Transformation of UML Models for Service-Oriented Software Architectures[*]

Ricardo J. Machado[†] João M. Fernandes[‡]
Paula Monteiro[†] Helena Rodrigues[†]
[†] Dep. Sistemas de Informação
[‡] Dep. Informática
Escola de Engenharia, Universidade do Minho
4700-320 Braga, Portugal

Abstract

The main aim of this paper is to present how to transform user requirements models into a software architecture for mobile applications. The technique (called "4SRS") is essentially based on the mapping of UML use case diagrams into UML object diagrams. UML sequence, activity, and state diagrams and other artifacts can also be considered within the transformation decisions. The applicability of this technique is illustrated by presenting some results from an e-government mobile application.

The development of mobile applications typically follow a service-oriented approach. A service is a software entity running on one or more machines and providing a particular type of function to a priori unknown clients. These services must communicate with each other, whose combination makes up a service-oriented architecture. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed, so workflow is a critical part of making services effective. When those services react to changes on user context, application are context-aware.

For mobile applications, the definition of the underlying service-oriented software architecture must consider the services themselves as user requirements, as well as the mobile operators entry-points and the final clients interfaces, and use them to characterize the platform.

1. Introduction

In the past few years, mobile communications and Internet technologies are enabling the access to information at any time and anywhere. In a mobile computing environment, the user interacts with a mobile information space with info-mobility context-aware services. These are services providing information relevant or useful to users in a given context. It is unreasonable to expect a user to manually configure his/her applications to provide the services he/she wants to use in a given instant, particularly when the "most appropriate" service changes as the user moves from one context to another. This is why mobile applications must be aware of the current computational and physical environment surrounding the user and dynamically modify their behaviour to suit the situation.

An effective software infrastructure for running mobile applications must be able to find, adapt, and deliver the appropriate services to the user computing environment based on his/her context. The current trend in the software industry is for service providers to supply reusable functions via application components called services. These services are described using a standard description language and, in the future, using standard ontologies. Such descriptions could enable automatic composition of services, which in turn enables an infrastructure that dynamically adapts to tasks. Building applications involves specifying the right compositon of services, building a user interface, and orchestrating the data flow among the various components [1].

Our approach for identifying the system components for such architecture (the conceptual model) requires the software engineer to start the development by defining the functional model (use case diagram) that reflects the system functionalities offered to its users from their perspectives [2] [3]. A system functionality is to be accomplished by the communication and activity coordination between system objects, and can be validated, at this level, by the construction of a specific workflow between system objects.

The problem of obtaining architectural design models from analysis models is not simple and easy and faces several difficulties [4]. Generically, it involves several decisions that cannot be done by a method or a tool, due to

* This work has been supported by projects USE-ME.GOV (IST-2002-002294), METHODES (POSI/CHS/37334/2001), and STACOS (POSI/CHS/48875/2002).

the natural discontinuity between functional and structural models. Holland and Lieberherr consider that the identification of objects and the description of the relationships between them are two of the three challenges of object-oriented design [5].

There are many works that propose solutions to tackle this problem, namely by guiding the transformation of use case models into object/class models [6] [7] [8] [9]. In [10] [11], two approaches are proposed for a use case rationale based on goal identification. These approaches can be used for a better supported transition for the architectural design issues. However, they lack an explicit scenario framework for capturing the semantic intentionality of each use case. This could be incorporated by adopting some scenario-based requirements engineering techniques, such as those suggested in [12] [13].

This paper presents a novel approach for solving a real problem faced by software engineers, which consists on the definition of the system objects based on the use cases and their respective textual descriptions. The strategy uses the object types (interface, data and control) defined in [6]. Our approach extends the results presented in [2] [3] [14] and it incorporates some mechanisms that allow software engineers to relate each object with the use cases that gave origin to it. Due to the relatively weak support of UML 1.5 to component-based design, we have decided to use the object concept of UML to represent system-level entities or components.

2. Requirements Modeling

Our approach for identifying the system components requires the software engineer to begin the development process by defining the functional model that captures the system functionalities offered to its users. Use cases are one of the most suitable technique for that purpose, since they are simple and easy-to-read. In fact, they only include three main concepts (use cases, actors and relations). This low number of concepts is a fundamental characteristics for involving, within the requirements capture process, non-technical customers and clients.

Although use cases are used in several object-oriented projects, they do not hold any intrinsic characteristic that can be classified as "pure" object-oriented. However, within the research community there is a large consensus on the recognition that use cases are a proper technique for object-oriented projects [15], namely for discovering and later specifying the behavior of the system, during the analysis phase. This is also highlighted by the fact that use cases are part of UML (Unified Modeling Language) notation [16]. Thus, adopting use cases for user requirements is undoubtedly a valid technique, but poses the problem related to the transformation of use cases into objects or components. This problem is one of the main topics of this paper.

The case study used in this paper to illustrate our transformation approach is a new open platform for mobile government applications, supporting usability, openness, interoperability and scalability. It is responsible for the deployment of a set of reusable service components that facilitates user context-aware application development that allow citizen to perform a set of high-level government related activities, allowing citizens to access the most appropriate services at any time, anywhere. Those activities include spontaneous communitity interactions such as reporting anomalies and making suggestions or getting thematic information on local events, city information, healthcare or education. As a particular scenario consider a public institution that offers a heterogeneous set of services for reporting a complaint. When the citizen whishes to report a complaint, the platform will support him/her by having a set of services coordinating such activity (use case {U0a.2.} in fig. 1).

The requirements for this system were acquired using requirements engineering techniques, and the end-result was a collection of artifacts, including UML diagrams. We next present some of the artifacts.

After identifying all the use cases of the system (right part of fig. 1), the next step is to describe their behavior. There are some alternatives for describing use cases, namely informal text, numbered steps with pre- and post-conditions, pseudo-code and activity diagrams [17]. Next, we present, as an example, the description of the top-level use case {U0a.1} with informal text. Similar descriptions were created for the other top-level use cases.

{U0a.1} send alert: *Send domain alert or disseminating domain information to the users informing of domain related events and situations or unexpected domain situations that are happening in the region. Only users that have previously subscribed this e-service will receive the alert messages (subscription made via {U0a.4} user profile subscription). This is an asynchronous e-service. If technically possible, the system acquires user context raw information (location, time, etc) from external context sources. Also, a contextualization process will assist the system in making the level of granularity of the information adequate to the geographic location of the user context (geographic location context, time context and activity context). For example: (ex. #1: an alert of a dangerous hole in a street should only be sent to the users geographically located in that street; ex. #2: an alert of a street obstructed should be sent to the users geographically located in that street or in any of the incident streets; ex. #3: an alert of weather storm should be sent to all the users in the region). The information as-*

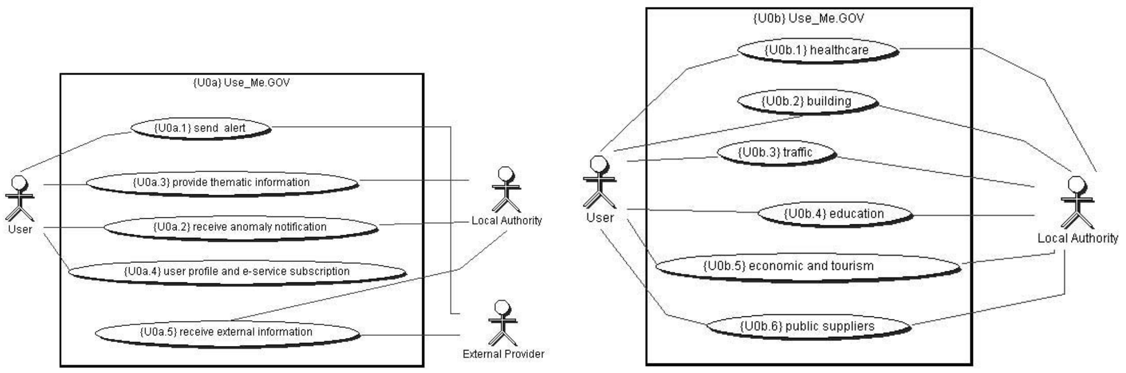

Figure 1. Top-level use case diagram according to two orthogonal criteria; Left: Functionality; Right: Domain.

sociated to the alert should always be up-to-date and match the user-specific request, excluding any extra information or undesired advertisements. For those users that require personalized information, a subscription must be made via {U0a.4} user profile and e-service subscription.

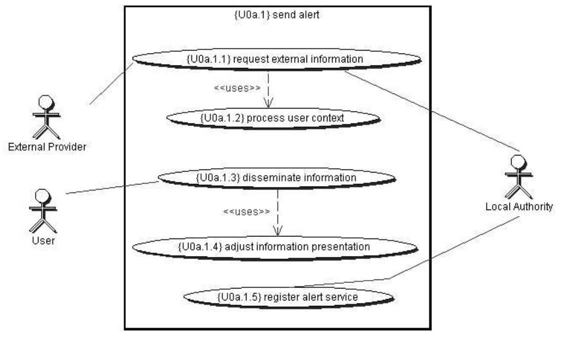
{U0a.1} send alert
{U0a.1.1} request external information
<<uses>>
{U0a.1.2} process user context
{U0a.1.3} disseminate information
<<uses>>
{U0a.1.4} adjust information presentation
{U0a.1.5} register alert service
External Provider
User
Local Authority

Figure 2. Refinement of use case {U0a.1}.

Since this use case was further refined, as the left diagram in fig. 1 depicts, we also present, as an example, the textual descriptions of two of its sub use cases.

{U0a.1.1} request external information: *The system explicitly requests for appropriate domain information, from the known external content providers (registered via {U0a.1.5} register alert service, for example), based on user context (location, time and activity), acquired and processed via {U0a.1.2} process user context, and/or based on user profile, processed via {U0a.4} user profile and e-service subscription.*

{U0a.1.5} register alert service: *The system registers domain alert services. When a local authority wants to offer an alert service, it registers it in the system providing the alert conditions and source of information. This registration will allow the execution of the following use cases: {U0a.1.3} disseminate information and {U0a.1.1} request external information.*

3. Model Transformation

Transforming uses cases into architectural models is a difficult task and in [3] we presented a technique called 4SRS (4-step rule set) to help on that task. At high-level, the 4SRS technique is organized as four steps to transform use cases into objects.

3.1. Step 1 - object creation

In this step, each use case must be transformed into three objects (one interface, one data, and one control). Each object receives the reference of its respective use case appended with the suffix (i, d, c) that indicates the object category (in our approach, object references start with an "O"). This is a fully "automatic" step, since there is no need to any kind of particular decisions or rationale for the specific context of each use case. From this step on, there is only objects as design entities. Use cases are still used in the following steps to allow the introduction of requirements into the object model.

In the case study presented in this paper, the 4SRS transformation was used in the 2nd level of use case refinement for orthogonal criteria a). This has considered as eligible use cases for this first step all 22 2nd level use cases (for criteria a): 5 from {U0a.1}, 7 from {U0a.2}, 8 from {U0a.3}, and 2 from {U0a.4}) plus use case {U0a.5} that had no refinements at all. This approach gave origin to 69 objects.

3.2. Step 2 - object elimination

In this step, it must be decided which of the three objects must be maintained to fully represent, in computational terms, the use case, taking into account the whole system and not considering each use case in isolation. These decisions must be based on the textual description for each use case. This step aims to decide which of the objects created in step 1 must be kept in the object model. This step also supports the elimination of redundancy in the user requirements elicitation, as well as the discovering of missing requirements.

This is the most important step of the 4SRS technique, since the definitive system-level entities are decided here. To cope with the complexity of the step, it has been decomposed into several micro-steps.

Micro-step 2i: use case classification
In this micro-step, the software engineer classifies each use case. To understand what the classification mechanism can be, we must first study how many combinations of objects of a given use case do exist. The 4SRS assumes that each use case gives rise to a maximum of three objects[1]: one interface-object (i), one control-object (c) and one data-object (d). Thus, we come up with 8 different combinations or patterns ($\emptyset$, i, c, d, ic, di, cd, icd), if we ignore the links among objects.

The idea behind this classification is to help on the transformation of each use case into objects. This classification would provide some hints on which object categories to use and how to connect[2] those objects. As an example of the execution of this micro-step, in the case study, the use case {U0a.1.5} was classified as being of type "i-d", which means that only the interface and the data objects are kept (the control object will be eliminatedin micro-step 2ii).

Micro-step 2ii: local elimination
The aim of this micro-step is to answer, for each object created in step 1, if it makes sense in the problem domain, since the creation of objects in step 1 is executed blindly, not considering the system context for the object creation.

1 In some situations, four or more objects can be created from the same use case. A typical example is the creation of two interface-objects, one for input and the other for output purposes. However, considering three as the maximum number of objects does not result in loss of generality.

2 Associations are explicitly handled in step 4.

Object {O0a.1.5.c} (control component of use case {U0a.1.5} *register alert service*) does not make sense in the problem domain, as it can be seen by carefully analyzing its textual description. Use case {U0a.1.5} is only responsible for allowing services registration by the local authorities (interface component) and the maintenance of a data repository of registered services configuration (data component). Thus, for the use case {U0a.1.5}, only the data and information components have been kept, giving origin to the inclusion of the objects {O0a.1.5.d} and {O0a.1.5.i} in the object model.

Micro-step 2iii: object naming
In this micro-step, objects that have not been eliminated from the previous micro-step must receive a proper name that reflect both the use case from which it is originated and the specific role of the object taking into account its main component.

As an example, objects {O0a.1.5.d} and {O0a.1.5.i} were named "*registered alert services*" and "*alert service registration*", respectively.

Micro-step 2iv: object description
Each named object resulting from the previous micro-step must be described, so that the system requirements they represent become included in the object model. These descriptions must be based on the original use case descriptions. It is not recommended to firstly introduce requirements at this stage that are not based on user requirements captured via use case descriptions. Exceptions to this recommendation are allowed, if they are classified as non-functional requirements (NFR) or as design decisions (DD).

The following object descriptions illustrate how to construct system requirements based on user requirements, by mapping use case descriptions into the corresponding remaining objects.

{O0a.1.5d} registered alert services: *This object stores the attributes about the services registered in the system (by "service" in this context we refer to the application functionalities that the system will offer to its users). This object must store the attributes that will enable service discovery, service aggregation, service automatic activation (when applied) and service access. A proper service description mechanism should be introduced. NFR1: This mechanism should be extensible, manageable, self explaining and semantically uniform. For service automatic activation purposes, this object must store information about internal and external alert conditions. This object must store information about the dependencies between registered services and external content providers. For context-aware service discovery, this object must store attributes defining* *the service context of use. NFR2: For scalability constraints, this must be a distributed object, mainly in a regional, national or international setting.*

{O0a.1.5i} alert service registration: *This object defines the system interface for service registration, deregistration, dynamic configuration and service attributes query. NFR1: For security constraints this interface must only be provided to the system components (not to the users).*

Micro-step 2v: object representation

This is the most critical micro-step of the 4SRS technique, since it supports the elimination of redundancy in the user requirements elicitation, as well as the discovering of missing requirements. Within the analysis phase, this micro-step constitutes an internal validation step that assures the semantic coherence of the object model and that discovers anomalies in the use case model.

In the case study, objects {O0a.2.1.d} *registered anomaly services* and {O0a.3.1.d} *registered thematic services*, that resulted from the use cases {U0a.2.1} *register anomaly service*[3] and {U0a.3.1} *register thematic information service*[4], are represented by the object {O0a.1.5.d} *registered alert services*. This means that, due to functional coherence, object {O0a.1.5.d} can represent a system-level entity within the object model that is able to inscribe its own system requirements, as well as those associated with objects {O0a.2.1.d} and {O0a.3.1.d}. This is an example of elimination of functional redundancy, since, from the functional point of view, is simpler and correct to support and justify the existence of a unique data repository for all kinds of registered services.

Micro-step 2vi: global elimination

This is a fully "automatic" micro-step, since it is completely based on the results of the previous one. Those objects that are represented by other ones, must be eliminated since its system requirements no longer belong to them. This micro-step is called "global elimination" due to its global awareness for generating a coherent and canonical object model, from the system requirements point of view.

In the example below, this micro-step gives origin to the elimination of the objects {O0a.2.1.d} and {O0a.3.1.d}, since the object {O0a.1.5.d} is representing the system requirements of those objects.

Micro-step 2vii: object renaming

This is the last micro-step of step 2. Its purpose is to rename the objects that have not been eliminated from the previous micro-step and that represent additional objects. The new names must reflect the plenitude of system requirements represented.

In the example, object {O0a.1.5.d} was renamed, since it is a not-eliminated object (from micro-step 2vi) and it represents not only itself, but also objects {O0a.2.1.d} and {O0a.3.1.d}. The chosen name was "registered services", so that all system requirements related with the "registered alert services", "registered anomaly services", and "registered thematic services" are properly represented.

3.3. Step 3 - object packaging & aggregation

In this step, the remaining objects (those that were maintained after executing step 2), for which there is an advantage in being treated in an unified way, should give origin to aggregations or packages of semantically consistent objects. This step supports the construction of a truly coherent object model, since it aids in introducing an additional semantic layer at an higher abstraction level, that works as a "functional glue" for the objects.

Packaging is a relatively immature technique, since it introduces a very light semantic cohesion between the objects. This cohesion can be easily reversed within the design phase, whenever needed. This means packaging can be flexibly used to allow a temporary obtainment of more comprehensive and understandable object models.

In the opposite way, aggregation imposes a strong semantic cohesion between the objects. The level of cohesion in aggregations is more difficult to reverse in next design phases, which suggests a more scrupulous approach in using this kind of "functional glue". This means, aggregation should only be used when it is explicitly assumed that the set of considered objects is affected by a conscious design decision. Typically, aggregation is used when there is a part of the system that constitutes a legacy sub-system, or when the design has a pre-defined reference architecture that constricts the object model.

In the case study, no aggregation was used, since there was no reference architecture, nor any kind of legacy subsystem. As an example of packaging, objects {O0a.4.1d}, {O0a.2.2d}, and {O0a.3.2d}, all originally obtained from different use cases, are kept together inside a common package (package {P7.} *user data*).

3 *{U0a.2.1} register anomaly service: The system registers available domain anomaly processing services in the region. When a local authority wants to offer a processing anomaly service, it registers it in the system providing relevant information such as context of use.*

4 *{U0a.3.1} register thematic information service: The system registers available domain thematic information in the region. When a local authority wants to offer a domain thematic information service it registers it in the system providing relevant information such context of use and references to external information providers that may complement the service.*

3.4. Step 4 - object association

This final step of the 4SRS technique supports the introduction of associations in the object model, completely based on the information existent in the use case model and generated in step 2i.

Regarding the information in the use case model, if the textual descriptions of use cases possess hints on the kind of sequences use cases are inserted in, this information must be used to support the inclusion of associations in the object model. As an example, use case {U0a.1.5} includes in its textual description the following sentence "*This registration will allow the execution of the following use cases: {U0a.1.3} disseminate information and {U0a.1.1} request external information*.". These hints were sufficient to include associations from the object {O0a.1.5.d} to the objects {O0a.1.1d}, {O0a.1.1.i}, and {O0a.1.3.c}. This kind of hint constitutes a pre-design decision related with the sequence of execution of use cases. If not carefully analyzed, it can give origin to premature scheduling decisions.

Alternatively, the use case model can include other kind of information to support associations, when there is UML relations between use cases. As an example, use case {U0a.1.1} ≪uses≫ use case {U0a.1.2}, which justifies the existence of an association between objects {O0a.1.1.d} and {O0a.1.2.c}, and between {O0a.1.1.i} and {O0a.1.2.d}.

Regarding the information generated in step 2i, it must be used to support the inclusion of associations between objects originally obtained from the same use case. As an example, in step 2i, use case {U0a.1.5} was classified as being of type "i-d", which implies that objects {O0a.1.5.d} and {O0a.1.5.i} must be associated.

4. Logic Architectural Model

In this section, we present the system architectural model, which in addition to an informal description of the objects, express the system requirements. Our main goal was to define a logic architecture for the system that could capture all its functional requirements and its non-functional intentionalities; the former gave origin to textual descriptions for each object in the model; the later have been classified as NFR and DD, design decision.

An object model shows how significant properties of a system are distributed across its constituent parts. Fig. 3 represents the raw object diagram, that identifies the system-level entities, their responsibilities and the relationships among them. Its purpose is to direct attention at an appropriate decomposition of the system without delving into details.

The packages used define, each one, decomposition regions, which contain several tightly semantically connected objects. Within the next design phases, these packages must be further specified, in what concerns its architectural structure, by using design patterns.

In fig. 3, we can identify a set of typical service-oriented components, such as the context processor component (package {P3.}*process context*), the service repository component (objects {O0a.1.5.d} and {O0a.1.5.i}), the service discovery protocol component (objects {O0a.2.5.c} and {O0a.2.5.i}), the user profile component (package {P7.}*user data*), the external providers components (packages {P1.}*external providers information* and {P2.}*external proviers interface*), as well as user interface components (package {P5.}*user interface*). These components were obtained by a decision-support technique that can help software engineers to reason about the characteristics of the service-oriented platform. To build applications on top of this architecture involves specifying the right compositon of services, building a user interface, and orchestrating the data flow among the various components. Configuring services and applications so they can be easily and reliably reused by other developers and composed into larger applications is a major chalenge, as already stated by [1].

The resulting raw object diagram can be used in the following development phases to support the definition of specific sub-projects, by using collapsing and filtering techniques. These techniques allow the redefinitions of the system boundary, giving origin, for instance, to the database project, services formalization, or platform pattern analysis.

Fig. 4 shows the collapsed object diagram that was obtained from the raw object diagram (fig. 3) by hiding packages details. Therefore, associations appear at a higher level of abstraction and the resulting object diagram is more nicely readable.

Fig. 5 shows the filtered object diagram that was obtained from the collapsed object diagram (fig. 4) by considering the objects {O0a.3.7.d}, {O0a.3.7.c} and {O0a.1.3.c} as one sub-system for design. This diagram was included here as an example of how raw object diagram can be used in the following phases of the product developing process to highlight parts of the system and allow sub-system specification and partition for sub-project execution within various teams. Criteria illegible for filtering can be dependent on project management issues, functional implementation domains, etc.

After obtaining the raw object model, it is possible to specify the complaint service described by use case {U0a.2.} (fig. 1), through a sequence diagram (fig. 6), which specifies the sevice as a workflow among object model components.

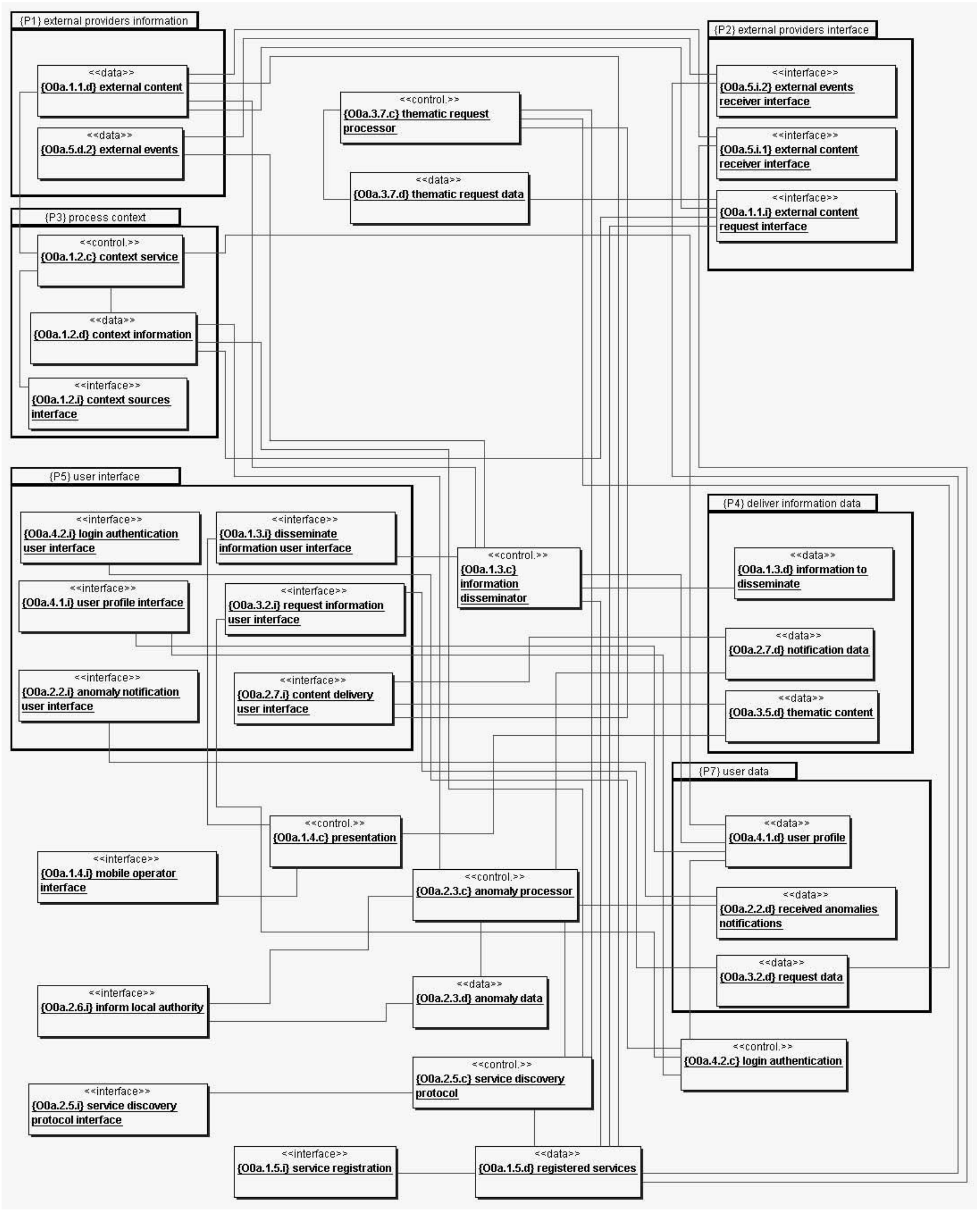
{P1} external providers information
<<data>> {O0a.1.1.d} external content
<<data>> {O0a.5.d.2} external events
{P2} external providers interface
<<interface>> {O0a.5.i.2} external events receiver interface
<<interface>> {O0a.5.i.1} external content receiver interface
<<interface>> {O0a.1.1.i} external content request interface
<<control.>> {O0a.3.7.c} thematic request processor
<<data>> {O0a.3.7.d} thematic request data
{P3} process context
<<control.>> {O0a.1.2.c} context service
<<data>> {O0a.1.2.d} context information
<<interface>> {O0a.1.2.i} context sources interface
{P5} user interface
<<interface>> {O0a.4.2.i} login authentication user interface
<<interface>> {O0a.1.3.i} disseminate information user interface
<<interface>> {O0a.4.1.i} user profile interface
<<interface>> {O0a.3.2.i} request information user interface
<<interface>> {O0a.2.2.i} anomaly notification user interface
<<interface>> {O0a.2.7.i} content delivery user interface
<<control.>> {O0a.1.3.c} information disseminator
{P4} deliver information data
<<data>> {O0a.1.3.d} information to disseminate
<<data>> {O0a.2.7.d} notification data
<<data>> {O0a.3.5.d} thematic content
{P7} user data
<<data>> {O0a.4.1.d} user profile
<<data>> {O0a.2.2.d} received anomalies notifications
<<data>> {O0a.3.2.d} request data
<<control.>> {O0a.1.4.c} presentation
<<interface>> {O0a.1.4.i} mobile operator interface
<<control.>> {O0a.2.3.c} anomaly processor
<<interface>> {O0a.2.6.i} inform local authority
<<data>> {O0a.2.3.d} anomaly data
<<control.>> {O0a.2.5.c} service discovery protocol
<<control.>> {O0a.4.2.c} login authentication
<<interface>> {O0a.2.5.i} service discovery protocol interface
<<interface>> {O0a.1.5.i} service registration
<<data>> {O0a.1.5.d} registered services

Figure 3. Raw Object Diagram.

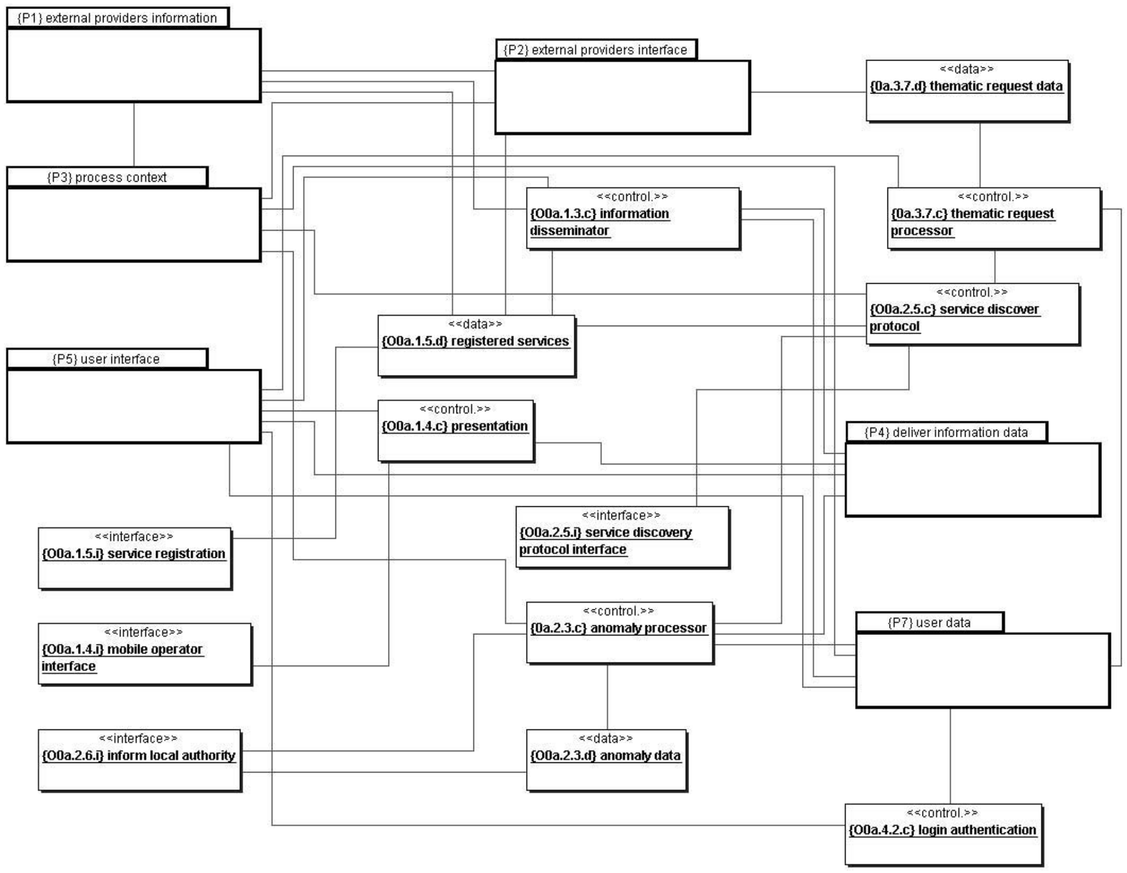
{P1} external providers information
{P2} external providers interface
<<data>>
{0a.3.7.d} thematic request data
{P3} process context
<<control.>>
{O0a.1.3.c} information disseminator
<<control.>>
{0a.3.7.c} thematic request processor
<<control.>>
{O0a.2.5.c} service discover protocol
<<data>>
{O0a.1.5.d} registered services
{P5} user interface
<<control.>>
{O0a.1.4.c} presentation
{P4} deliver information data
<<interface>>
{O0a.1.5.i} service registration
<<interface>>
{O0a.2.5.i} service discovery protocol interface
<<control.>>
{0a.2.3.c} anomaly processor
{P7} user data
<<interface>>
{O0a.1.4.i} mobile operator interface
<<interface>>
{O0a.2.6.i} inform local authority
<<data>>
{O0a.2.3.d} anomaly data
<<control.>>
{O0a.4.2.c} login authentication

Figure 4. Colapsed object diagram.

The user connects to the system and selects the complaint service. Object {O0a.2.2.i} receives the user request, stores the complaint description in object {O0a.2.2.d}, and contacts object {O0a.2.3.c}. Object {O0a.2.3.c} processes the user complaint request and it gets the user context from object {O0a.1.2.d}. This object contacts object {O0a.1.2.c}, which determines the needed external raw context sources, it receives user context information (through {O0a.1.2.i}), and it maps user context information into appropriate symbolic context information. If necessary, it starts a dialogue with the user to get more precise context information. The user context information is returned to object {O0a.2.3.c}. Taking into account the user context, this object determines which service is the most appropriate to receive the user complaint (through interactions with object {O0a.2.5.c}) and it contacts it through object {O0a.2.6.i}.

5. Conclusions

The correct derivation of system requirements from user requirements is an important topic in requirements engineering research. This activity assures that the design phase is based on the effective clients needs without any misjudgment arbitrarily introduced by the software engineers during the process of system requirements specification. One approach to support this derivation is by transforming user requirements models into system requirements models, by manipulating the corresponding specifications. User requirements are, typically, described in natural language and with informal diagrams, at a relatively low level of detail and are focused in the problem domain. System requirements comprise abstract models of the system, at a relatively high level of detail, and constitute the first system representation to be used at the beginning of the design phase.

This papers has presented a technique (called "4SRS") that aids software enginers to transform user requirements models into the first logical architecture of the system. The

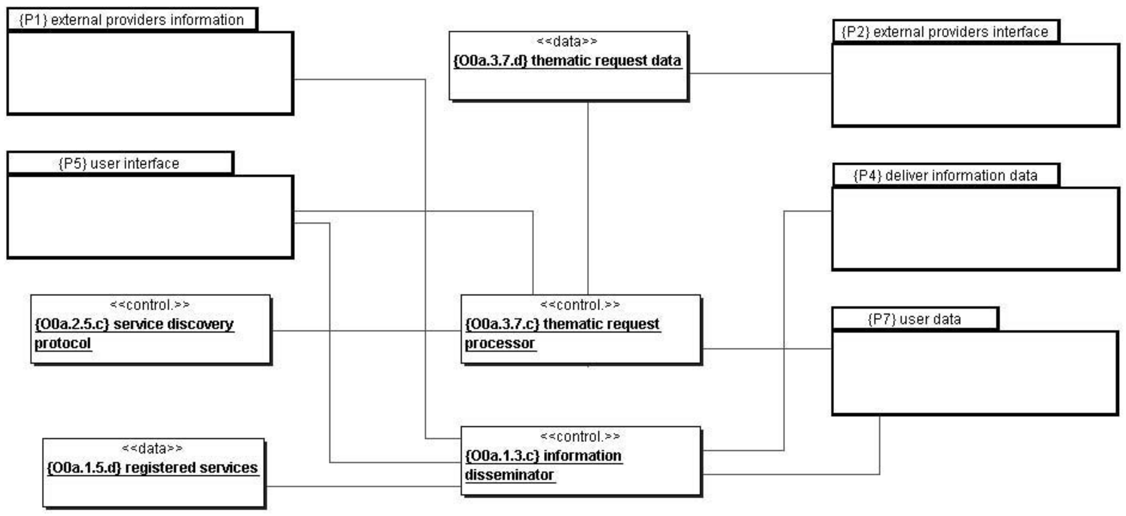
{P1} external providers information
<<data>>
{O0a.3.7.d} thematic request data
{P2} external providers interface
{P5} user interface
{P4} deliver information data
<<control.>>
{O0a.2.5.c} service discovery protocol
<<control.>>
{O0a.3.7.c} thematic request processor
{P7} user data
<<data>>
{O0a.1.5.d} registered services
<<control.>>
{O0a.1.3.c} information disseminator

Figure 5. Filtered object diagram.

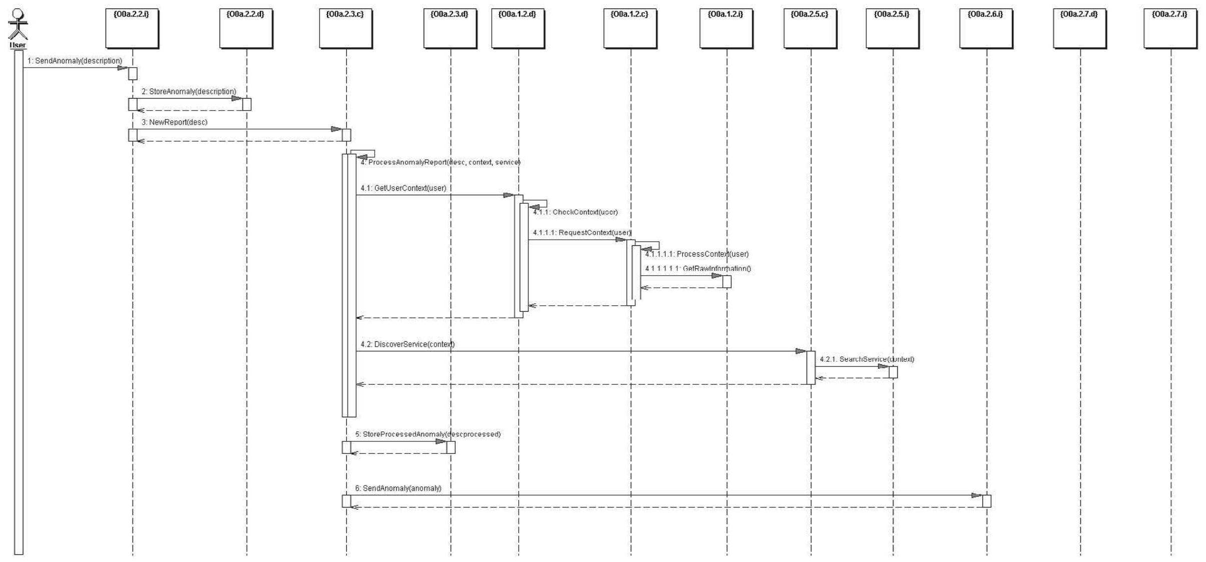
User
{O0a.2.2.i}
{O0a.2.2.d}
{O0a.2.3.c}
{O0a.2.3.d}
{O0a.1.2.d}
{O0a.1.2.c}
{O0a.1.2.i}
{O0a.2.5.c}
{O0a.2.5.i}
{O0a.2.6.i}
{O0a.2.7.d}
{O0a.2.7.i}
1: SendAnomaly(description)
2: StoreAnomaly(description)
3: NewReport(desc)
4: ProcessAnomalyReport(desc, context, service)
4.1: GetUserContext(user)
4.1.1: CheckContext(user)
4.1.1.1: RequestContext(user)
4.1.1.1.1: ProcessContext(user)
4.1.1.1.1.1: GetRawInformation()
4.2: DiscoverService(context)
4.2.1: SearchService(context)
5: StoreProcessedAnomaly(descprocessed)
6: SendAnomaly(anomaly)

Figure 6. Sequence diagram for the scenario "make a anomaly report".

transformation is based on the mapping of UML use case diagrams into UML object diagrams, but other diagrams and artifacts may also be used (namely, UML sequence, activity, and state diagrams).

The application of this technique is illustrated by presenting some results from the design of a real e-government mobile application. Taking into account the particularities of this case study, the obtained software architecture presents the typical functional components of mobile context-aware architectures. Within this example, the 4SRS technique has shown its usefulness by assuring the generation of a seamless specification of the service-oriented architecture requirements. This generation was supported by a decision-assisted rationale that helps software engineers to find the architectural requirements, completely based on the elicited user requirements.

This transformational approach shows that model continuity is a key issue and highlights the importance of having a well defined process to relate, map and transform requirements models.

Presently, the 4SRS technique forces the software engineer to unfold the use case hierarchy. As future work, we plan to devise heuristics that help to map directly use cases and their hierarchical organization into objects, so that the unfolding is not necessary. This would allow to maintain an hierarchical structure in the object model analogous to the existent hierarchy in the use case functional refinement tree. It is also planned to incorporate OCL inscriptions to support the formal description of non-functional requirements into the object model.

References

[1] Guruduth Banavar and Abraham Bernstein. Software Infrastructure and Design Challenges for Ubiquitous Computing Applications. *Communications of the ACM*, 45(12):92–96, 2002.

[2] João M. Fernandes, Ricardo J. Machado, and Henrique D. Santos. Modeling Industrial Embedded Systems with UML. In *8th ACM/IEEE/IFIP Int. Workshop on Hardware/Software Codesign (CODES 2000)*, pages 18–22. ACM Press, May 2000.

[3] João M. Fernandes and Ricardo J. Machado. From Use Cases to Objects: An Industrial Information Systems Case Study Analysis. In *7th International Conference on Object-Oriented Information Systems (OOIS'01)*, pages 319–28. Springer-Verlag, August 2001.

[4] H. Kaindl. Difficulties in the Transition from OO Analysis to Design. *IEEE Software*, 16(5):94–102, 1999.

[5] Ian M. Holland and Karl J. Lieberherr. Object-Oriented Design. *ACM Computing Surveys*, 28(1):273–5, 1996.

[6] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[7] Ralph-Johan Back, Luigia Petre, and Ivan Porres. Analysing UML Use Cases as Contracts. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 518–33. Springer, 1999.

[8] Doug Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Object Technology. Addison-Wesley, 1999.

[9] L. B. Becker, C. E. Pereira, O. P. Dias, I. M. Teixeira, and J. P. Teixeira. MOSYS: A Methodology for Automatic Object Identification from System Specification. In *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 198–201. IEEE CS Press, March 2000.

[10] Y. Liang. From Use Cases to Classes: a Way of Building Object Model with UML. *Information and Software Technology*, 45:83–93, 2003.

[11] M. Saeki and H. Kaiya. Transformation Based Approach for Weaving Use Case Models in Aspect-Oriented Requirements Analysis. In *4th AOSD Modeling With UML Workshop*, 2003.

[12] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting Scenario-Based Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(12):–, December 1998.

[13] A. van Lamsweerde and L. Willemet. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, 24(12):–, December 1998.

[14] João M. Fernandes and Ricardo J. Machado. System-Level Object-Orientation in the Specification and Validation of Embedded Systems. In *14th Symposium on Integrated Circuits and System Design (SBCCI'01)*, pages 8–13. IEEE CS Press, September 2001.

[15] Grady Booch. *Best of Booch: Designing Strategies for Object Technology*. SIGS, 1996.

[16] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology. Addison-Wesley, 1999.

[17] Geri Schneider and Jason P. Winters. *Applying Use Cases: A Pratical Guide*. Object Technology. Addison-Wesley, 1998.