Two Party Aspect Agreement using a COTS Solver

Eric Wohlstadter
University of British Columbia
wohlstad@cs.ubc.ca

Stefan Tai
Thomas Mikalsen
Isabelle Rouvellou
IBM Watson Research Center
stai,tommi,rouvellou@us.ibm.com

Prem Devanbu
University of California, Davis
devanbu@cs.ucdavis.edu

ABSTRACT

A number of researchers have proposed an aspect-oriented approach for integrating concerns with component based applications. With this approach, components only implement a functional interface; aspects such as security are left unresolved until deployment time. In this paper we present the latest version of our declarative language, GlueQoS, used to specify aspect deployment policies. Our work is focused on automating the process of configuring cooperating remote aspects using a client-server handshake. During the handshake the two parties agree on aspect configuration by using mixed integer programming. A security example is presented as well as initial performance observations.

1. INTRODUCTION

Extending component interfaces directly with information about non-functional concerns limits the reusability of an interface. Each component implementing the interface must be prepared to handle these concerns appropriately. Furthermore, it also limits customizability, for example, the ability of local security officers to tailor policy enforcement code to suit their settings.

To address this shortcoming, a number of researchers have proposed an aspect-oriented approach for integrating concerns with component based applications [18, 6, 8, 11, 14, 19]. With this approach, components only implement a functional interface; aspects such as security are left unresolved until deployment time. A pointcut specification, written by a deployment expert, can be used to weave aspects and the original components. As presented, the approach does not consider the issue of matching client-side aspects to the deployment on the server. This is important when client and server-side aspects must cooperate [19, 13], security or fault-tolerance aspects being prime examples.

This inflexibility limits the use of the approach in new, emerging application areas such as service-oriented architectures (SOA). In a SOA, client applications and server applications from the same product family are not always consistently deployed across a wide-area. This may yield variation in the features[1] of client and server software. We propose to provide *dynamic and symmetric reconciliation* between the (potentially different) features (implemented as aspects) of two communicating processes. However, different aspects can interact in various ways, and this complicates reconciliation.

We use the term *interaction* [21] to reflect how aspect combinations affect each aspect's ability to function as it would separately. Interactions can be complex, subtle, and very difficult to identify. Finding such interactions is outside the scope of this paper. In addition, aspect configuration is a matter of deployment policy and can vary.

In this paper we present the latest version of our declarative language, GlueQoS, used to specify aspect deployment *policies*. A middleware-based resolution mechanism uses these specifications to dynamically find a satisfying set of aspects that allow a client and server to inter-operate. We have presented a previous version of this work in [20]. The motivation and example we present are an update of our previous work. However, this paper additionally describes a completely revised language design and implementation.

The remainder of this paper is organized as follows: Section 2 presents the GlueQoS language, Section 3 presents an example, Section 4 presents the COTS solver we used, Section 5 presents implementation, Section 6 reviews related work and finally we conclude in Section 7.

2. POLICY LANGUAGE

Policies are specified in the GlueQoS policy language. The language provides a set of built in operators to specify acceptable aspect *configurations*, as well as the ability to extend the system with functions to measure operating conditions (such as load, available energy, and bandwidth). A configuration includes which aspects should be used as well as what their operating parameters should be.

Each client to server session is associated with a set of *adaptlets* [19]. An adaptlet conceptually encapsulates a pair of client and server aspects. This includes the provided and required interfaces of client and server aspects. This scoping of aspects can be viewed in analogy to the "`aspect per`" scoping of AspectJ but based on a client/server negotiated session. The adaptlet abstraction serves to properly type the "connection" between cooperating aspect instances. Our work is focused on automating the process of configuring these instances using a client-server handshake. During the handshake the two parties agree on a set of adaptlets to use, as well as the values of any parameters they expose. In this section we detail the constructs in GlueQoS used

[1] We use the term *feature* to denote an artifact of software requirements and *aspect* to denote an artifact of software implementation.

to automate this handshake including support for boolean constraints, linear constraints, and run-time monitoring.

2.1 Boolean Constraints

Aspect agreement can range from a very simple problem (e.g., when all aspects are orthogonal (non-interacting)), to a very hard problem (e.g., when aspect interactions are arbitrary). Since aspect-oriented middleware systems are not widely deployed, we draw on work in other areas [5, 21, 1] in order to hypothesize what an ideal framework requires. Therefore, our description is highly expository in nature rather than purely prescriptive.

Each host (client or server) will need to include in their policy a set of statements to impose some requirements on the adaptlets used in a session. If an adaptlet is used in a session, we say the adaptlet's *status* is *on* (true) for that session; otherwise, the adaptlet's status is *off* (false).

Due to the nature of interactions, sometimes one adaptlet may be dependent on another adaptlet. Also, adaptlets might conflict: viz, they cannot be used together. Finally, since hosts do not have *a priori* knowledge of which adaptlets are supported by a peer, it is useful to provide choices amongst a number of adaptlets, in order to meet some requirement. Here we present the encoding of these constraints in GlueQoS, whose syntax follows from boolean logic:

- $\mathcal{Dependency}$: The deployment of an aspect, A, depends on the deployment of another aspect, B. This can be encoded as an implication, `(A implies B)`.
- $\mathcal{Conflict}$: Two aspects conflict if their combination has a negative effect on the behavior of the entire application. The deployment of one aspect should exclude the deployment of the other. The decision that an effect is negative is application dependent but may include effects such as introducing deadlock, putting data in inconsistent states, or degrading performance. $\mathcal{Conflict}$ is encoded by, `not(A and B)`.
- $\mathcal{Choice}$: Either aspect or both can be chosen to meet some requirement. This is encoded as `(A or B)`.

Based on these examples, it is straightforward to encode other requirements such as those stemming from three-way interactions. Now, given that in a SOA agreement may need to be performed at run-time, one can see it is desirable to provide for efficient computation of aspect status based on host policies. However, even deciding status for policies of $\mathcal{Choice}$ and $\mathcal{Conflict}$ is not easily computable (i.e., not tractable) in general. The important question that we address in the remainder of this section is, "Is there a reasonable restriction of arbitrary boolean constraints that is tractable?". We believe the answer is no, so we have opted not to impose any restrictions on boolean constraints.

One way to answer this question would be list all known tractable restrictions [7] and argue why each one is not reasonable. We believe this is possible, however the list in lengthy, and contains relatively few restrictions actually used in any practical setting. Noteworthy examples from the list are 2-SAT and Horn-SAT. Instead, we describe a minimal restriction of boolean constraints which is still intractable, yet we argue is reasonable. By minimal we mean that it is difficult to imagine how one could usefully restrict it further.

We start with 2-SAT. It requires that dependencies be of the form `(A implies B)`. For instance, `((A and B) implies C)` is not allowed. This seems reasonable since software dependencies are usually cast in terms of binary relationships. Now, conflicts are required to be of the form, `not(A and B)`. For instance, `not(A and B and C)`, is not allowed. This seems reasonable since it is difficult to imagine a case where two software packages don't conflict *until a third* is present. Finally, choices are required to be of the form `(A or B)`. For instance, `(A or B or C)` is not allowed. Our argument rests on the fact that we believe this restriction seems unreasonable. So, relaxing 2-SAT slightly to allow multiple choice gives us our restriction which is reasonable yet intractable. An equivalence reduction between 3SAT (a classical intractable problem) and this relaxation of 2-SAT is straightforward. This is by no means a formal proof, but we hope it gives the reader insight into our language design.

Since no shortcuts seem likely, we appeal to brute force provided by a COTS constraint solver. In terms of language design, we have traded off scalability for expressiveness. Arbitrary interactions are supported but can only be reasoned over efficiently if the total number of aspects on a given system is limited. From our initial experiments we believe support for up to 50 aspects should be easily manageable.

2.2 Linear Constraints

Now that we have addressed the motivation and interpretation for policies regarding acceptable adaptlet status, we turn to the matter of adaptlet parameter constraints.

Every adaptlet may need to be configured according to a set of parameters. This is analogous to Component Oriented Programming [16]. For example, in the Java Beans component model every bean may expose a set of attributes for deployment time configuration. However, in our scenario we must allow for joint agreement, between client and server, of the session-time parameters.

For this purpose, we allow the representation of *linear constraints* [15, 4] over adaptlet parameters. For example, a linear constraint could be used with two adaptlets implementing a service-level agreement,

$$-2.0 * PayFeature.price + 1.0 * QoS.guarantee = -100.0$$

This constraint sets the price of a connection at fifty dollars plus half the amount of bandwidth reservation. Graphically, this allows clients and servers to negotiate a choice of the two aspect parameters (referenced as fields of the aspects) anywhere along the line defined by the equation.

In contrast to systems of non-linear constraints, linear systems are decidable and tractable. Thus far we have not explored support for non-linear constraints. Modern solvers are usually based on the *Simplex* [4] algorithm due to Dantzig.

2.3 Run-time Policy Adaptation

Recall that hosts execute in an environment that is continuously changing; they might need to be configured according to a dynamic deployment context. Rather than force deployment experts to constantly update policies manually, our policy language includes constructs to reflect these environmental changes. The constructs are of two types: user-defined value functions and user-defined predicate functions.

The values of coefficients or constants in linear constraints can be input through user-defined value functions. Evaluation of these functions occurs periodically throughout the execution of client and server applications. Before policy resolution occurs, a "snapshot" of the client and server policies is taken to reflect their current states. A similar approach is used in QuO [11], however, not in the context of aspect agreement. For example, we can update the example given as,

$$-2.0 * PayFeature.price + 1.0 * QoS.guarantee = \{cpuLoad() * 100.0 - 100.0\}$$

Graphically, this allows the expression of a line which is shifted vertically based on the current value of the user-defined function `cpuLoad`.

Likewise, requirement of a particular adaptlet in an acceptable configuration may also depend on the state of the execution environment. A security feature, for example, may only be required for certain types of network connections e.g.,
Password and (Encryption when { linkType("mobile") }).

Here, the required configuration varies between using the Password feature alone and using both the Password and Encryption feature. This variation is based on evaluation of the `linkType` user-defined predicate using our `when` keyword.

We have shown that the acceptable feature configurations may vary dynamically. The actual policies expressed depend on the moment when resolution occurs. We have provided two constructs in our language to express this variation.

2.4 Special Functions

In addition to the language elements we have laid out so far, two special types functions are supported. These are the `Supports` and `Preference` functions.

Our client/server scenario must account for the fact that client and server policies are written in isolation. Therefore, the sets of adaptlets mentioned in each policy might not be the same. We chose the semantics that any adaptlet not mentioned in both policies would be assumed to have its status forced to off. This default assumption can be suppressed by adding a `Supports` clause. For example, the clause `Supports(A)` can be interpreted as adding the tautology `(A or not(A))` to the set of constraints.

Assuming the client is given some $\mathcal{C}hoice$ (as in Section 2.1) between adaptlets to meet a particular requirement, the `Preference` function provides a way to instruct the COTS solver which adaptlet to choose. Optimization methods based on linear programming allow for computation of a solution which maximizes some utility function over the constraint variables. Leveraging this utility function we can support preferences over the configuration of aspects from any possible configurations. Currently the `Preference` function is only available to clients because the asymmetry in our handshake protocol (see Section 5.1) cannot provide proof that any server `Preference` functions would be respected. We plan to revisit this restriction in future work.

In the following section we demonstrate a possible usage of the language elements laid out in this section.

3. SECURITY EXAMPLE

Consider deployment of a client/server application in an environment where two security adaptlets are required. The first is *authentication*. The server must protect certain services from unauthorized access; so client requests must be preceded or accompanied by an authentication step involving the presentation of credentials in order to gain group membership for those services. Credentials can be based on a password, or on public-key signatures. In this case, an aspect on the server side is responsible for checking credentials, and the corresponding aspect on the client-side is required to present the appropriate credentials.

The second adaptlet, the *client-puzzle protocol* (CPP) [5], defends against denial-of-service (DoS) attacks. A DoS attack occurs when a malicious client (or set of malicious clients) overloads a service with requests, hindering timely response to legitimate clients. Certain components of the server may be prone to DoS attack because of the amount of computation required by the components. CPP protects a component by intercepting client requests and refusing service until the client provides a solution to a small mathematical problem.

CPP and Authentication interact in interesting ways. For example, suppose the server's only requirement is to prevent DoS attacks. If we trust authenticated clients not to mount DoS attacks, then the authentication and client-puzzle adaptlets are equivalent and one can be substituted for the other; it would be redundant to use both. However, sometimes authentication may not imply a decreased risk of DoS attacks, so these adaptlets would be viewed as orthogonal. In other situations, we may require both authentication and DoS defense.

Client-side preferences must also be considered when selecting the adaptlets that govern a client-server interaction. A client may consider CPP and Authentication to be equivalent, and express a policy that it can use either. A client with a performance requirement, however, would naturally prefer to employ authentication to avoid computing puzzle solutions. A client who values its privacy would prefer to expend CPU cycles in order to not have to reveal their identity; this client may prefer to use CPP rather than provide identity-revealing credentials.

Figure 1 is a realization of the security policy as expressed in GlueQoS. The first policy is shown for the server.

Each line (1, 2, and 3) represents a different configuration constraint. The first is an implication between the status

Server:
```
(1) (Authentication implies (CPP.size = { cpuLoad()*8 }));
(2) (CPP when { cpuLoad() > .5 });
(3) (Authentication or (CPP.size = { cpuLoad()*16 }));
```

Client1:
```
(4) Authentication;
```

Client2:
```
(5) Supports(CPP,Authentication);
(6) Preference(not(Authentication),Authentication);
(7) (CPP.size <= 4);
```

Figure 1: Security Example

of the Authentication adaptlet and a constraint on the size parameter of the CPP adaptlet. It states that with Authentication, the size of puzzles varies linearly from 0 to 8 depending on CPU load. Another constraint (line 2) uses a predicate `(cpuLoad() > .5)` to determine whether CPP is required. When CPU load is less than .5, the server allows Authentication to be used without the CPP; otherwise just CPP, with the largest puzzle size, can be used. This shows how run-time conditions can dynamically adapt the acceptable adaptlet combinations expressed by hosts.

The first client policy is shown on line 4. This client will only use the Authentication adaptlet (perhaps because of software availability, or because it is too performance-limited for CPP). Therefore, this client can only create a session with the server when the server's load is less than 0.5.

The second client policy (lines 5-7) uses parameter constraints to choose between two adaptlet combinations. Note that the `Preference` semantics in our language denotes a preference for the first alternative. Consider a situation where this client wishes to maintain its anonymity by not using the Authentication feature. However, it also has a performance requirement that takes precedence. Perhaps the client is on a mobile device with low computing power. Line 6 expresses the client's preference to maintain anonymity. However, in order to keep performance at a certain threshold the client will also use Authentication if it will keep the puzzle size low. By comparing to the sample server's policy (lines 1 and 3 in particular): if this client contacts the server when the server's CPU load is 25 percent or lower the client can maintain its anonymity by using CPP only (from line 3 and 7, $16 * .25 <= 4$). However, if it contacts the server and the server's CPU load is between 25 percent and 50 percent it will agree to reveal its identity to maintain higher performance (from line 1 and 7, $8 * .5 <= 4$). When the server's load passes 50 percent the client will be not be able to find a solution to the constraints imposed by the policy.

4. MIXED INTEGER PROGRAMMING POLICY MATCHING

In Section 2, we described our policy language for expressing adaptlet configurations. We have made some informed design decisions and arrived at an implementation based on mixed integer programming. Pragmatically, the best choice for these decisions would be based on best practices observed over a number of years. Certainly this is difficult as aspect-oriented middleware is not widely deployed in practice.

Mixed Integer Programming has been used widely in the area of Operations Research [4] for decades. Here we apply this technique for automating the configuration of aspect-oriented software in a client/server setting.

Mixed Integer Programming extends the theory of linear programming. In a mixed integer program (MIP) a subset of variables can be constrained to integer values. Hence, the "mixed" denotation refers to a mix of real and integer variables. A popular strategy for solving a MIP is based on the Branch-and-Bound [15] algorithm. In this paper we view the MIP algorithm as a COTS component that is utilized for the purpose of resolving policies. This is achieved by modeling adaptlet status as 0/1 integers and adaptlet parameters as integer or real variables.

5. IMPLEMENTATION

Our prototype implementation builds on the existing DADO dynamic AOP middleware [19] and the Lindo API [10] for mixed integer programming. This involves attaching policies to applications, maintaining a run-time representation of policies, and finally deploying the properly parameterized resolved aspects.

A deployment expert considers local requirements and aspect interactions to design a policy. The policies are associated with CORBA interface types, before an application is executed. Our implementation currently does not support policies on a per-method basis; a single policy can be assigned to each interface type. At application load-time the GlueQoS middleware builds a data-structure representing these policies. Now we describe the overall set-up as in Figure 2.

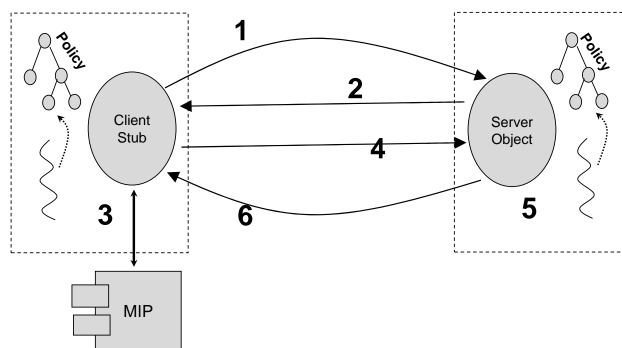

Figure 2: The overall flow of the GlueQoS runtime, including client stub, server stub, and the Mixed Integer Programming (MIP) runtime component

The figure represents the client and server runtime using our GlueQoS middleware, separated on the left and right sides respectively. The dotted-line boxes represent the boundary between middleware related functionality and the black-box MIP component.

Inside the dotted lines are three pieces. First, the large circles represent the client stub and server object to which the session based aspect agreement applies. Second, the tree of nodes represents the policy data-structure. Third, a separate thread, shown as the curved line, is responsible for updating this data-structure based on the values retrieved from user-defined functions. Now we focus on the interaction defined by the numbered flow of the diagram.

5.1 Client/Server Interaction

The GlueQoS middleware at each end of an interaction determines adaptlet configuration for each application session. These aspects and their operating parameters remain fixed for the lifetime of the session. In the future, we plan to investigate support for continuous adaptation of operating parameters.

When a client locates a server, it sends a *policy request* (1) to the server object to initiate a session. Policy requests are implemented as a CORBA operation that is transparently introduced to all IDL interfaces. This is performed by a compiler that is part of our DADO toolset.

The server creates a session for the client in the form of a *cookie*. Now, the server serializes the policy data-structure, associates it with the newly created session and returns the serialization to the client. Note that at this point, all runtime adaptation functions have been evaluated out of the policy, creating a static policy based on the current environment. This means that in the example, the server does not have to reveal the fact that policy is based on current cpu load.

Now, the client must match its own policy with the server and choose an adaptlet configuration acceptable to both. First, client and server data-structures are merged. Now, a client matches policies by carrying out the mixed integer program resolution. The merged data-structure and a vector representing the client's preferences are passed to the Lindo API. It will return a satisfying assignment for all variables or signal unsatisfiability (3). These results are used to control the execution of aspects. In the case of unsatisfiability, an exception is thrown to the application to signal incompatible policies.

The configuration chosen by Lindo is used in the creation of aspects which implement the adaptlet collaboration. These aspects are instantiated using the Java Reflection API. The parameter values chosen are passed to the aspect's constructor. The signature of the constructor and parameters for each aspect are part of the adaptlet type. The values can then be used by the advice to configure aspect execution. In this way the resolved aspects are activated and configured according to the policies of both client and server.

The setup chosen by the client is then serialized and sent to the server (4). This message is piggybacked on a subsequent application request to the server. The server must verify that the setup chosen by the client actually satisfies its own policy. This requires only a simple linear time check of constraint satisfiability (5). The values for the variables are plugged into the policy which was associated with the client's session. If verification is successful, the server can discard the associated policy and create aspects in the manner described for the client side. On subsequent requests, the cookie from the client is used to execute aspects and advice on a per-client basis. If verification is unsuccessful an exception is thrown back to the client (6).

5.2 GlueQoS Prototype

Our GlueQoS implementation has been tested on the example presented in this paper and a previous version on an example in a related paper [17].

To understand some of the performance impact induced by the GlueQoS software we measured the overhead of the GlueQoS handshake phase (Figure 2, steps 1 - 5) on the example of Figure 1 with the second client policy. Our experiments showed that the overhead is dominated by the communication costs of steps 1 and 2 in Figure 2.

An important detail missing from this experiment is the fact that only a single example policy was used. Since the policy solver of step 3 grows exponentially with the number of integer variables required in the policy encoding, it will be important to repeat the experiments for a range of policy sizes. We could draw from the approach described in [12]. This work shows how to generate random 3-SAT instances of a desired size and difficulty (i.e., time required to solve the instance). In the future it may be possible to extend that work for generating random policies of varying difficulty that can be used for further experiments.

6. RELATED WORK

Aspect-Oriented middleware is motivated by the need to provide flexible customization with a simplified deployment process, combining the benefits of reflective middleware with container based deployment.

Recently, the open-source JBoss [9] application server announced aspect-oriented deployment of container services using the Javassist [2] byte code editing toolkit. A similar approach is used in the Java Aspect Components (JAC) framework [14] that also utilizes load-time byte code weaving (using BCEL [3]) in Java. New services can be constructed by implementing aspect-specific interceptors. Deployment takes place using the notion of pointcuts from the AspectJ language. In JAC, aspects can be un-deployed/re-deployed dynamically using a standardized API.

The Quality of Objects (QuO) [11] project aims to provide consistent availability and performance guarantees for distributed objects in the face of limited or unreliable computation and network resources. QuO defines an abstraction known as the operating region for processes (client or servers) cooperating in a distributed object environment. Changes in perceived run-time conditions move a process into different operating regions. Advice that is bound to a particular operating region or region transition is the main vehicle by which adaptation is achieved.

The aspect-oriented middleware presented in this section achieve both flexible customization and simplified deployment. This is made possible by a clear separation between adaptation programming and deployment. Deployment is facilitated by pointcut based descriptions which map adap-

tation behavior to application events. Our work on GlueQoS could be used to simplify run-time deployment of cooperating aspects in client-server applications. Previously, we have presented the notion of an adaptlet collaboration which serves to properly type client and server aspect roles.

7. CONCLUSION

GlueQoS is middleware software to support dynamic adjustment of aspects between clients and servers. Configuration preferences are specified in the GlueQoS policy language. These policies are exchanged at binding time between systems interacting in an ad-hoc setting. The polices are then matched up, and resolved by the middleware. The resolved aspects are then deployed and executed. GlueQoS has been implemented in the context of adaptlets.

8. REFERENCES

[1] BEA, IBM, Microsoft, and SAP AG. Web services policy framework (WS-Policy), May 2003.

[2] S. Chiba. Load-time structural reflection in Java. In *Proc. of the European Conference on Object-Oriented Programming*, pages 313–336, 2000.

[3] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universit at Berlin, Institut fur Informatik, 31 pages, 2001.

[4] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1962.

[5] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. of the USENIX Security Symposium,* 9 pages, 2001.

[6] F. Duclos, J. Estublier, and P. Morat. Describing and using non-functional aspects in component based applications. In *Proc. of the International Conference on Aspect-Oriented Software Development*, pages 65–75, 2002.

[7] J. Franco and A. van Gelder. A Perspective on Certain Polynomial Time Solvable Classes of Satisfiability. In *Abstracts of the International Symposium on Mathematical Programming*, 1997.

[8] F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. Aspectix: a quality-aware, object-based middleware architecture. In *Proc. of the 3rd IFIP Int. Conf. on Distrib. Appl. and Interoperable Sys.*, 2001.

[9] JBoss. <http://www.jboss.org>. 4.0 edition.

[10] Lindo API. <http://www.lindo.com/>. 2.0 edition.

[11] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. QuO Aspect languages and their runtime integration. In *Proc. of the Workshop on Languages, Compilers and Runtime Systems for Scalable Components,* 16 pages, 1998.

[12] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Prof. of the Conference on Artificial Intelligence*, pages 459–465, 1992.

[13] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed aop. In *Proc. of the International Conference on Aspect Oriented Software Development*, 2004.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible framework for AOP in Java. In *Proc. of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection),* 24 pages, 2001.

[15] M. Simonnard. *Linear Programming.* Prentice Hall, 1966.

[16] C. Szyperski. *Component Software – Beyond Object Oriented Programming.* Addison Wesley, 1997.

[17] Stefan Tai, Thomas Mikalsen, Eric Wohlstadter, Nirmit Desai, and Isabelle Rouvellou. Transaction policies for service-oriented computing. *Data and Knowledge Engineering Journal: Special Issue on Contract-based Coordination and Collaboration*, 51:59–79, 2004.

[18] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B.N. Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the International Conference on Software Engineering*, pages 233–242, 2001.

[19] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Dado: Enhancing middleware to support cross-cutting features in distributed, heterogeneous systems. In *Proc. of the International Conference on Software Engineering*, pages 174–186, 2003.

[20] Eric Wohlstadter, Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, and Premkumar Devanbu. Glueqos: Middleware to sweeten quality of service policy conflicts. In *Proc. of the International Conference on Software Engineering*, 2004.

[21] P. Zave. An experiment in feature engineering. *Programming Methodology*, pages 353–377, 2003.