Requirements Analysis for Middleware Aspects

Charles Zhang and Hans-Arno Jacobsen
Middleware Systems Research Group,
Department of Electrical and Computer Engineering,
Department of Computer Science
University of Toronto
{czhang, jacobsen}@eecg.utoronto.ca

Abstract[1]

In RE models such as KAOS, a complex system is directly represented in terms of its purposes, which makes its functionality much easier to understand and to reason comparing to code-level implementations. Part of the difficulty in maintaining a stronger correspondence between requirements and code is possibly due to the sufficient modularization capabilities of traditional architectures where many functionalities do not exist in distinct modular entities. This paper reports on an investigation of how and where some distinct design requirements lead to crosscutting concerns when decomposed into code in the KAOS model, using middleware as a case study. In doing so, aspect oriented design can be applied as early as possible in the development process. We match our past experience in aspect discovery at the code level with a detailed requirements modeling of the same architecture in KAOS. We observe that satisfying OR-decomposed subgoals in the KAOS model typically leads to tangled implementations, and agents responsible for multiple OR-decomposed goals should be implemented in the aspect oriented manner.

1. Introduction

As middleware gains popularity, the elegance of its transparent remote invocation (TRI) mechanism is getting increasingly difficult to maintain. An important reason is that the modularization capability of traditional languages and architectures is becoming insufficient in accommodating the magnitude of the diversification of middleware. The coherence of the TRI architecture gets very difficult to maintain and many middleware properties become less distinguishable and reusable in the architecture. In [9], we have shown that many middleware features do not exist in modular forms. This problem of "losing design in code" [2] is a result of the difficulty in maintaining modularity over rapid evolutions, extensions and expansions. This is of costly consequences in achieving adaptability, configurability, and maintenance. A new school of techniques aiming at alleviating this problem focuses on improving the modularization capabilities of traditional languages by offering new types of modules such as features [1], aspects [2], subjects [3], and compositional filters [13]. A common characteristic of these new techniques is that they are capable of breaking the modularity boundaries of conventional languages and of offering a finer modularity to express different views of the target system. However, these techniques are elaborated more as mechanisms than methodologies of building real systems. For complex systems such as middleware platforms, it is unclear and challenging how new modularization capabilities could benefit the implementation of the immensely complex design requirements.

In this article, we focus on one of well established emerging techniques, aspect oriented programming (AOP), and attempt to conjecture the "whys" and the "wheres" for the use of this technique in building middleware systems as part of the overall strategy of making middleware architecture look like the design. We embark on this task from the observation that, in the domain of requirements engineering, design requirements are naturally supported in distinctive entities, i.e., good modular forms, and, therefore, convenient to modify, extent, interact, and verify. For instance, KAOS [4] defines ***goals*** as the basic unit for representing the intents of a complex system. Goals are not concerned with how

[1] Technical Report, Middleware Systems Research Group, University of Toronto, July 2004

[2] Gregor Kiczales Website: http://www.cs.ubc.ca/~gregor

system is built but how it should function. It is desirable to have direct correspondences between the architecture and goals or, at the least, to understand how this correspondence disappears as goals are decomposed into code. We want to analyze if the phenomenon of "crosscutting" contributes to the loss of the distinctiveness of requirement entities, such as goals, in the final architecture, and whether the use of aspects can help maintain this correspondence if incorporated early in the requirements gathering phase.

This analysis is carried out as follows:

1. We first perform a consolidated modeling of middleware requirements. These requirements are extracted from a number of middleware design documents [middlewaredesign, taort, mdarts].
2. We then start from the code level and, through the technique of aspect mining, identify the architectural entities, existing in either conventional modules or aspects, which take part in supporting the goals.
3. Through the comparison between the goal decomposition and the actual code decomposition, we try to identify the connection patterns in the goal decomposition graph which give possible rises to aspects. This then helps us to link the existence of aspects to the properties and the interactions of requirements.

The consolidation of middleware design requirements is not a trivial task. The modeling of design requirements of the core middleware functionality yields more than two hundred concepts. However, a preliminary comparison reveals that supporting the ***OR*** relationships between goals usually cause violation of cohesion in modules and, hence, should be dealt with AOP. The **OR** relationship means that satisfying a particular design goal is equivalent to satisfying either of its many possible refinements. The ***Agent*** entity, which is responsible for executing the goals, is likely to contain aspect functionality in the code if it is responsible for multiple OR-related goals.

The rest of the article is presented as follows: Section 2 gives a brief introduction of middleware and the RE modeling language, KAOS, used in this analysis. Section 3 presents a partial KAOS modeling of the design requirements of the core middleware functionality. Section 4 discusses how the KAOS model is used in analyzing aspect orientation. Section 5 discusses the related work. Section 6 presents the conclusion.

2. Background

1. Middleware

The term ``middleware'' has various interpretations. In the context this discussion, we focus on middleware that facilitates the development of distributed systems in a heterogeneous networking environment. Middleware can be categorized depending on its identity coupling and temporal coupling characteristics among participants who request services (clients) and ones who provide computing services (servers). For example, message oriented middleware (MOM) systems, such as the MQSeries[3], allow communication among clients and servers through message queues. The communication style in MOM is temporally decoupled since the message passing is asynchronous. The sender and receiver can have different lifecycles. However, they are related via their identities because the sender needs to specify the identity of the receiver. Distributed object middleware systems, as well as the close kin, remote procedure call systems, typically provide a much stronger temporal coupling. Middleware technologies in this category include DCOM, CORBA, J2EE and Web Service. The common communication style in these systems not only requires prior knowledge of the receiver's identity but also require the synchrony of message passing. A third category, the publish/subscribe systems, sits at the opposite end of the spectrum. Neither does it require sender and receiver to know about each other, nor does it require the communication to be synchronous. An example of publish/subscribe systems is the Java Messaging Service (JMS)[4].

2. KAOS

The knowledge acquisition in automated specification framework (KAOS) [4] is a methodology for obtaining, explaining and justifying design requirements. The essence of KAOS consists of a set of entities representing concepts in deriving design requirements from goals. A goal is an objective of the composite system which can either be decomposed into sub-goals or achieved through some responsibility of an agent. A goal can be achieved in the equivalence of achieving all or either of its multiple subgoals. Therefore, a goal can be further decomposed or refined into more specific goals in either AND or OR relationships with each other. For example, if enabling transparent inter-program communication is a goal of middleware design, this goal can be achieved in either synchronous or asynchronous ways. Satisfying either subgoal is equivalent to satisfying the primary goal, i.e.,

[3] http://www-3.ibm.com/software/ts/mqseries/
[4] http://java.sun.com/products/jms/

the primary goal can be decomposed to two OR-related sub-goals as shown in Figure 1 using KAOS notations. Figure 2 shows an AND decomposition which specifies, to provide hard realtime guarantees, four more specific subgoals must be simultaneously satisfied. The realization a goal is primarily the responsibility of an agent. An agent is an autonomous unit capable of carrying out certain operations. For example, in Figure 3, three agents are responsible for achieving the goal of synchronous inter-program communication: a sender, a receiver, and a request dispatcher. In this article, we use Objectiver[5] as the modeling tool for KAOS diagrams.

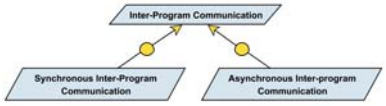

Figure 1. OR Decomposition of Goals.

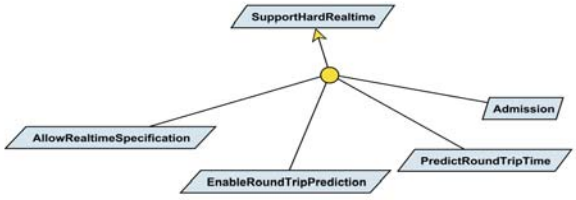

Figure 2. AND Decomposition of Goals

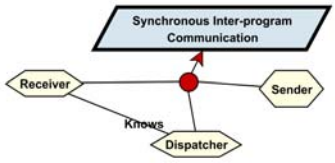

Figure 3. Goal Realization through Agents

3. Goal Decomposition of Middleware Requirements

[5] Objectiver http://www.objectiver.com

In this section, the KAOS model for a major portion of the middleware core design goals is presented. To limit the complexity, this model only concerns several middleware functionalities including basic inter-program communications, explicit and group identity coupling, hard realtime temporal coupling, and simple illities [10]. The formal definitions of KAOS concepts are skipped, as the intent is not to verify the consistency of the model.

3.1 High level goals of middleware

The core functionality of middleware needs to support its most basic functionality: the facilitation of inter-program communication. In addition, middleware provides certain identity and temporal properties. The low level API is hidden from the user application. Common illities also need to be supported. A trivial expectation for satisfying the overall goal is that user applications do not cancel the request after issuing it. The decomposition of goals is illustrated in Figure 4.

.

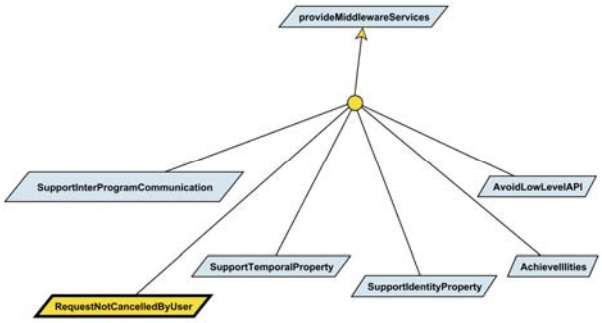

Figure 4. High Level Goals

The explanation for each goal is as follows:

AchieveIllities: Illities represent softgoals of the system because they are desired properties of the system but subject to subjective interpretations. Examples of Illities include configurability, programmability, traceability, and others.

AvoidLowLevelAPI: Hides the detailed knowledge about the platform such as network interface, encoding scheme, scheduling policies, etc.

RequestNotCancelledByUser: To satisfy the overall middleware service, the service must not be cancelled by the user.

SupportIdentityProperty: Provide properties about the identity of the communication party.

SupportInterProgramCommunication: Enable request-and-response based communication style between two communicating parties.

SupportTemporalProperty: Provides explicit properties about duration of the response.

3.2 Alternative Decompositions of Goals

Many middleware functional requirements are to be simultaneously satisfied by multiple alternatives. For instance, one of the basic functionality of modeled middleware is to support inter-program communications including both inter-network communications and inter-process communication. These two goals are alternative ways of satisfying the **SupportInterProgramCommunication** goal as illustrated in Figure 5.

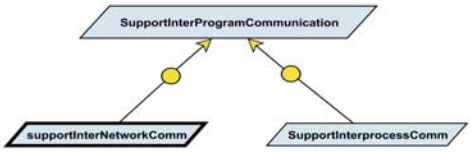

Figure 5. Decomposition of SupportInterProgram-Communication

Inter-network communication between two programs is one of the basic middleware functionalities. Its goal is to transparently bridge the gap between the application semantics and the networking semantics of underlying platform. The basic communication style is based on request and response pairs. Figure 6 shows the KAOS requirement model for this goal. Interprogram communication can also be supported through transparent inter-process communication. These two subgoals independently satisfy their parent goal which provides transparent inter-program communication.

In additional to inter-program communication, it is also necessary for middleware to support a wide range of temporal properties from hard realtime to complete decoupling in time. Therefore, the goal SupportTemporal-Properties can be satisfied by satisfying either of the following three sub-goals also illustrated in Figure 7:

StringentTemporal The response time of the request is constrained by certain user expectations.

NormalTemporal The response time of the request is limited by how long the shared system resources hold the client waiting.

NoTemporal The response time of the request is infinite. Typically the user either does not expect a response or response is a separate notification.

Among these subgoals, the **StringentTemporal** subgoal can be further OR-decomposed into two alternatives and one expectation explained as follows and illustrated in Figure 8:

SupportHardRealTime The response time of the request is limited by a user-specified duration.

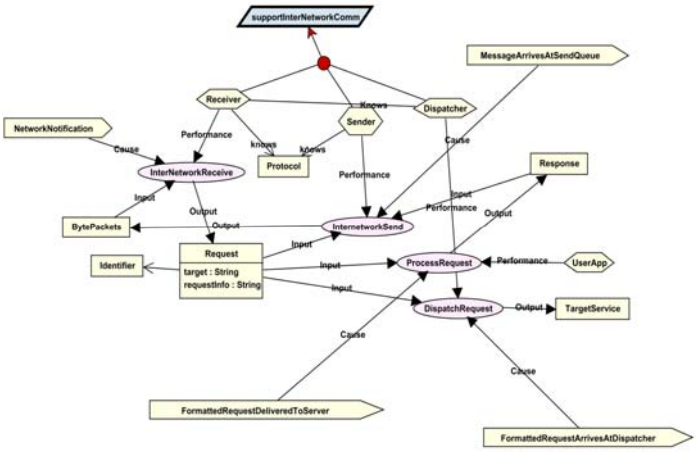

Figure 6. Decomposing the supportInterNetworkCom-munication

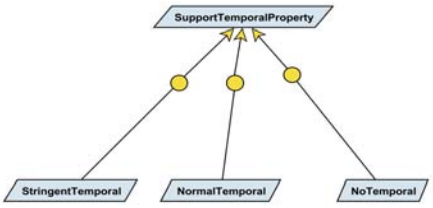

Figure 7: Sub-goal decomposition of the SupportTem-poralProperty goal

SupportSoftRealTime The response time of the request is limited by how long the shared system resources hold the client waiting.

PredictableRuntimeEnvironment The running platform of middleware needs to provide worst-case guarantees in providing CUP performance, memory access, IO bounds, and others.

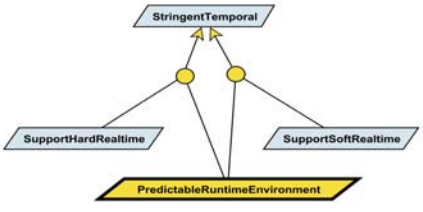

Figure 8: Decomposing the StringentTemporal goal

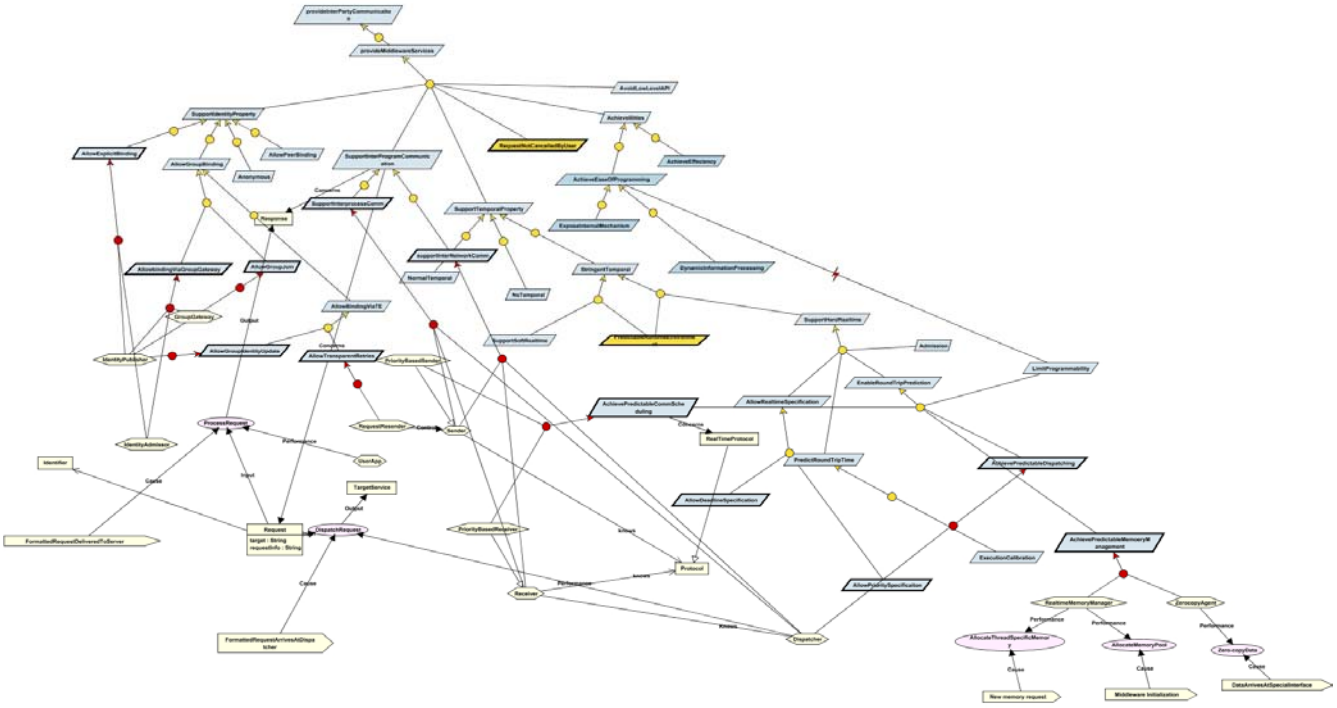

Figure 9: Consolidated KAOS Model

4. Aspect Detection in KAOS Model

The design requirements of middleware are already extraordinarily complex at the requirements level even in our preliminary attempt. Figure 9 shows a glimpse of the consolidated view of the KAOS models for achieving part of the enumerated goals. The entire model, although still simplistic and abbreviating, consists of over 200 concepts while implementing 6 out of 12 high level goals (we count the five essential middleware goals and their immediate sub-goals as high level goals). This complexity will almost certainly lead to unmanageability at the level of architecture and code. The rise of the aspect oriented programming provides new opportunities for enabling a more direct mapping of a system's functional intent to its code representation because, like other competing paradigms such as subject oriented programming, it provides constructions of multiple views in code.

4.1 Architectural and Code Level Aspects

Plain code inspection of middleware implementations reveals a similar problem as described earlier: a single architecture often supports multiple distinct functionalities which are alternative solutions to the same problem. Each functionality is often not cleanly modularized and difficult to identify, configure, and maintain. We categorize this phenomenon as concern crosscutting. Concern crosscutting is an inherent phenomenon in legacy middleware implementations. In [9], we have observed that over 50% of all classes in three different mature middleware implementations are crosscut by a certain concern, or, equivalently speaking, an aspect. Concern crosscutting occurs at two levels: architectural level where aspects exist in parts of class compositions, i.e., properties and methods, and code level where aspects exist in interactions among classes, i.e, in implementations of class methods.

Figure 10 illustrates an example of crosscutting at the architectural level. In this simple class hierarchy of a group of CORBA classes, the highlighted attributes and operations are part of the support for the dynamic composition of remote CORBA requests, whereas typical CORBA remote requests are composed statically. The functionality of dynamic request composition, although semantically independent from the static request composition, does not exist in separate modules but rather spreads across the main middleware class hierarchy.

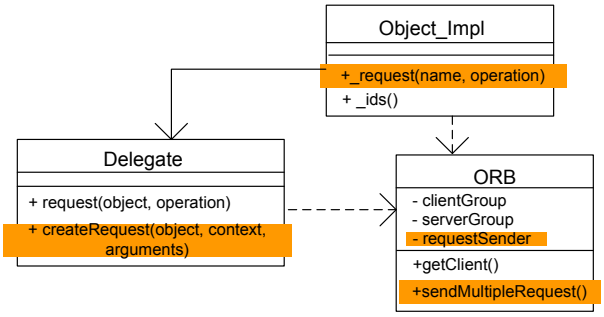

Figure 10. Architectural Level Crosscutting

Crosscutting also happens at the code level. Figure 11 shows crosscutting among multiple functionalities in an actual ORBacus code snippet. In this short piece of code, three concerns are present: portable interceptors (PI), which allow the normal call flow of ORB to be intercepted and altered; oneway, which supports the asynchronous communication style; and dynamic programming interface (DII), which supports dynamic request composition at the client side. This high degree of tangling undoubtedly makes all three functionality harder to understand, to change and to configure.

```
public Downcall
createPIDIIDowncall(String op, boolean resp,
       org.omg.CORBA.NVList args,
       org.omg.CORBA.NamedValue result,
       org.omg.CORBA.ExceptionList
        exceptions)throws FailureException
{
      ProfileInfoHolder profile =
            new ProfileInfoHolder();
      Client client =
      getClientProfilePair(profile);
      Assert._OB_assert(client != null);
      if(!policies_.interceptor)
          return new Downcall(orbInstance_, client,
          profile.value, policies_, op, resp);
          PIManager piManager =
                orbInstance_.getPIManager();
      if(piManager.haveClientInterceptors()){
           return new
           PIDIIDowncall(orbInstance_, client,
           profile.value, policies_, op, resp, IOR_,
           origIOR_, piManager, args,result,exceptions);
      }
      else{
            return new Downcall(orbInstance_, client,
            profile.value, policies_, op, resp);
      }
}
```

DII

Oneway

PI

Figure 11 Code Level Crosscutting

4.2 Mapping OR-Reduction to Aspects

The OR relationships in the KAOS model entails that there are alternative ways of achieving the same goal. This is an un-dictating view of how system functionality should be reasoned. If the system construction paradigms are as tolerant as KAOS, there is no use in the early criticism of the functional decomposition methodology for its limitation in the single dimension decomposition [8]. Aspect oriented programming shows more respect to the multiview problem and defines concern crosscutting as "In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they cross-cut each other." The referred "two properties" are easy of reuse and the efficiency of memory usage. They are two alternatives for implementing the same system, which appeals to different stakeholders: a system architect for the former and an end user for the latter. We think this view is very similar to the concept of the OR-reduction of goals in the KAOS model. In addition, a goal in KAOS is a collectively achieved functionality not by any single agent. This conceptually corresponds to the scattering problem which is a sufficient condition for the use of aspects.

The inspection of the aspect analysis in existing middleware architecture corresponds to this conjecture. The previous aspect analysis work [9] has shown that a number of features within the monolithic middleware core architecture are not simultaneously modularized in code and orthogonal to other middleware functionality in semantics. For instance, Figure 12 shows that the "InterProgramCommunication" goal can be satisfied by supporting communication between programs either over the network or across address spaces. In practice, it is actually common to implement both alternatives simultaneously to support different types of user applications. Our aspect analysis has shown that the support for inter-process communication is an aspect in the presence of supporting inter-network communications. Another example is in case of providing Illities. Figure 8 shows that one of the illities is the support for dynamic information processing. In the middleware architecture, this goal is usually achieved by providing dynamic types and associated facilities. This functionality is also shown as an aspect [9].

Therefore, we state the following:
A goal involved in an OR-relationship is likely to cause concern scattering and should be treated in the aspect oriented way.

4.3 Locating Aspects From Concepts to Architecture

From the previous conjecture, alternative subgoals give rise to the use of aspects. It is also interesting to know

how effectively the KAOS models can predict places in

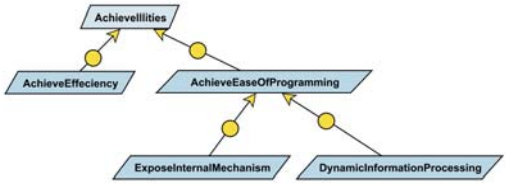

Figure 12: Goal decomposition of AchieveIllities

the architecture where AOP can be actually applied. We think the KAOS concept of Agent is similar to an architectural entity as it represents an autonomous computing unit which has clear responsibilities. This is at the least similar to classes that encapsulate certain functionalities and carry certain responsibilities. Agents that are simultaneously involved in OR relationships must handle different responsibilities. Therefore, it is hard to achieve coherent architecture to support these agents due to dramatic differences of their roles in supporting multiple alternatives of the same parent goal.

In the responsibility diagrams (Figure 13-14) generated by Objectiver, there are three agents: Dispatcher, Sender, and Receiver, all initially incepted to support inter-network communications. At the end of the modeling process, al of them support multiple goals (requirements). Each of the requirements is an alternative in achieving the six core goals in Figure 2. The inspection of the code shows that both Sender and Receiver have corresponding architectural entities to support inter-process communication. These entities incur concern scattering and can be implemented as aspects. The dispatcher's responsibility of supporting both inter-process and inter-networking also constitutes the breaking of cohesion and should be treated in an AOP way.

Therefore, we state the following:
Agents which are simultaneously responsible for multiple subgoals of certain goals should be decomposed in an aspect oriented manner.

5. Related Work

[11] describes an approach of using patterns of goal decomposition graph to discover aspects in early requirements. In this work, candidate aspects are identified if a goal is being intensively dependent upon by a number of other goals. A web application is then analyzed to verify whether goal aspects correspond to act

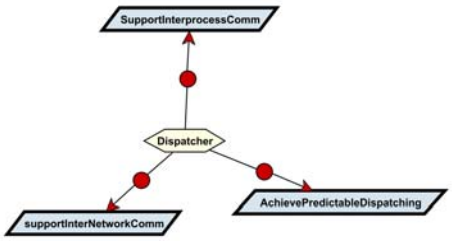

Figure 13: Responsibilities of Dispatcher

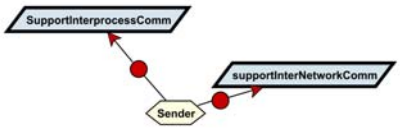

Figure 14: Responsibilities of Sender

actual aspects in code. Our approach is inspired by this work and complements it by matching requirement engineering models with our empirical aspect analysis accumulated from our past experience. We contribute another explanation of why and where aspects would exist deriving from the requirements model.

[12] presents a similar approach to [yijunre2004] by assessing the relationships between concerns and requirements in the aspect oriented requirements engineering framework (AORE). If a particular concern is present in multiple requirements, it is labeled as a candidate aspect.

6. Conclusion

This article makes a simplistic, less formal but important initiation of consolidating middleware design requirements using a requirements engineering modeling framework: KAOS. The modeling process starts from the very high level goals that are common for middleware to satisfy. Further decompositions and concretizations are pursued for a few core goals of middleware design including inter-program communications, explicit and group identity coupling, hard realtime support, as well as certain illities. These design goals and requirements are extracted from several research works and author's own experience. KAOS and Objectiver offer a convenient

platform for expressing different views of how middleware should function and still enjoying the ease of consolidating all the design requirements under one model. More importantly, the consolidated model offers some degree of interpretation of the use of new programming paradigms, such as aspect oriented programming. A comparison between the goal structures and the aspect mining results reveals that the goals in OR-relationships likely to give rise to aspects, and the architectural entities corresponding to agents handling multiple ORred subgoals are likely to be where AOP can help to improve their modularity.

As stated, this exercise has only leveraged part of the power in the KAOS modeling framework. None of the definitions in the presented model is formal, although KAOS provides a rich of logic tools to formally describe concepts. Formal definitions allow verification of inconsistencies and inference of new interactions in achieving middleware goals. And perhaps new insights regarding how aspects can be discovered or applied can come into light. This is undoubtedly interesting future exercises. The aspect analysis is conducted from a retrospective. To further validate the idea, the modeling should be conducted independent of the aspect knowledge, and the aspect analysis should involve new systems. It is interesting to see how such exercises would expend our vocabulary of aspect functionalities.

7. Refernces:

[1] Christian Prehofer. *Feature-oriented programming: A fresh look at objects*. Lecture Notes in Computer Science, 1241:419–??, 1997.

[2] G. Kiczales. *Aspect-oriented programming*. ACM Computing Surveys (CSUR), 28(4es), 1996.

[3] William Harrison and Harold Ossher. *Subject-oriented programming: a critique of pure objects*. In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 411–428. ACM Press, 1993.

[4] Anne Dardenne, Axel van Lamsweerde and Stephan Fickas, *Goal-directed Requirements Acquisition*, Science of Computer Programming, Vol. 20, North Holland, pp. 3-50.

[5] Anna Liu. *Gathering middleware requirements*. In The 15th International Conference on Information Networking, 2001.

[6] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, *The Design of the TAO Real-Time Object Request Broker*, Computer Communications, Elsivier Science, V21, No. 4, April, 1998

[7] Victor B. Lortz, Kang G. Shin, Jinho Kim. *MDARTS: A Multiprocessor Database Architecture for Hard Real-Time Systems*. IEEE Transactions on Knowledge and Data Engineering, p621-644

[8] G. D. Bergland. *Structured design methodologies.* In Proceedings of the no 15 design automation conference on Design automation, pages 475–493. IEEE Press, 1978.

[9] Charles Zhang and Hans-Arno Jacobsen. *Re-factoring middleware with aspects*. IEEE Transactions on Parallel and Distributed Systems, 2003.

[10] Robert Filman. *Achieving ilities.* URL: http://ic.arc.nasa.gov/~filman/text/oif/wcsa-achieving-ilities.pdf.

[11] Yijun Yu, Julio Cesar Sampaio do Prado Leite, John Mylopoulos. *From Goals to Aspects: Discovering Aspects from Requirements Goal Models*. In Proceedings of the 12th IEEE International Requirements Engineering Conference. Kyoto, Japan, 2004

[12] A. Rashid, A. Moreira, and J. Araujo (2003) *Modularisation and Composition of Aspectual Requirements* . 2nd International Conference on Aspect-Oriented Software Development. ACM. Pages 11-20.

[13] L. Bergmans and M. Aksit, *Composing Crosscutting Concerns Using Composition Filters*, Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.