines – in addition to terminology and viewpoints – a middleware model ([11], clauses on engineering viewpoint, functions and transparencies) that can be used to support inter-organisational applications. Key service elements there include trading service [20] for service discovery, type repository service [9] for storing type definitions and deciding about their mutual conformance, and binding framework [12]. Common language concepts of domains, communities, and federations support organisation of the interoperability solutions. Federation contracts based on the explicit business architecture descriptions give an interesting application to the ODP enterprise language [13].

The Pilarcos project (Production and Integration of Large Component Systems) develops middleware solutions for automatic management of inter-organisational applications. The federated architecture requires advanced middleware services for contract negotiation, for distributed startup of services, and for establishment of heterogeneous communication channels. The Pilarcos project prototypes these services, mostly in a CORBA environment.

The rest of this paper is structured as follows. Section 2 addresses the requirements for cooperative management services and the expectations set by them to the local management services. Section 3 outlines the cooperative management services, and the local services expected, together with the meta-information model elements related to them. Section 4 gives an overview of the suggested management approach with examples drawn from the Pilarcos project. Section 5 concludes the presentation by some related work and standardization issues.

2 New management requirements

The requirements for inter-organisational management services are discussed starting from the challenges of cooperative management. Four areas are identified to form separate management areas: control of the "global" application structure, control of the member components, testing the interoperability of the members, and running the interoperability facilities.

2.1 Expectations and challenges

We expect inter-organisational applications to have similar level of management support from platform services as applications restricted to a single domain. Application management is generally considered to include

- software engineering support for application production, such as reuse mechanism and composition of applications from components;
- application and component version management; and
- application installation, instantiation (startup), configuration and termination.

For inter-organisational applications, the task is complicated by two types of domains involved. First, there is technological heterogeneity, while applications cross platform domains. Separate platform domains can present different component models, language environments, transparency and quality of service levels of platform services, and communication protocols. Second, there is business-related heterogeneity, while applications cross administrative domains. Separate administrative domains present independent development and deployment of services, sovereignty of business policies (operational decisions) that affect cooperation patterns, and authorization policies.

Furthermore, the task of inter-organisational management is complicated by the asynchrony of development steps in each organisation. The composition and communication mechanism must therefore be flexible enough to allow late discovery and binding of services.

We approach the application management goal by seeking solutions for the following management challenges.

- Creating an information element – a federation contract – that defines the application structure and the mutual interaction patterns of its components so that the contract can be used for governing the component services locally at each domain.
- Selecting the performing components for an application, so that the process takes into consideration shared business logic, conformity of components, and potential of successful communication between them.
- Uniform deployment and instantiation of components in a multidomain environment (both technical and organisational domains involved).
- Distributed deployment of bindings between peer components in a multidomain environment (both technical and organisational domains involved). This includes automatic configuration of channels and insertion of necessary interceptors and bridges into them.

2.2 Federation contract issues

Inter-organisational applications need to be managed based on explicitly stated, platform independent contracts. The contract must capture the platform-independent structural model for the application. The contract must also capture the performing components, in terms of their services and technological properties, or alternatively, in terms of their identities and locations.

Modern IT systems are expected to run nonstop applications. For example, telecom service providers cannot afford downtime for system upgrade, and may not be able to synchronize version changes across the network. Because the providing organisations are autonomous and can change versions asynchronously, multiple versions of components should be present and available simultaneously. Therefore, application components should be replaceable while the application (federation) is active, and if the upgrade of a component fails for some reason, the system should default back to the latest working configuration.

Thus, it is essential that the contract can change during the application lifetime: components can be changed because of upgrade needs or failures. For example, an upgraded service component can be introduced to the system, to coexist with previous versions as an independent service. The software engineering history of the component is not relevant for the runtime environment. The application can be triggered to component upgrade. The new component can be started, configured and prepared for quick switchover without affecting the running application. If the new component does not fit in into the application environment, it does not pass the phase where interoperability and conformance tests are performed, and the upgrade event fails and the application continues to use the old configuration.

Also related to the need for nonstop applications there is a requirement of allowing the application structure to evolve while the application is active. For example, a banking application can alternate between two phases, one for opening time when all interactions are allowed and another for off-time when only a few services are running. A more technical example is a new platform service that is introduced and should be used by applications for additional functionality.

Thus, in terms of the federation contract, there is a need for separate epochs. An epoch is a period where a defined set of services are provided and the community of components working towards that goal remains stable in structure.

2.3 Component selection issues

Selecting components for the application involves tests for component conformity. Conformity of components covers questions such as interface type matching, component behaviour compatibility and contract breaches, and expected binding semantics, to mention a few. In a multi-organisational environment, components involved in a federation cannot inherit properties that would ensure interoperability between components. Therefore, several levels of interoperability problems must be solved based on explicit meta-information exchange.

First, the participants should have matching views of the logical business process they are involved in and matching policy decisions where alternative cobehaviour models are possible. For example, in a tourist service application, where clients can make hotel reservations through a tourist office, there might be alternative payment methods, such as prepaid vouchers for hotels or credit card payment after the visit, embedded as alternative behaviours into a shared application logic.

Second, there should be matching views of the computational communication solution involving the participants. For example, a payment interaction can be computationally implemented by a sequence of protocol messages between buyer, seller and bank, and each party should have similar assumptions about the message formats and ordering; if not, some bridge solutions should be used between the parties for creating a coherent view.

Finally, there should be matching views of the engineering of the communication mediating solutions. For example, each party expects the transport protocol to preserve message sequences or support transaction transparency; if that is not the case at some domain, additional services can be used to intercept and upgrade the transport service.

Because the component interoperability matching should ensure technical interoperability, matching application interface structures is not sufficient alone. The underlying communication platform properties and the set of supporting protocols must be matched too. A concept of binding type or connector (e.g., [3, 2]) captures the set of related interfaces at peer components, and communication patterns between those interfaces. For complex binding needs, the binding type is considered as an integral part of an extended component interface description. For example, a heterogeneous networking environment, stream interfaces and multiparty communication all present complex bindings.

The conformance testing should in principle be based on specification matching, but this is too heavy for runtime usage. Instead, the matching can be supported by a two-sided repository service: one one hand, slow specification matches can be performed semiautomatically and the resulting relationships stored, and on another hand, a name-based matching process is provided for runtime matching.

2.4 Deployment and instantiation issues

Uniform deployment of components in a heterogeneous environment is possible when based on platform independent descriptions of components and on a defined mapping onto specific platforms. For example, a service type description only reveals the kind of interface a client interacts with, but there are numerous alternatives for provision of that service, ranging from a monolithic object to a distributed configuration with replication for load balancing.

The models and mappings can be used in a software engineering process resulting in an application implementation on a specific platform, as intended in the MDA. The MDA basic concepts are a platform-independent model (PIM) of the application business logic and a platform-specific model (PSM) that faithfully supports the PIM but also includes implementation design details for a specific platform. Semiautomatic tools are planned for making the PIM to PSM translation easy. The tools are based on standard patterns for implementing some computational features.

However, the models and mappings can also be used as a reuse mechanism that matches together service requirements in a federation contract and service providing components. Here, the whole PIM is not directly mapped to a single PSM. Instead, the PIM is split into parts (business architecture roles), each of which match onto a set of alternative PSMs. It is not necessary to have all partial PSMs selected for the same platform technology, but intermediate PSMs can be inserted where necessary.

The mapping process from the whole PIM to a complete PSM can be automated. The relationships between partial PIMs to corresponding PSMs need to be stored into a repository as well as the intermediate PSMs available.

In this paper we focus on the reuse style. The business architecture definition introduces requirements for the components to be selected, either at design time or at runtime. However, the approaches are not exclusive, but can be combined.

2.5 Federated binding issues

Current trend in component-based software engineering requires, with good reason, that connectors between components become first class elements in software architectures. However, the bindings between two or more component interfaces can be viewed both at computational and engineering levels. The computational level can be considered to be visible for application component programmers, the engineering level only to system service programmers.

At the computational level, we choose to consider bindings as component relations with some properties, such as binding type, QoS agreements and transparency support expected from the underlying communication platform. The properties can be inherent for the programming language concepts or system environment. At engineering level, we choose to use an explicit binding object to model the communication connection. The binding object captures the binding properties in an explicit, technology-independent contract.

The federated bindings are also governed by contracts. A binding contract captures the binding type supported and the required structure of the communication channel. The communication channel structure is determined by identifying the transparency support required, transport protocols, and peer component locations.

For the multidomain environment the model has to be extended so that the abstract binding object is in fact supported by a group of cooperating, domain specific binding objects. To overcome the technical differences between domains, the bindings are managed according to the agreed binding contract that is expressed in platform-independent terms. At each domain, local mappings from the contract to the implementation can be done separately.

3 Infrastructure services for management

A set of suggested management services, located into middleware, are briefly presented. Each of these services manipulates meta-information elements that are expressed in a universal management language. The meta-information structure is introduced, together with the requirement analysis of the language.

3.1 Pervasive management services

Middleware services for management fall into two categories: cooperative management services for multidomain applications and local element management services. We refer to ORBs, CORBA services, applications, and management applications by the term element.

We suggest that CORBA Management Services would include the following services.

- Enhanced trader for populating business architectures with selected components [17]. The business architecture description contains roles as placeholders for performing components, and each role description can be used as a selection rule. However, the selections for neighbouring roles is not independent, due to, for example, the need for shared binding requirements. The effect of architecture population is that of component composition or component reuse.
- Standard trading service [20] for maintaining a repository of service offers, for the use of the enhanced trader.
- Federation manager for negotiating, maintaining and renegotiating the federation contract that represents an application instance.
- Enhanced version of ODP type repository [9] for holding relationship information between generic types (service types, binding types, interface types) that are technology-independent and used for matching purposes and technology-dependent templates that are used for instantiating the corresponding components and objects [14]. This mapping information is created by system programmers separately from business architecture descriptions and service offers.
- Service deployer for instantiating components for each role according to the contract that represents the application instance. The service deployer uses type repository information to map the contract onto appropriate technology solutions [17].
- Binding factory for instantiating communication channels between components. Because no remote instantiation service is supported across organisational boundaries, the binding factories at each computing system involved must cooperate. Again, the factories use repositories for mapping contract information onto appropriate engineering solutions [17, 16].
- Implementation repository for storing software packaging and maintaining their automatic installation scripts.

The local element management services add lifecycle services and local bindings to this list. Other management services needed for other goals than inter-organisational application needs are here omitted.

The cooperative management services – enhanced trading, trading, type management, federation management, federated binding – are all services that have a local server running at each domain. These active agents take care of making requests to their peers at other domains, as there is no authority to otherwise invoke management actions at a foreign domain [15]. The requests carry contracts to pass relevant meta-information that identifies what should be done and how.

3.2 Management language issues

The meta-information exchanged by management services is an instance of a common, abstract management information model (MIM). Each component, platform entity or application is considered as a managed object. Platform-specific agents maintain the actual management data near the managed object, and provide a view to it to the manager according to a shared MIM.

The fundamental MIMs of the multidomain applications are those of

- federation contract,
- binding contract,
- service offers,
- component descriptors, and
- platform descriptors.

Some structuring rules are common to all MIMs within the same category, but some structures are dependent on the type of their elements. For example, the service offer structure is dependent on the service type in question, and federation contracts are structured according to the application structure. Furthermore, as interoperability solutions evolve, the rigour of MIM structure increases. Therefore, the MIMs need to be stored and made available at runtime.

The pervasive management services must provide a uniform language to express the various management information aspects involved. The management language must be technology independent because contracts are expressed across heterogeneous technology domains. The language expressions need technology dependent mappings maintained by each domain.

The federation contracts should be able to express

- large scale PIMs i.e. business architecture descriptions, in terms of roles, interactions, and policies;
- dynamic PIM structures, parameterisable by policies for alternative behaviour patterns and containing epochs;
- assignment rules for selecting components into the business architecture roles; and
- separate federation contracts at platform and at application level.

The service offers should be able to express

- component properties in platform-independent terms, i.e. the abstract interfaces and a behaviour model with policies; and
- components' platform requirements;

The binding contracts should be able to express

- binding type that indicates the assumed application interfaces of each peer;
- channel structure in terms of transport protocol and additional services required.

Finally, platform descriptors should express platform properties, e.g. transport protocols supported, and transparency support.

4 Management approach in Pilarcos

In the following, we consider the inter-organisational application lifecycle and management operations supporting it illustrated by the work performed in the Pilarcos project.

The lifecycle elements are

- definition of the application structure;
- deployment of the application, i.e., population of the application structure and subsequent instantiation of the involved components and bindings;
- partial repopulation;
- epoch change; and
- termination of the application.

4.1 Business architecture description

The essence of an application architecture is the technology-independent business logic description. The description must cover the logical processes involved, flow of abstract information through the multi-organisational system, and business responsibilities of each participant. Performing components are then selected to take care of the responsibilities set, to obey the set operational policies, and to implement protocols for information exchange modeled.

In the Pilarcos project, the application structure definition is called a business architecture description [17, 31]. The description expresses how the service is provided as a cooperation of some performing components. The cooperation is specified in terms of roles, i.e. placeholders for components, and rules for selecting performers for those roles.

The ODP enterprise language [13], that is used as a basis for the ad-hoc business architecture notation used in the Pilarcos project, introduces concepts for defining cobehaviour of members in a community. A community is a configuration of enterprise objects with a contract on their collective behaviour. The community specification includes

- a set of roles; the role specification gives requirements and restrictions for the behaviour of an object;
- rules for assigning enterprise objects to roles; the policy rules can address individual objects or relationships between objects, and can make restrictions on behavioural and non-behavioural properties of the potential objects;
- policies that apply to roles; policy values act as selectors on alternative behaviours for the objects – and thus also for the community;
- description of behaviour that changes the structure or the members of the community during its lifetime.

Each role in the community specification denotes a possible behaviour. The behaviour descriptions are refined with policy statements indicating which parts of the behaviour are prohibited, permitted or obliged to take place and under what conditions.

A role can be populated by an object that represents another community. In this way, larger systems can be composed of subservices. Functional composition is better supported by inclusion of multiple community specifications into a system specification and definition of the relationships between communities.

A community specification may be divided into several epochs, each presenting a different set of services supported by the community. For instance, a service might have a configuration phase and an operational phase; during the configuration phase only a management interface is available, during the operational phase also the actual service interfaces.

An example application is described in Figure 1. The example captures roles, bindings between roles for interaction relationships between the roles, and some policies. Semantically, the example application is a tourist service that composes subservices such as hotel reservations from other organizations. For the federation, hotel services are searched and accepted as potential performers in the subservice role if the payment policy appears to be an accepted credit card.

So far, we have focused on the semantics and requirements of the architecture descriptions, pushing forward the systematic description notation design.

An architecture definition language typically models the basic elements of

- components for computation and information storage,
- connectors for interaction patterns and rules, and
- configurations for expressing topologies of components and connectors [19].

We have chosen components to be large and highly abstracted, modeling business services or even complete IT systems. The component descriptions in this case are not only platform-technology independent, but may capture only criteria for the selection of a component. The selection criteria capture the offered and used interfaces of the component, the behaviour patterns the component is technically capable of participating, and the policies that restricting the actual behaviour.

For connectors we have two separate requirements. First, for the application programmers and business architecture designers the connectors are hidden and just the connectivity within the configuration is of interest. Second, the underlying middleware managing the application brings in connectors that are developed as first-class components. We also expect to have multiparty connections.

For configurations we required notations for dynamic changes.

We considered a group of existing architecture languages (for example, Darwin [18], ACME [8] , Wright [1] and LEDA [7]) and UML as alternatives for the language notation. We found LEDA and Wright interesting because they both provide behaviour concepts, LEDA provides also dynamic configurations. However, none of these fulfilled all our needs, as selection criteria have not been within the interests of architecture language developers.

```
policy framework TouristInfoPolicies {
    DoubleInterval Price;
    string Area;
}
policy framework PaymentPolicies {
    StringSet of
    { "VISA", "AMEX", "Mastercard" }
       PaymentMethod;
}

service type TouristInfoService {
    interface type
        TouristInfoInterface touristInfoI;
    interface type BillingInterface billingI;
    policy framework
        TouristInfoPolicies touristInfoP:
        attached to touristInfoI;
    policy framework PaymentPolicies paymentP:
        attached to billingI;
}
service type PaymentService {
    interface type BillingInterface billingI;
    interface type PaymentInterface paymentI;
    policy framework PaymentPolicies paymentP:
        attached to billingI, paymentI;
}
service type TouristInfoClient {
    interface type
        TouristInfoInterface touristInfoI;
    interface type PaymentInterface paymentI;
    policy framework
        TouristInfoPolicies touristInfoP:
        attached to touristInfoI;
    policy framework PaymentPolicies paymentP:
        attached to paymentI;
}

architecture TouristInfoArchitecture {
    role client {
        service type TouristInfoClient;
        < behaviour description i.e.
          restrictios for operation sequencing
          in terms of prohibitions,
          permissions and obligations >
    }
    role server {
        service type TouristInfoService;
        touristInfoP.Price = [0, infinity];
    }
    role paymentMediator {
        service type PaymentService;
    }
    binding touristService amongst
       (client.touristInfoI, server.touristInfoI) {
        # Policies related to this binding
    }
    binding payment amongst
      (client.paymentI,
       paymentMediator.paymentI) {
    }
    binding billing amongst
      (server.billingI,
       paymentMediator.billingI) {
    }
}
```

Figure 1. An example business architecture [26].

A further language development direction would most probably propagate the identified needs to UML; this process is already in progress by OMG as a result of MDA needs to change focus from implementation design language to modeling.

In the Pilarcos project, the current prototype software uses computational architecture descriptions as the federation contract baseline. However, the abstraction level might be higher, and allow computation-independent contract structure to be used. In that case, more relationship information would be needed to map business services first into local computational solutions at each domain.

Because the business architecture descriptions form a semantical basis for the federation contract, the descriptions need to be stored and available for all participants. It is promising that MDA has brought up the need for explicit models between application components for improving the markets of domain services [23].

4.2 Service offers and component management information

Components are implemented and developed independently from the overall application and from other components. A component implementation follows a PSM and can be done with the help of advanced software engineering tools introduced by the MDA.

The services provided by these components form a component market that is supported by service trading integrated into the middleware. However, the component implementations must be made instantiable first. For this purpose, the implementations are registered to a computing system via an implementation repository that maintains packages, automatic installation scripts, etc. From this repository, a uniform service deployer process is able to pick up the necessary implementations and locate the instances according to service type and platform property requirements. The service deployer is local to the administrative domain in question. Once the component implementation is made usable at a platform, it can be made available on the service market.

Because the performing components are created and managed separately, independent component production teams can create service components through structured market, as the business architectures in common use give guidance on the market request. The more challenging task of declaring new business architectures can be left for a smaller group of specialized designers. Because the controlling metainformation about the cooperation between application components is separated out from the application implementation, the application components become "dumb", highly reusable, and automatically personalisable for different user needs.

```
service offer for TouristInfoService {
 interfaces {
   touristInfoI = {
    "IDL:/repository/interfaces/idl/touristinfo:3.0",
     < references to contact point>, ...};
   paymentI = {
    "IDL:/repository/interfaces/idl/payment:3.0",
     < references to contact point>, ...};
   billingI = {
    "IDL:/repository/interfaces/idl/billing:3.0",
    < references to contact point>, ...};
 }
 policies {
     paymentP.paymentMethod =
        any of { "Visa", "AMEX" };
     touristInfoP.price = 5;
     touristInfoP.paymentPolicy = {"prepayment"];
     touristInfoP.offeredServices =
        { "HotelInfo", "TravelInfo"};
     touristInfoP.supportedTerminals =
        {"PC-SVGA", "PDA"};
 }
 requires < platform requirements >
 requires < binding type >
          < channel type >
}
```

Figure 2. An example service offer [26].

As a benefit, the business architecture description fulfills the need for supporting reuse and component composition, and even supports introduction of new services and component versions by allowing several versions to live simultaneously. Also dynamic application configurations can be easily supported.

The provided components are made available by exporting their service offers to the trading service. A service offer lists the service types supported by the component, policy values that can be applied, and technology requirements of the component, especially in respect to communication. In the MDA terms, the service offers include a partial PSM description as it only reveals externally visible features of the service, such as contact point for the service (or its transparent deployer). An example service offer is given in Figure 2.

The contents of the service offers slightly varies in a range of industrial products and research projects, depending on the underlying platform on which the traded elements run and on the ontology used to describe their properties. Here, probably a set of ontologies is needed to capture properties typical for a business domain and to capture properties of communication subsystems.

For the Pilarcos service offers, the speciality is in communication support requirements formed by the needs of federated binding process.

4.3 Application population and repopulation

To run an application, the performing components need to be selected and the result captured into the federation contract. The contract carries references to all required information for the application components' instantiation and binding. The process of selecting cooperable components for each role is called a population process. In the MDA terms, the population process maps a PIM to a set of cooperable PSM elements.

The selection of components is based on service offers and is run by an enhanced trading service. The population process first retrieves potential offers for each role based on the service types and other static properties. Then, alternative population combinations are tested in sequence for viability. The service offers include for example binding-related information that is not part of the architecture description, and therefore, the first offer retrieval cycle may bring in offers that are not interoperable with others. Not all combinations are tested, the process runs either until it reaches a time limit or finds an ample amount of viable populations.

The population process results in a federation contract. It contains a business architecture reference and references to the selected component description for each role. The components that are selected are known to be interoperable, although they may need some proxy or translator elements to mediate communication. The components are also known to be configurable by those shared policies, the values of which are determined during the population process.

Repopulation allows replacement of some components of the application either for fault-tolerance or for component upgrades. The replacement components are searched again according to the selection criteria expressed in the business architecture description.

In principle, the business architecture can be populated at different times: design, installation, instantiation, or even runtime. At design time, the designer might use repositories of components and reuse appropriate ones. The application would then be instantiated by calling identified components at each organisational computing system.

For runtime population, advanced middleware services have a fundamental role. The population process requires an extended trader-like facility for selecting matching components across organisational boundaries and for selecting interoperability solutions to be used. It also calls for component factories that have a platform-independent interface and language for requesting components to be created, but include a platform-sensitive selection mechanism and create platform-specific components.

This late deployment approach supports management of heterogeneity between technology domains and protects from the effects of asynchrony of service development cycle at different administrative domains. It also supports the evolving market of component implementations.

Consequently, it is possible that there is no full-featured static platform-dependent model of an application, except of a contract formed at runtime, when the players become chosen and connected. This is in contrast to the expectations on traditional software engineering methods.

4.4 Contract-based instantiation

Logically, the application is presented by the federation contract produced by the population process maintained by a contract manager. The federation contract contains a business architecture description and a service offer for each role. The contract is expressed in platform-independent terms and can thus be changed using a platform-independent management language. The contract manager provides an interface with operations for initiating repopulation, epoch change, termination, and renegotiation of policies.

The component instantiation processes involved in the application startup take place in a distributed fashion because there is no shared authority allowed to make direct management operations at the sovereign domains. The contract manager can only request certain cooperative management operations to be performed.

The components referred to in the federation contract are instantiated by a uniform, but platform-aware instantiation process. In this process, the technology-independent contract concepts are mapped onto technology solutions.

To organize the mappings, the ODP concepts of type and template can be used. A type is a predicate on some entity, characterizing some features of the entity. For example, a PIM can be considered a type. A template is more specific, carrying enough information for instantiation on a specific platform. A very detailed PSM might be a template, given appropriate tools for direct code generation, installation facilities and factory services.

For matching and instantiation purposes, the Pilarcos project uses an enhanced ODP type repository. The type repository supports operations for registering types (and templates as a subtype) and their relationships. We group together, under a PIM of an element, different PSM versions. The PIM serves as a type description, and the PSM elements as implementation selectors. The relationships that collect together the group carry references to necessary transformator components, when appropriate.

The federation contract covers also shared policy values for components to follow. The components should be parametrisable or (re)configurable as suggested in CCM (CORBA Component Model) draft. A technical example of already existing policies are activation and threading policies. However, as the domain application PIMs are developed, also policy frameworks related to them become better understood. This would allow business policies to be incorporated into components.

4.5 Federated binding

The federated binding process [16] is initiated asynchronously after the federation between application components has been established. All binding contracts have their initial agreements specified during the federation establishment, as the federation establishment process selects the components for the application in such a way that the required facilities for the bindings are available. The service offers provide information about the application services, but also express the requirements for the binding object architecture to be selected.

The logical binding object is split into domain specific binding objects. At each domain, a channel controller is created to represent the binding and manage the binding contract. The channel controller is responsible of instantiating the channel implementation through local factories.

For channel management, the channel controller will either be able to add or delete channel elements (stubs, translators, protocol objects, etc.) [6], or use a deployment script to instantiate the whole group as a unit [2]. Whichever design is used, the overall PSM will be selected from a repository based on the information carried in the binding contract (formed starting from the information available at service offers). The solution here resembles the design time selection of connector elements in a middleware repository in Aster [5]. Also a trader-like mechanism has been suggested to locate suitable binding factories [24].

It should be noted that each party only assumes that the peers have similar structure for the channel, although in reality, there might be a rigorous configuration of interceptors, bridges and additional application components at place.

The federated binding process is consistent with the ODP IRB standard [12] which also explains how interface references can be expanded to carry the above described additional binding contract information. The Pilarcos binding mechanism differs from the recursive binding of SUMO [4] by choosing the whole protocol stack simultaneously, thus keeping the number of negotiation rounds fixed, but using late and lazy instantiation of the channel.

4.6 Epoch change and termination

An epoch change means the change of provided services and the change of roles in the application. An epoch change is triggered by an explicit action in the community or by an external management action that is directly addressed to the contract manager of the application. Epoch change preserves those components that fulfill surviving roles, uses repopulation to search new components to the new roles, and removes those components that fulfil terminating roles. The role constellation for the new epoch has to be included in the business architecture description, as well as the triggering event.

Termination of the application invalidates the federation contract and allows termination of services and bindings according to their termination policies. It is possible that the same elements are used for other applications simultaneously, so they may need to outlive a single federation.

4.7 Status and future work

The Pilarcos project is in progress, having produced a first prototype version and working on a second. The first version focused on the population process and the Pilarcos enhanced trader facility [31].

The Pilarcos trading system is currently able to populate federations based on architecture descriptions like illustrated in Figure 1. The rigor of the architecture description language does not yet support epochs or complex community systems, nor does it include behaviour descriptions. The Pilarcos trading system uses the standard CORBA Trading service to store the service offers. Currently there is only a single trader involved, forming an expected bottleneck to the system. Further studies try out the effect of federated traders.

Interoperability tests between peer component interfaces are performed by the type repository. The first version was plainly the Trader type management interface. The second version we are currently working on will trust heavily on predefined relationships between interface types and banging types respectively. The processes for some heuristical rules for creating these relationships automatically have been discussed both within this and an earlier project [16] and within the ODP type repository standard group.

The service deployer tasks were initially mapped to functionalities for installing CORBA component packages, and instantiating and locating services. In the first Pilarcos prototype, all bindings were created across CORBA ORBs, and no heterogeneity of application interfaces were presented.

The second version focuses on the federated binding mechanism and heterogeneity of the used technologies. In addition to CORBA platforms, application components and some infrastructure services will be developed for EJB platform.

5 Conclusion

This paper summarizes the suggestions we provide for the OMG CORBA management RFI requesting information on the management requirements on CORBA systems, CORBA-based applications and interoperability needs. The focus in this paper is on the new standardization issues we believe need to be opened.

We suggest a new category of services for CORBA middleware, the cooperative management services to take care of the needs of multidomain applications.

The model suggested here is applicable not only on CORBA platforms, but on any other component-based or object-based platform too. Requirements for interoperability support are minimal: shared means to express services and locations. There is no embedded requirements for example to unify component models, because such decisions are left internal for each technology domain.

As this paper is directed for standardization efforts, we need to relate the model to those present in OMG target environments, i.e. JINI, Web Services and UDDI (Universal Description, Discovery and Integration); .NET being one of the UDDI users.

In the JINI [28] distributed computing environment, services can announce their presence through lookup services and implementations can be transferred to the client system automatically. The implementations are portable because a shared platform, Java Virtual Machine, is required. The major concern is in security – code that cannot be verified and managed with local administrative routines must be trusted as is. The sandbox solutions for disallowing certain actions from foreign code solve the problem only partially.

The Web Services goal is faster integration of applications into B2B solutions [30]. Web Services are self-contained business applications that use Internet standards at their interfaces. The self-containment idea is similar to the sovereign service components presented above, although the acceptable communication protocols are restricted to Internet protocols. The UDDI registry acts as a yellow pages service for Internet accessible services, into which service providers themselves can export their information. The UDDI registry is analogous to the use of object trading service described in Section 3. Also Web Services and UDDI development line is still looking forward to a universal service interoperability protocol [29].

The major feature that makes the Pilarcos approach different from the others is that instead of moving stubs or application code to a target computer possibly in a remote domain, the Pilarcos architecture uses platform-independent contracts for expressing what kind of elements should be present and where, and lets the local management services map that need to local technology solutions. Although not discussed in this paper, this restriction is significant element also in support of privacy and security of computing systems.

All of these systems promote evolution of open service markets for business services and computational components; automatic access to these markets should become an integral part of the global computing infrastructure. Achieving this requires some major development in terms of joint understanding on the reasonable structure of service markets and development of cooperative management services to support these markets. Furthermore, the components at markets should be guided towards commonly used roles within business architectures.

6 Acknowledgements

This article is based on work performed in the Pilarcos project at the Department of Computer Science at the University of Helsinki. The Pilarcos project is funded by the National Technology Agency TEKES in Finland, together with Nokia, SysOpen and Tellabs.

References

[1] R. J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.

[2] D. Balek. Connectors in Software Architectures. PhD Thesis, Charles University, Czech Republic, 2002.

[3] A. Berry and K. Raymond. The $A1\surd$ Architecture Model. In *Open Distributed Processing; Experiences with distributed environments, Proc. of the 3rd IFIP TC6/WG 6.1 International Conference on Open Distributed Processing*, pages 55–67, Brisbane, Australia, 1995. Chapman and Hall.

[4] G. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley Publishing Company, 1997.

[5] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarra. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. In *Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, New York, USA, Apr. 2000.

[6] G. S. Blair, G. Coulson, N. Davies, P. Robin, and T. Fritzpatric. Adaptive Middleware for Mobile Multimedia Applications. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1997.

[7] C. Canal, E. Pimentel, and J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *First Working IFIP Conference on Software Architecture*. Kluwer Academic Publishers, 1999.

[8] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169– 183, Toronto, Ontario, Nov. 1997.

[9] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Type Repository Function*. IS14746.

[10] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 2: Foundations*, 1996. IS10746-2.

[11] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 3: Architecture*, 1996. IS10746-3.

[12] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – ODP Interface References and Binding*, Jan. 1998. IS14753.

[13] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Enterprise Language*, 2002. DIS13235.

[14] P. Kähkipuro, L. Marttinen, and L. Kutvonen. Reaching Interoperability through ODP type framework. In *TINA'96 Conference: The Convergence of Telecommunications and Distributed Computing Technologies*, pages 283 – 284. VDE Verlag, Aug. 1996. Extended abstract.

[15] L. Kutvonen. Management of Application Federations. In H. Konig, K. Geihs, and T. Preuss, editors, *International IFIP Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, pages 33 – 46, Cottbus, Germany, Sept. 1997. Chapmann & Hall.

[16] L. Kutvonen. *Trading services in open distributed environments*. PhD thesis, Department of Computer Science, University of Helsinki, 1998.

[17] L. Kutvonen, J. Haataja, E. Silfver, and M. Vähäaho. Pilarcos architecture. Technical report, Department of Computer Science, University of Helsinki, Mar. 2001. C-2001-10.

[18] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Fifth European Software Engineering Conference ESEC'95*, Barcelona, Sept. 1995.

[19] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer / ACM Press, 1997.

[20] Object Management Group. *OMG Trading Object Service Specification*, June 2000. OMG Document formal/2000-06-27.

[21] Object Management Group. *Model Driven Architecture. Opportunities and Challenges*, Feb. 2001. Draft version 0.4. Document number ab/2001-02-03.

[22] Object Management Group. *Request for Information – CORBA Management Services*, 2001. OMG document orbos/2002-09-08.

[23] Object Management Group, Architecture Board ORMSC. *Model Driven Architecture*, July 2001. Document ormsc/2001-07-01.

[24] H. O. Rafaelsen and F. Eliassen. Trading and negotiating stream bindings. In *Proceedings of IFIP/ACM International Conference of Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, New York, Apr. 2000. Springer.

[25] J. Siegel. *Developing in OMG's Model-Driven Architecture*. Object Management Group, Nov. 2001. White paper, revision 2.6.

[26] E. Silfver, J.-P. Haataja, and M. Vähäaho. Pilarcos business case II – Enhanced Tourist Info Service. Technical report, Department of Computer Science, University of Helsinki, Mar. 2002. Internal Pilarcos project report.

[27] R. Soley. *Model Driven Architecture*. Object Management Group, Nov. 2000. White paper, draft 3.2.

[28] SUN Microsystems, Inc. *Jini Architectural Overview*. Technical White Paper.

[29] UDDI. *UDDI Technical White Paper*, Sept. 2000. http://www.uddi.org/pubs/UDDI_Technical_White_Paper.pdf.

[30] UDDI. *UDDI Executive White Paper*, Nov. 2001. http://www.uddi.org/pubs/UDDI_Executive_White_Paper.pdf.

[31] M. Vähäaho, E. Silfver, J. Haataja, L. Kutvonen, and T. Alanko. Pilarcos demonstration prototype – design and performance. Technical report, Department of Computer Science, University of Helsinki, Dec. 2001. C-2001-64.