Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 61 (2006) 58–71
Science of Computer Programming
www.elsevier.com/locate/scico

Modeling and performance analysis for security aspects

Lirong Dai[a,*], Kendra Cooper[b]

[a] *Department of Computer Science and Software Engineering, Seattle University, 901 12th Avenue, P.O. Box 222000, Seattle, WA 98122-1090, USA*

[b] *Department of Computer Science, The University of Texas at Dallas, Mail Station ECS 31, P.O. Box 830688, TX 75083-0688, USA*

Received 19 March 2005; received in revised form 15 September 2005; accepted 14 November 2005
Available online 29 March 2006

Abstract

The problem of effectively designing and analyzing software to realize non-functional requirements is an important research topic. The significant benefits of such work include detecting and removing defects earlier, reducing development time and cost while improving the system's quality. The Formal Design Analysis Framework (FDAF) is an aspect-oriented approach that supports the design and analysis of multiple non-functional properties for distributed, real-time systems. In this paper, a security attribute, data origin authentication, is defined as a reusable aspect based on its security pattern definition. The FDAF provides a UML extension to weave the security aspect into a UML architecture design. This is accomplished by abstracting Aspect-Oriented Programming concepts join point and advice up to the design level. The FDAF supports the automated translation of a UML architecture design into Rapide [D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering 21 (4) (1995) 336–354], a formal architecture description language, allowing the simulation of a system's response time. Thus, the response time of a design with and without the security aspect can be respectively analyzed, and the performance cost of this aspect can be predicted. One of the translation algorithms, which have been implemented in the FDAF tool support, and its proof are presented. The FDAF approach is illustrated using a Domain Name System example.

1. Introduction

A fundamental goal of software development is to deliver high quality products that are correct, consistent, and complete. This goal has driven researchers to investigate methodologies that support the effective design and analysis of non-functional properties for systems at the software architecture design level. The benefits of such work are detecting and removing defects earlier, which in turn reduce development time and cost.

A software architecture describes the high-level structure of a software system, where the structure focuses on the specification of a collection of components and their behavior, connections among components, and constraints of the system [3]. A software architecture must support the functionality required of the system, and also realize its non-functional requirements, such as performance, security, accuracy, etc. [13]. Non-functional requirements have a very important role in the software system development. When facing more than one competing designs, architects can use non-functional requirements as selection criteria, leading to rational decisions.

* Corresponding author.
E-mail addresses: daia@seattleu.edu (L. Dai), kcooper@utdallas.edu (K. Cooper).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.11.006

However, in an architecture, a system's non-functional properties tend to be global, and their crosscutting characteristic makes it relatively difficult to encapsulate non-functional properties within Object-Oriented model elements, such as classes and operations. Two main problems of crosscutting non-functional properties are lower cohesion (i.e., their design or implementation is scattered throughout the whole system) and higher coupling (i.e., one model element may be comprised of more than one concern, which causes numerous connections between different concerns). The new design paradigm, Aspect-Oriented Software Development (AOSD) [8], has been proposed to enable the modularization of such crosscutting concerns. AOSD adapts the principles of aspect-oriented programming to the design phase. Aspect-oriented programming techniques provide linguistic mechanisms for the separate expression of concerns, along with implementation technologies for weaving these separate concerns into working systems. In AOSD, a system's tangling concerns or pervasive properties are encapsulated in a model element called *aspect*, and subsequently a *weaving* process is employed to compose core functionality model elements with these aspects, thereby generating an architecture design. AOSD allow each aspect model to be constructed and evolved relatively independently from other aspect models, thus it dramatically reduces the complexity of understanding, change, and analysis of design models while promotes software reusability.

Software architecture can be specified using informal notations (e.g., English), semi-formal notations (e.g., the Unified Modeling Language (UML) [25]), or formal notations (e.g., Architecture Description Languages (ADL)). UML has become a de facto design notation standard in the software engineering community; its graphical presentation is considered relatively easy for humans to understand and formulate. However, although much of the syntax of UML has been defined, the semantics of UML are specified informally in English, resulting in a lack of tool support to formally reason about the systems' specification. Formal notations have both formally defined syntax and semantics, and their associated tools support the rigorous specification, design, and verification of software systems. However, formal specifications are considered by many to be more difficult to understand and modify.

Expanding on the interesting results accomplished by other research groups, an aspect-oriented architectural framework, Formal Design Analysis Framework, has been proposed to support the design and analysis of multiple non-functional properties for distributed, real-time systems. In the FDAF, non-functional properties are defined as reusable aspects in the repository. The support for the design of two performance aspects, the response time aspect and the resource utilization aspect, and their analysis in ADL Rapide [17] and Armani [22] has been presented in previous work [7]. Here, the focus is on extending the repository to include security aspects; the aspect definitions are adapted from their corresponding security patterns [26]. A UML extension is proposed to represent aspect-oriented programming concepts join point and advice, and support the weaving of reusable aspects into a UML design. This extension can help architects to generate aspect classes that are supported by AspectJ [15]. Once a security aspect is woven into a design, the performance impact of the security aspect can also be included. Hence, the design developed using the FDAF more accurately reflects the fact that aspects, such as a security feature, can also have aspects, such as performance, associated with them. The FDAF tool supports the automated translation from an aspect-oriented UML design into Rapide formal specification. Thus, the performance cost of adding security aspects into a design can be predicted using Rapide's architecture simulation tool. This paper uses the Domain Name System (DNS) [21] and a security aspect, data origin authentication aspect, to illustrate the FDAF approach. The aspect definition of the data origin authentication, weaving the aspect into the DNS architecture design, and the aspect analysis in Rapide are presented. The translating algorithm from an extended UML class diagram into Rapide, its proof, the FDAF tool support, and the response time performance cost analysis result of this security aspect for the DNS are also presented.

The remainder of the paper is organized as follows. Section 2 presents the related work. Section 3 presents an overview of the FDAF. Definitions of aspect-oriented programming concepts join point and advice at the design level are introduced in Section 4. Section 5 presents the algorithm of translating an extended UML class diagram into Rapide and its proof. The DNS example is used to illustrate the approach in Section 6, and a discussion of the FDAF is presented in Section 7. Finally, conclusions and future work are presented in Section 8.

2. Related work

The FDAF draws upon aspect-oriented approaches to support the use of defined, reusable aspects, and proposes an extension and translation of UML into ADLs. A survey of work related to these areas is provided in this section.

```
1: public aspect MyAspect {
2:   public pointcut sayMethodCall(): call (public void TestClass.sayHello());
3:   public pointcut sayMethodCallArg(String str): call
                  (public void TestClass.sayAnyThing(String)) && args (str);
7:   after(): sayMethodCall() {
8:   System.out.println("\n TestClass." +
       thisJoinPointStaticPart.getSignature().getName() + "end...");}
9:   before(String str): sayMethodCallArg(str) {
10:     if (str.length() < 3) {
11:     System.out.println("Error: I can't say words less than 3 characters");
12:     return;}
13:  } }
```

Fig. 1. AspectJ example.

2.1. Aspect-oriented approaches

Approaches using AOSD in software development can be found in [6,9,29]. Theme/UML [6] presents an approach to design systems based on the object-oriented model, but extends this model by adding new decomposition capabilities. The new decomposition capabilities support a way of directly aligning design models with individual properties. Each model contains its own theme, or design of an individual requirement, with concepts from the domain (which may appear in multiple requirements) designed from the perspective of that requirement. Crosscutting themes are called "aspects". Standard UML is used to design the models that have been decomposed in this way. A component model is presented in [9] to enable the specification and runtime support of non-functional aspects in component-based systems. A clear separation of non-functional aspects and functionally motivated issues is provided. Non-functional aspects are described orthogonally to the application structure using descriptors, and are woven into the running application by the component container acting as a contract manager. The election of actual implementations depends on the particular client's non-functional properties. An approach to developing secure application using aspect-oriented programming language AspectJ [15] is presented in [29]. The application is a Personal Information Management (PIM) system and there are two security access rules for the system: first, for appointments, the owner of the PIM can invoke all their operations and other persons are only allowed to view them; second, for contacts, only owners can perform their operations. These rules have been implemented as aspects using AspectJ. All three approaches have been focused on introducing the concept of aspects into an early design phase, however they do not provide a systematic approach to define, model, and analyze aspects for software architectures based on UML.

AspectJ is one of the aspect-oriented programming languages. Components of AspectJ include join points, pointcuts, and advices. Fig. 1 presents an aspect in AspectJ. In AOP, join points represent well-defined points. Pointcut is a language construct that picks out a set of join points based on defined criteria. Advice is used to define additional code to be executed before, after, or around join points. An aspect class includes a set of pointcuts and advice.

2.2. Unified modeling language formalization

The semi-formal graphical notation UML is well established in the software engineering community. Its useful capabilities to a software designer include multiple, interrelated views, a semi-formal semantics expressed as a UML meta-model, and a constraint language, the Object Constraint Language. In the case that the current UML is not semantically sufficient, the UML can be extended with three mechanisms: stereotypes, tagged values, and constraints. Meanwhile, several approaches have been proposed to assign UML a formal semantics. In general, there are four ways to assign formal semantics to a language (i.e., UML): axiomatic, operational, denotational, and translational, as described in [10,20]. In the translational semantic approach, as in [14,18,19], models specified in a certain modeling technique are given a semantics by the definition of a mapping to models of a formal language. The operational semantic approach, as in [23], focuses on expressing the semantics of a modeling technique by giving a mechanism that allows us to determine the effect of any model specified in the technique. The denotational semantic approach, as in [5], may be seen as a variant of translational semantics. In this approach, the syntactic constructs of a language

are mapped onto constructs in another language with a well-defined meaning. In contrast with translational semantics, the "target" of denotational semantics is a mathematical domain and not another modeling technique. The axiomatic semantic approach, as in [16], treats a model as a logical theory; it does not try to define what the model means, but what can be proved about the model only. The translational semantic approach has been used in FDAF, where the aspect-oriented UML specification is automatically translated into a formal notation suitable for analyzing an aspect. An evident advantage of the translational semantic approach is that formal verification and validation tools can be used to analyze a specification documented in UML automatically, thus leveraging a large body of work in the research community.

3. Overview of formal design analysis framework

This section presents an overview of the FDAF. As non-functional properties tend to interact with each other, for example, increasing the level of security may adversely affect performance, the long term goal of the FDAF is to support the aspect-oriented design and analysis of multiple (possibly) conflicting non-functional properties using UML and a set of existing formal methods.

The FDAF is illustrated in Fig. 2. In the figure, the FDAF is represented with a rectangle surrounded by the stakeholders, inputs, and outputs of the framework. Stakeholders are those who have a stake in the success of the enterprise, including architects, designers, requirement engineers, etc. who use the framework to develop a system design. Inputs are entities used by the stakeholders, including a system design documented in the UML and a requirement specification that includes the system's functional and non-functional requirements, and a set of formal methods used in the FDAF. Outputs are entities produced by the stakeholders, including a set of aspect-oriented formal design models and the analysis results. FDAF components include an extended UML notation and tool support for the FDAF. The tool provides support for defining an extended UML design model, modules to support the translation of an extended UML design model into formal models, and interfaces with existing formal method tools. More specifically, the tool allows users to browse and select reusable aspects, define new aspects, define join points and advice in a design, weave an aspect into the design, extract an aspect out of the design, assist the user in selecting a formal method, and translate an extended semi-formal UML design into a formal notation. A brief overview of FDAF components is presented below.

Aspect repository. The aspect repository provides a collection of reusable aspects for architects to use. An architect can search the repository to select the appropriate aspect(s) according to the system's non-functional requirements. In the repository, aspects have been classified as two types: *substantive* aspects and *property* aspects. Substantive aspects are aspects that will be constructed in code, such as many security aspects. Substantive aspects are defined in UML as subsystem designs, as these aspects are capabilities delivered by developing classes (e.g., provide secure access control): the static view of the aspect is captured in UML class diagrams, where attributes and operations to realize the aspect capability (e.g., data authentication, access control) are defined; the dynamic view of the aspect is captured in UML sequence diagrams, where the execution behavior of the aspect is illustrated; the OCL is also used to define aspect constraints. Property aspects are aspects that are properties of systems or substantive aspects. The purpose of these aspects is to analyze a system's property, and there is no need to code them in the final product. Examples of such aspects are performance aspects. Property aspects are primarily defined in UML stereotypes with tag values, as these aspects are prescribed properties that permeate all or part of a system (e.g., meet a response time requirement). Guidance for adding an aspect into the system design is also included in the definition using text descriptions with examples.

Aspects are organized hierarchically in the repository, currently including security aspects and performance aspects. The definition of many security aspects is based on their existing security pattern definitions (i.e., the security feature they provide). Security aspects include authentication, confidentiality, authorization, etc., where each of these may be further categorized. The FDAF supports the definition for data origin authentication, role-based access control, and log for audit security aspects. Performance aspects are defined based on well-recognized system performance evaluation criteria, such as response time, resource utilization, probability of errors, etc. The FDAF supports the definition for response time, resource utilization, and throughput performance aspects. The repository is extensible; when a suitable aspect is not found, architects can define their own aspect and extend the repository to include the new aspect.

Extended UML design model. Architects extend the original UML design with additional, specific aspects related to the non-functional properties, creating the extended UML model. UML extensions are needed to support describing

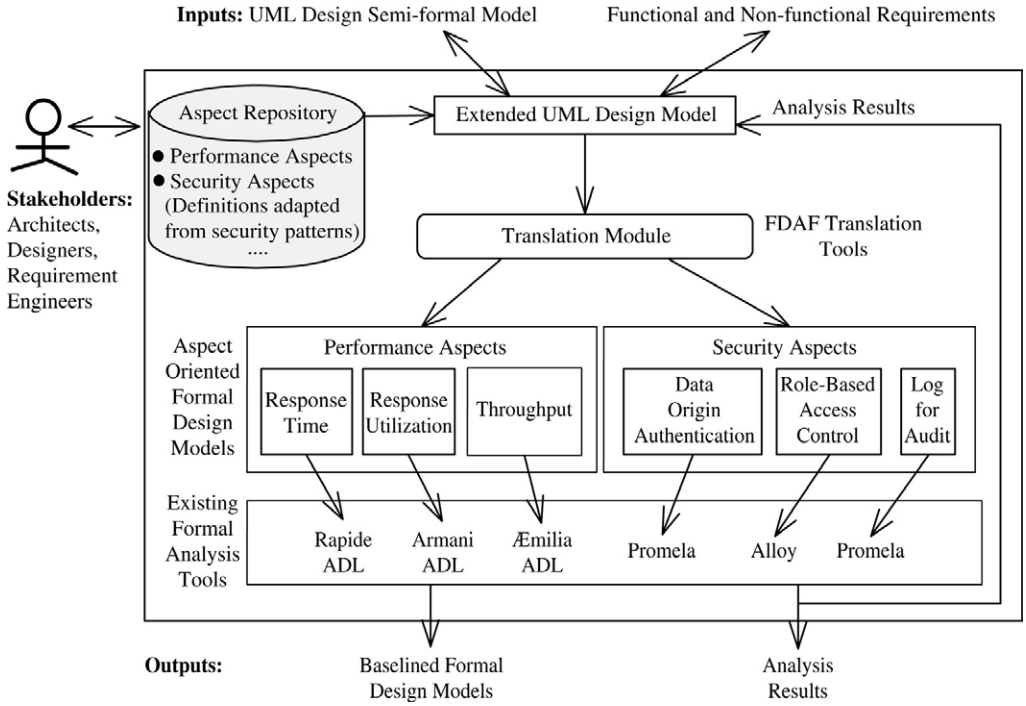

Fig. 2. Formal design analysis framework.

non-functional properties, their automated translation into formal methods, and weaving aspects into an existing design. To distinguish aspect design elements from the conventional UML design elements, a parallelogram notation is used in the FDAF to present aspect related information. The new notation helps architects manipulate aspects, e.g., adding aspects into a design, identifying joint points and advice, extracting aspects out of a design, and automatically generating aspect implementation classes that can be compiled by AspectJ.

Translation module. In the FDAF, the extended UML is formalized by translating into formal notations. An automated translation ensures the consistency between the semi-formal UML models and the formal models. Algorithms have been defined, verified with proofs, and implemented for automating the translation.

Aspect-oriented formal models. These are those models generated by the FDAF translation modules. Using the principle of separation of concerns, a set of simpler models, each built for a specific purpose (or aspect) of the system, can be constructed and evolved relatively independently from other aspect models. This is expected to reduce the complexity of understanding and analyzing the models dramatically. Currently, the FDAF can automatically generate Rapide [17] ADL, Armani [22] ADL, Ælmilia ADL [2], Promela [11], and Alloy [27] specifications from extended UML design models.

Existing formal analysis tools. Once translated, the formal model can be analyzed for specific aspects using its existing tool support. For example, the extended UML model with the response time performance aspect has been translated into Rapide specification. With Rapide's analysis tools, architects can simulate the possible system response time for a design, which makes it easier for architects to evaluate design alternatives. Architects then use analysis results to iteratively modify the semi-formal models and re-create formal models. Armani's analysis tool can analyze resource utilization aspect for an architecture design, where potentially overloaded components can be detected. Ælmilia's analysis tool can analyze action throughput for an architecture design. Promela's analysis tool, SPIN, can be used to analyze whether the data origin authentication or log for audit security aspect has been correctly included into a UML architecture design or not, and, finally, Alloy's analysis tool can detect the inconsistency among multiple access policies for a role-based access control security aspect UML design.

FDAF tool support. Fig. 3 presents a screenshot of the FDAF tool support. The tool has been developed to support the aspect-oriented design in UML, including selecting an aspect from the FDAF aspect repository and adding the aspect into a UML design etc., and the automated translation of an extended UML design into formal specifications (e.g., Rapide, Armani, etc.). It is build based on the open source software engineering tool ArgoUML [1].

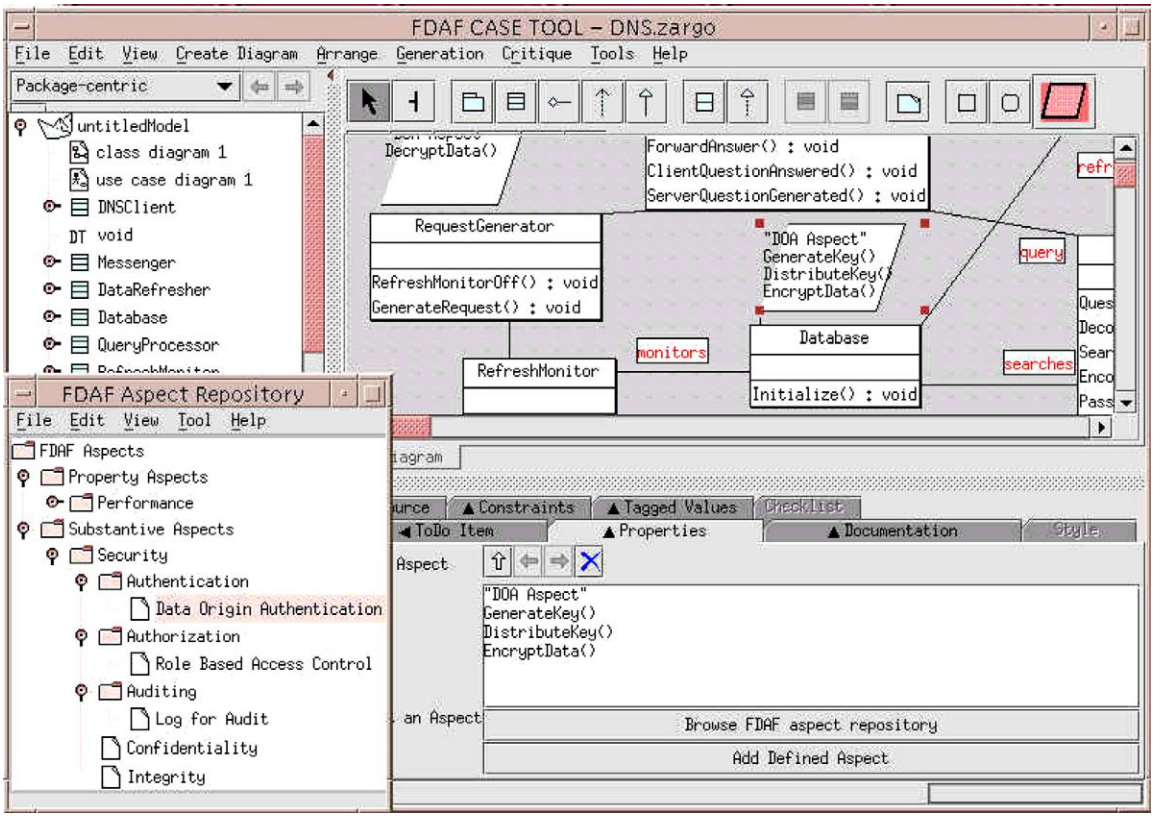

Fig. 3. FDAF tool support screenshot.

4. UML extension for aspect-oriented design

This section presents the FDAF UML extension for capturing aspects. The definition for the data origin authentication security aspect is presented, followed by a discussion of defining aspect-oriented programming concepts (i.e., join point and advice) at the design level, and a weaving example of this aspect. This example also shows how the FDAF UML extension can help architects identify these concepts. In FDAF, aspects are presented using a parallelogram notation.

Data origin authentication (DOA) aspect. Data origin authentication is a security service that verifies the identity of a system entity that is claimed to be the original source of received data. This service has been identified as a problem in the data origin authentication security pattern. A security pattern [26] is a well-understood solution to a recurring information security problem. The ISO security standard (ISO 1989) [12] digital signature mechanism has been presented as a solution to the data origin authentication problem. In the mechanism, a sender has a pair of public and private keys. At the beginning, the sender distributes its public key(s) to potential receivers and encrypts its data with the private key(s). Encrypted data are called digital signatures. During each data transmission, data are sent with these digital signatures. Finally, when receiving all the data, the receiver uses the sender's public key(s) to decrypt the digital signatures and verify the source of the data.

A static view of the defined data origin authentication (DOA) aspect is presented in Fig. 4, where a parallelogram notation is used to present the attributes and operations of the aspect. This new notation is used in the FDAF to present the aspect. Attributes in this aspect include:

- keys, including public and private key
- algorithm, used to encrypt and decrypt data, such as RSA (inventors: Rivest, Shamir, and Adelman) and DSA (Digital Signature Algorithm)
- data, represents the data that are going to be encrypted
- sig, the encrypted data (digital signatures).

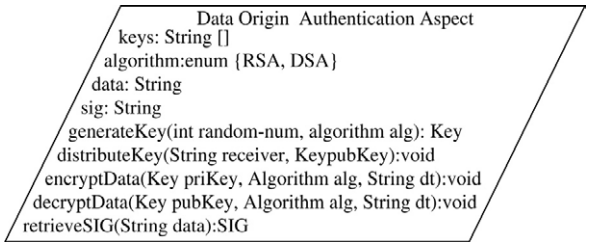

Fig. 4. Data origin authentication aspect.

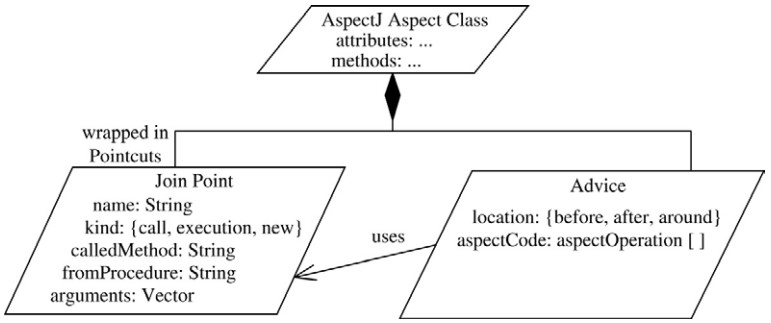

Fig. 5. Join points and advice at the design level.

Several aspect operations are defined to support the use of this aspect; they are called *aspect operation*s:

- generateKey (int random-num, algorithm alg): the operation takes a random number and a specified algorithm as parameters. The output is a key
- distributeKey (String receiver, Key pubKey): the operation sends a public key to a receiver
- encryptData (Key priKey, Algorithm alg, Data dt): the operation encrypts data with the private key and specified algorithm
- decryptData (Key pubKey, Algorithm alg, Data dt): the operation decrypts data with the public key and specified algorithm
- retrieveSIG (Data data): the operation retrieves generated digital signatures for the specified data. The outputs are a collection of digital signatures.

Join points and advice at design level. The semantics of join points and advice have been discussed in [28]. The FDAF adopts these semantics and introduces the concepts of a join point and advice at the design level. At the design level, join point is abstracted as an entity: (*name*, *kind*, *calledmethod*, *fromProcedure*, *arguments*). Descriptions of these attributes are: *name* is the name of a join point; *kind* is an enumerate type which could be "call", "execution", or "new" (i.e., "call" represents method or constructor calls; "execution" represents method or construction execution; and "new" represents static and dynamic initialization); *calledMethod* is the name of the method or constructor being called, executed, or instantiated; *fromProcedure is* the name of the procedure (e.g., Java classes) from where the method or constructor is called; *arguments* are parameters for this join point. In AspectJ, join points are represented by the pointcut language construct. An advice is abstracted as an entity: (*location, AspectCode, Joinpoints).* The *location* attributes specify when (i.e., before, after, around) to run the aspect program statement; *AspectCode* encapsulates these program statements; and *Joinpoints* represents the join points used by the advice. These statements are a part of an aspect's definition and they are encapsulated in *aspect operation*s. Aspects in AspectJ are implemented in a Java class with the keyword "aspect" and can contain pointcuts, advice, methods, fields, etc. The relationships among AspectJ aspect class, join point, and advice are presented in Fig. 5.

Weaving an aspect into UML design. An example of weaving an aspect into a UML class diagram is presented in Fig. 6. To weave an aspect into a UML class diagram, architects decide the "places" (i.e., UML conventional operations) from where one or more aspect operations need to be added. Then, architects "join" these two kinds of operations together. This "join" relationship is represented by a black solid arrow. In Fig. 6, the UML conventional operation "processAnswer()" is joined with an *aspect operation*, "decryptData(. . .)", which is from the data origin

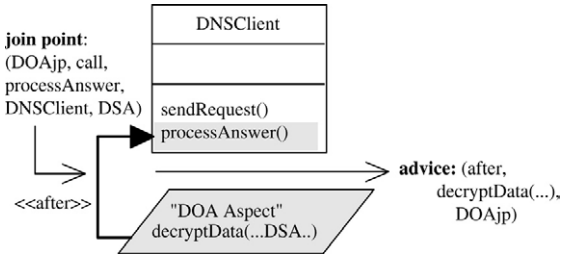

Fig. 6. Weaving an aspect into a UML design.

authentication aspect. A "join" relationship identifies a join point, such as: (DOAjp, call, processAnswer, DNSClient, DSA), where DSA represents the digital signature algorithm used to instantiate the decrypting algorithm in decryptData(. . .) aspect operation. The "join" relationship is also denoted by the phrase "≪after≫", which expresses that the execution of "decryptData(. . .)" occurs after the execution of the execution of "processAnswer()". As the join point DOAjp has been defined for "processAnswer()", advice is then identified as: (after, decryptData(. . .), DOAjp). The identified join point and advice are used to develop AspectJ aspect classes in the coding phase. This will facilitate the implementation of substantive aspect, in the system, which will help reduce development cost and time.

5. Translating algorithm: Extended UML to Rapide

Once the DOA security aspect is woven into a UML design, the response time of the design can be analyzed in the FDAF; this is achieved by using the FDAF automated translation of an extended UML design into Rapide [17], an ADL, whose analysis tool allows the architectural response time simulation. Response time has also been defined as an aspect [7]. The response time aspect analysis in Rapide requires three UML diagrams for the translation: class, statechart, and sequence diagrams. The algorithms for translating these diagrams have been defined, proven, and implemented in the FDAF tool. Complete presentations of these algorithms and their proofs of correctness are available in [7]. Here, the algorithm of translating an extended UML class diagram into Rapide is presented in Table 1 and its proof is presented in Table 2. Assume that a UML class diagram includes N classes, the maximum number of attributes a class has is X, and the maximum number of operations a class has is Y; in the worst case, the time complexity of this algorithm is O(N*(X+Y)).

6. Illustration: Modeling and analysis of security aspects

This section presents the modeling and analysis of the response time performance cost for the data origin authentication security aspect in the Domain Name system. A brief description of the DNS is provided. The FDAF steps include: create semi-formal UML architecture base model, extend the base model with DOA aspect, extend DOA aspect with response time aspect, and translate extended UML model into *Rapide* to analyze response time for the security aspect.

Domain Name System. The DNS [21] is a complex system with a rich set of functional and non-functional properties. The DNS provides a mechanism of conversion with a double functionality: it translates both symbolic host names to IP address and IP addresses to host names. A particular scenario of the DNS is DNS client sends queries to a DNS server and the DNS server processes these queries. The DNS is selected as an example system because it is real-time, distributed, needs to be secure, and (optionally) supports recursive queries. In addition, the DNS is a non-proprietary standard.

Create UML architecture base model. The DNS UML architecture base model in the UML class diagram is presented in Fig. 7(A). In the diagram, the "DNSClient" represents a DNS client, which sends queries to DNS servers. The rest of the UML classes belong to a DNS server. "Messenger" is used to send and receive messages; "Query-Processor" is used to process clients' queries; "Refresh-Monitor" monitors whether the data in the database has expired or not; if some data has expired, then the "Request-Generator" generates a request and sends it to other DNS servers; and "Data-Refresher" refreshes the database.

Extend UML architecture base model with DOA aspect. As defined in its static view, the data origin authentication aspect has several aspect operations to use. Architects extend the UML class model by weaving each aspect operation with the correct UML operations (Fig. 7B). The weaving rationale for this example is:

Table 1
Algorithm for translating UML extended class diagram into Rapide

```
Input: A UML class diagram CD, and a security aspect A, where CD.Classes = {C1, C2, C3 ... Cm | Ci is a UML class, 1 ≤ i ≤ m}
(|CD.Classes| = m), and A.Operations = {A1, A2, A3 ... Ax | Ai is an aspect operation, 1 ≤ i ≤ x}.
Output: S, which is a finite set of Rapide interface types and defined as S = {R1, R2, R3 ... Rn | Ri is a Rapide interface type, 1 ≤ i ≤ n}
(|S| = n), and satisfies m = n.
Translate (CD.Classes = {C1, C2, C3 ... Cm}) : S
S ← ∅;
while CD.Classes ≠ ∅
  newR : a Rapide type;
  uC ∈ CD.Classes;
  newR.Name ← uC.Name;
  while uC.Attributes ≠ ∅
    uAttr ∈ uC.Attributes;
    newRC : RC;
    newRC.Identifier ← uAttr.Name;
    newRC.Denotation_Speficification ← uAttr.Type;
    if (uAttr.Visibility == "public") || (uAttr.Visibility == "protected")
      newR.Provide-Interface ← newR. Provide-Interface ∪ {newRC};
    else
      newR.Private-Interface ← newR.Private-Interface ∪ {newRC};
    uC.Attributes ← uC.Attributes - {uAttr};
  while uC.Operations ≠ ∅
    uOpera ∈ uC.Operations;
    newRC : RC;
    newRC.Identifier ← uOpera.Name;
    newRC.Denotation-Speficification ← uOpera.Parameters + uOpera.type;
    if (uOpera.Visibility == "public") || (uOpera.Visibility == "protected")
      if (uOpera.Invocation-Style == 'synchronous')
        newR.Provides-Interface ← newR.Provides-Interface ∪ {newRC};
      else
          if (uOpera.Event-Occurrence == "generation")
            newR.OutAction-Interface ← newR.OutAction_Interface ∪{newRC};
          else newR.InAction-Interface ← newR.InAction_Interface ∪{newRC};
    else newR.Private-Interface ← newR.Private-Interface ∪ {newRC};
  if (uOpera is joined with aspect operations)
    while there is an aspect operation
      newRC : RC;
      newRC.Identifier ← aOpera.Name;
      newRC.Denotation-Speficification ← aOpera.Parameters + aOpera.type;
      newR.Private-Interface ← newR.Private-Interface ∪ {newRC};
      A.Operations ← A.Operations - {aOpera};
  uC.Operations ← uC.Operations - {uOpera};
CD.Classes ← CD.Classes - {uC};
S ← S ∪ {newR};
```

- DOA aspect operations generatkey(...), distributeKey(...), encryptData(...) need to be executed after the "Database" initializes (initialize(...))
- DOA aspect operation retrieveSIG(...) needs to be executed after a client query is processed (searchAnswer())
- DOA aspect operation decryptData(...) needs to be executed after a request answer is processed (processAnswer()).

Extend DOA aspect with response time aspect. To analyze the response time performance cost for the DOA, the DOA aspect is extended with the response time aspect (Fig. 7C). Other DOA operations are extended the same way, however they are not presented here due to space constaints. The response time aspect is defined by a stereotype ≪PAstep≫. This stereotype is defined in [24]. It models a step in a performance analysis scenario and has a defined tag PAdemand. PAdemand = ('assm', 'mean', (5, 'ms')) means that the mean execution time of the step is assumed to be 5 ms.

Translate extended UML model into rapide. Finally, the extended model is automatically translated into Rapide specification by the FDAF tool support. The result Rapide specification is sent to Rapide's analysis tool for

Table 2
Algorithm proof

Algorithm Verification:

Part 1 (Verify the "Attributes" while loop, P1):

/ Pre: uC is a UML class* $\wedge$ *|uC.Attributes|* $= X \wedge$ *|newR.Provide-Interface|* $= Y \wedge$ *|newR.Private-Interface|* $= Z$ */

while uC.Attributes $\neq \varnothing$

……

*/*Post: X* - *|uC.Attributes|* = *|newR.Provider-Interface|* - *Y* + *|newR.Private-Interface|* - *Z */*

Proof:

(1) On the initial entry to the loop:

|uC.Attributes| $= X \wedge$ *|newR.Provide-Interface|* $= Y \wedge$ *|newR.Private-Interface|* $= Z$ (from Pre), and X - *|uC.Attributes|* $= X - X = 0$, *|newR.Provider-Interface|* - *Y* + *|newR.Private-Interface|* - *Z* = *Y- Y* + *Z* – *Z* = *0*, thus,

P : X - *|uC.Attributes|* = *|newR.Provider-Interface|* - *Y* + *|newR.Private-Interface|* - *Z* is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume |uC.Attributes| = A, |newR.Provide-Interface| = B, |newR.Private-Interface| = C, and X – A = B – Y + C – Z.

The after the iteration, two cases:

Case (i); *(uAttr.Visibility == 'public')* || *(uAttr.Visibility == 'protected'):*

From *newR.Provide-Interface* ← *newR.Provide-Interface* ∪ {*newRC*}

|newR.Provide-Interface| = |newR.Provide-Interface| + 1 = B + 1

From *uC.Attributes* ← *uC.Attributes* - {*uAttr*}

|uC.Attributes| = |uC.Attributes| - 1 = A -1

Thus, X – (A – 1) = B + 1 – Y + C – Z

Case (ii); ¬ [*(uAttr.Visibility == 'public')* || *(uAttr.Visibility == 'protected')*]:

From *newR.Private-Interface* ← *newR.Private-Interface* ∪ {*newRC*}

|newR.Private-Interface| = |newR.Private-Interface| + 1 = C + 1

From *uC.Attributes* ← *uC.Attributes* - {*uAttr*}; |uC.Attributes| = |uC.Attributes| - 1 = A -1

Thus, X – (A – 1) = B – Y + C + 1– Z; After the iteration, Post is true;

(3) On exit from the loop:

P $\wedge$ ¬(uC.Attributes $\neq \varnothing$) = uC.Attributes = $\varnothing \wedge$ P

→ X = |newR.Provider-Interface| - Y + |newR.Private-Interface| - Z

→ X - |uC.Attributes| = |newR.Provider-Interface| - Y + |newR.Private-Interface| - Z = post

So long as the loop has not terminated,

P $\wedge$ (uC.Attributes $\neq \varnothing$) = X - |uC.Attributes| = |newR.Provider-Interface| - Y + |newR.Private-Interface| - Z $\wedge$ (uC.Attributes $\neq \varnothing$)

→ |uC.Attributes| $> 0 \rightarrow$ t > 0 (t is an integer-valued function)

After each iteration, one element in uC.Attributes is deleted from the set. This implies that |uC.Attributes| (i.e., t) is decreased by a positive integer amount.

Thus the loop will terminate, and the part of algorithm is correct.

Part 2 (Verify the "Operations" while loop, P2):

/ Pre: uC is a UML class* $\wedge$ *|uC.Operations|* $= X \wedge$ *|newR.Provide-Interface|* $= Y \wedge$ *|newR.OutAction-Interface|* $= Z \wedge$ *|newR.InAction-Interface|* $= M \wedge$ *|newR.Private-Interface|* $= N \wedge$ *A is an aspect* $\wedge$ *|A.operations|*$=K$ */*

while uC.Operations $\neq \varnothing$

……

*/*Post: X* - *|uC.Operations|* + *|A.Operations|* - *K*= *|newR.Provider-Interface|* - *Y* + *|newR.OutAction-Interface|* - *Z* + *|newR.InAction-Interface|* -*M* + *|newR.Private-Interface|* - *N */*

Proof:

(1) On the initial entry to the loop:

|uC.Operations| $= X \wedge$ *|newR.Provide-Interface|* $= Y \wedge$ *|newR.OutAction-Interface|* $= Z \wedge$ *|newR.InAction-Interface|*$= M \wedge$ *|newR.Private-Interface|* $= N \wedge$ *|A.operations|*$=K$ (from Pre), and X - *|uC.Operations|* + *|A.Operations|* - *K* = *X* – *X* + *K* - *K* = *0*, *|newR.Provider-Interface|* - *Y* + *|newR.OutAction-Interface|* - *Z* + *|newR.InAction-Interface|* - *M* + *|newR.Private-Interface|* - *N* = *Y- Y* +*Z* – *Z* + *M* – *M* + *N* - *N* = *0*,

P : X - *|uC.Operations|* + *|A.Operations|* - *K* = *|newR.Provider-Interface|* - *Y* + *|newR.OutAction-Interface|* - *Z* + *|newR.InAction-Interface|* -*M* + *|newR.Private-Interface|* - *N* is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume |uC.Operations| = A, |A.Operations| = T, |newR.Provide-Interface| = B, |newR.OutAction-Interface| = C, |newR.InAction-Interface| = D, |newR.Private-Interface| = E, and X – A + K - T = B – Y + C – Z + D – M + E - N.

The after the iteration, four cases:

Case (i); *[(uAttr.Visibility == 'public')* || *(uAttr.Visibility == 'protected')] && (uOpera.Invocation-Style == 'synchronous'):*

From *newR.Provide-Interface* ← *newR. Provide-Interface* ∪ {*newRC*}

|newR.Provide-Interface| = |newR.Provide-Interface| + 1 = B + 1

From *uC.Operations* ← *uClass.Operations* - {*uOpera*}

|uC.Operation| = |uC.Operation| - 1 = A -1

Thus, X – (A – 1) + K - T = B + 1 – Y + C – Z + D – M + E – N
Case (ii); *[(uAttr.Visibility == 'public') || (uAttr.Visibility == 'protected')] &&*
¬ (uOpera.Invocation-Style == 'synchronous') && (uOpera.Event-Occurrence == 'generation'):
From *newR.OutAction-Interface* ← *newR.OutAction_Interface* ∪{*newRC*}
|newR.OutAction-Interface| = |newR.OutAction-Interface| + 1 = C + 1
From *uC.Operations* ← *uC.Operations* - {*uOpera*}
|uC.Operations| = |uC.Operations| - 1 = A -1
Thus, X – (A – 1) + K - T = B – Y + C + 1 – Z + D – M + E – N
Case (iii); *[(uAttr.Visibility == 'public') || (uAttr.Visibility == 'protected')] &&*
¬ (uOpera.Invocation-Style== 'synchronous')&& ¬ (uOper.Event-Occurrence == 'generation'):
From *newR.InAction-Interface* ← *newR.InAction_Interface* ∪{*newRC*}
|newR.InAction-Interface| = |newR.InAction-Interface| + 1 = D + 1
From *uC.Operations* ← *uC.Operations* - {*uOpera*}
|uC.Operations| = |uC.Operations| - 1 = A -1
Thus, X – (A – 1) + K - T = B – Y + C – Z + D + 1 – M + E – N
Case (iiii); ¬ [*(uAttr.Visibility == 'public') || (uAttr.Visibility == 'protected')*]:
From *newR.Private-Interface* ← *newR.Private-Interface* ∪ {*newRC*}
|newR.Private-Interface| = |newR.Private-Interface| + 1 = E + 1
From *uC.Operations* ← *uC.Operations* - {*uOpera*}
|uC.Attributes| = |uC.Attributes| - 1 = A -1, Thus, X – (A – 1) + K - T= B – Y + C – Z + D – M + E + 1 – N
Case (V); if the operation is joined with P aspect operation, X - (A - 1) + K - (T - P) = B – Y + C + P – Z + D – M + E + 1 – N
After the iteration, Post is true.

(3) On exit from the loop:
P ∧ ¬(uC.Operations ≠ ∅) = uC.Operations = ∅ ∧ P
→ X = |newR.Provider-Interface| - Y + |newR.OutAction-Interface| - Z + |newR.InAction-Interface| - M + |newR.Private-Interface| - N
→ X - |uC.Attributes| = |newR.Provider-Interface| - Y + |newR.OutAction-Interface| - Z+ |newR.InAction-Interface| - M
+ |newR.Private-Interface| - N = post

(4) So long as the loop has not terminated,
P ∧ (uC.Operations ≠ ∅) = X - |uC.Operations| = |newR.Provider-
Interface| - Y + |newR.OutAction-Interface| - Z + |newR.InAction-Interface| - M
+ |newR.Private-Interface| - N ∧ (uC.Operations ≠ ∅)
→ |uC.Operations| > 0 → t > 0 (t is an integer-valued function)

(5) After each iteration, one element in uC.Operations is deleted from the set. This implies that |uC.Operations| (i.e., t) is decreased by a positive integer amount.
Thus the loop will terminate, and the part of algorithm is correct.

Part 3 (Verify the whole algorithm):
/ Pre: CD is a UML class diagram ∧ |CD.Classes| = X ∧ |S| = 0 */*
S ← ∅;
while CD.Classes ≠ ∅
. . . . . .
*/*Post: X - |CD.Classes| = |S| */*

Proof:
(1) On the initial entry to the loop:
|*CD.Classes*| = *X* ∧ |*S* | = *0* (from Pre), and *X* - |*CD.Classes*| = *X* – *X* = *0*, |S| = 0, thus, ***P*** : *X* - |*CD.Classes*| = |*S*| is true.

(2) Suppose that P is true before an arbitrary loop iteration. Assume |CD.Classes| = A, |S| = B, and X – A = B .
From *CD.Classes* ← *CD.Classes* - {*uC*}; (line 37)
|CD.Classes| = |CD.Classes| - 1 = A - 1
From *S* ← *S* ∪ {*newR*} (line 38)
|S| = |S| + 1 = B + 1, thus, X – (A – 1) = B + 1

(3) On exit from the loop:
P ∧ ¬(CD.Classes ≠ ∅) = CD.Classes = ∅ ∧ P → X = |S|→ X - |CD.Classes| = |S|= post

(4) So long as the loop has not terminated,
P ∧ (CD.Classes ≠ ∅) = X - |CD.Classes| = |S| ∧ (CD.Classes ≠ ∅)
→ |CD.Classes| > 0 → t > 0 (t is an integer-valued function)

(5) After each iteration, one element in CD.Classes is deleted from the set. This implies that |CD.Classes| (i.e., t) is decreased by a positive integer amount.
As P1 and P2 have been proved to be correct, thus the loop will terminate, and the whole algorithm is correct.

architectural simulation. The response time of this DNS example without the data origin authentication aspect has already been analyzed in previous work [7], where the average response time for the system is 43 logic time units. A screenshot of the Rapide analysis tool is presented in Fig. 7D. Each row in the output lists information of events

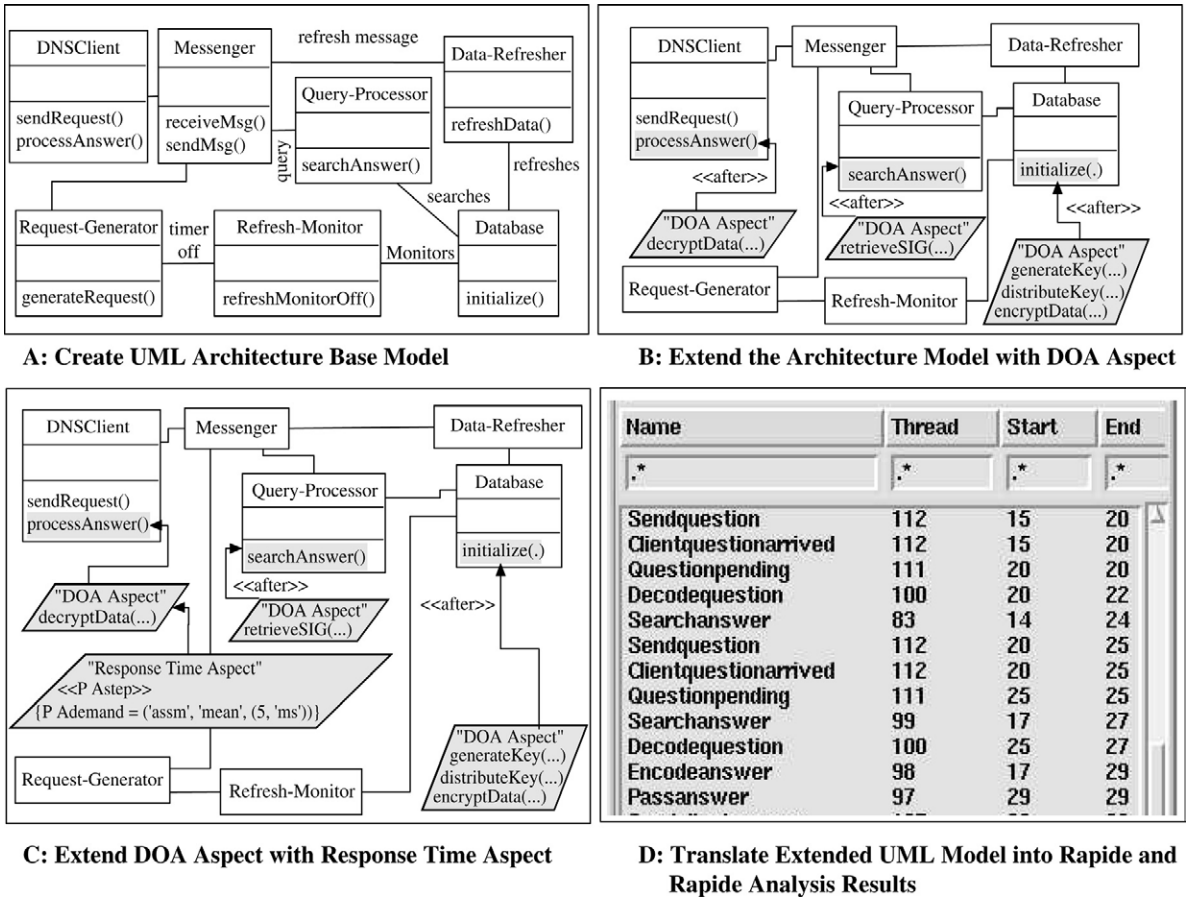

Fig. 7. FDAF approach illustration using the DNS.

generated during the simulation, which includes the events' start timestamp and end timestamp. In this simulation, the DNS server's mean response time (by computing the events' timestamps) to a client is 54 logic time units. The performance cost of the data origin authentication aspect is 11 time units on average, which is an increase of 26%. In the next step of the work, different encryption algorithms are going to be used in the simulation, and the results will help architects to perform tradeoff analysis.

7. Discussion

This section presents a discussion of the FDAF approach.
The strengths of the FDAF approach include:

- In the FDAF, security patterns are adapted as security aspects. Security aspects are defined using the well-known design notation UML with attributes and operations, thus providing more concrete information regarding the inclusion of security aspects into a UML design. This helps bridge the gap between software architects and security professionals
- The FDAF provides automated translation from an extended UML design into formal specifications. This facilitates the use of existing useful tools to analyze an architecture design, where defects and errors can be detected earlier
- The FDAF approach is flexible and extensible: new aspects can be added into the repository and, meanwhile, corresponding formal analysis tools can be included.

The issues of the FDAF approach:

- There are also performance issues related to the dynamic view of software architecture, for example the architecture's configuration under a specific runtime environment, the interaction between multiple processes with

difference priority, and the shared resource distribution between multiple processes, data persistency, etc. The modeling and analysis of the dynamic properties of a software architecture is an important research problem, which needs to be addressed in the FDAF

- The FDAF aspect repository needs to be maintained and, as more aspects are included, the repository should support sophisticated searching techniques
- The analysis in the FDAF is limited by the existing formal analysis tools. For example, in the response time aspect analysis, as Rapide's timing model only supports fixed delays, the stochastic delays (e.g., the "mean" delay) in the DNS example have been translated to fixed delays under the assumption that the variance of these delays is 0. This has affected the accuracy of the performance analysis results, since commonly used distributions (e.g., exponential distribution) are not supported.

8. Conclusions and future work

Software architecture is an area of software engineering directed at developing large, complex applications in a manner that reduces development costs, increases the quality, and facilitates evolution.

The paper presents a UML based approach to model and analyze aspect-oriented designs called the Formal Design Analysis Framework. To support the modeling of aspect-oriented designs, the FDAF provides a repository of predefined, reusable aspects for designers to use, including performance and security aspects. In addition, aspect-oriented programming components join point and advice are abstracted into the design level to support the weaving of reusable aspects in a UML design. Another important contribution of FDAF is that it integrates the well-established, semi-formal UML and a set of existing formal methods into one aspect-oriented framework, and uses existing formal tools to analyze aspect-oriented designs. In the paper, a security aspect, the data origin authentication aspect, is defined. The Domain Name System example has been used to illustrate weaving this aspect and its response time performance aspect into a UML design. The performance cost of including the data origin authentication security aspect is analyzed by the automated translation of extended UML into Rapide. The analysis results show an increase of 26% in response time with the inclusion of the security aspect. Such analysis results are very valuable at the design phase. With the help of these results, it would be easier for architects to select suitable designs for a system between alternatives and/or revise their designs, and subsequently, improve the quality of the system.

Ultimately, the FDAF is intended to support the design and analysis of multiple, conflicting or synergistic non-functional properties. There are several interesting directions for the future work of the FDAF. One direction is to investigate the modeling and analysis of security aspects, and the interactions among performance and security aspects. The NFR Framework [4] is going to be used to systematically analyze the synergistic and conflicting relationships among the aspects. This is expected to be an interesting and challenging problem, as it necessitates a measurement, either quantitative or qualitative, of security capabilities (e.g., one encryption algorithm is in some way better or worse than another and by how much). A second direction is to augment the FDAF aspect repository with additional aspects and explore the possibility of using UML 2.0 to support modeling for aspect-oriented designs.

References

[1] ArgoUML, archived at http://argouml.tigris.org/.
[2] S. Balsamo, M. Bernardo, M. Simeoni, Combining stochastic process algebras and queueing networks for software architecture analysis, in: Proceedings of ACM International Workshop on Software and Performance, 2002, pp. 190–202.
[3] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, Reading, MA, 2003.
[4] L. Chung, B. Nixon, E. Yu, J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishing, 2000.
[5] T. Clark, A. Evans, Foundations of the unified modeling language, in: Proceedings of the Second Northern Formal Methods Workshop, Electronic Workshops in Computing, Springer Verlag, 1998.
[6] S. Clark, R. Walker, Generic aspect-oriented design with Theme/UML, in: Aspect-Oriented Software Development, Addison Wesley Professional, 2005, pp. 425–459.
[7] K. Cooper, L. Dai, Formal modeling and analysis of performance aspects in software architectures, Technical Report UTDCS-31-04, The University of Texas at Dallas, August 2004.
[8] R. Filman, T. Elrad, S. Clarke, M. Aksit, Aspect-Oriented Software Development, Addison Wesley Professional, 2005.
[9] S. Göbel, C. Pohl, S. Röttger, S. Zschaler, The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects, in: Proceedings of the Third International Conference on Aspect-Oriented Software Development, 2004, pp. 74–82.
[10] A.H.M. ter Hofstede, H.A. Proper, How to formalize it? Formalization principles for information system development methods, Information and Software Technology 40 (10) (1998) 519–540.

[11] G.J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
[12] ISO 7498-2: Information processing systems – Open Systems Interconnection – Basic reference Model – Part 2: Security Architecture, 1989.
[13] S. Keller, L. Kahn, R. Panara, Specifying software quality requirements with metrics, in: Tutorial: System and Software Requirements Engineering, IEEE Computer Society Press, 1990, pp. 145–163.
[14] M. Kyas, H. Fecher, F.S. Boer, J. Jacob, J. Hooman, M. Zwaag, T. Arons, H. Kugler, Formalizing UML models and OCL constraints in PVS, in: Proceedings of Semantic Foundations of Engineering Design Languages, SFEDL'04, 2004.
[15] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, 2003.
[16] K. Lano, J. Bicarregui, A. Evans, Structured axiomatic semantics for UML models, in: Proceedings of Rigorous Object-Oriented Methods, ROOM 2000, 2000.
[17] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Transactions on Software Engineering 21 (4) (1995) 336–354.
[18] T. Massoni, R. Gheyi, P. Borba, A UML class diagram analyzer, in: Proceedings of the Third International Workshop on Critical Systems Development with UML, 2004.
[19] W.E. McUmber, B. Cheng, A general framework for formalizing UML with formal languages, in: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 433–442.
[20] B. Meyer, Introduction to the Theory of Programming Languages, Prentice-Hall, 1990.
[21] P.V. Mockapetris, Domain Names — Implementation and Specification, IETF STD0013, November 1987.
[22] R.T. Monroe, Capturing software architecture design expertise with Armani, Technical Report No. CMU-CS-98-163, Carnegie Mellon University School of Computer Science, 1998.
[23] I. Ober, An ASM semantics for UML derived from the meta-model and incorporating actions, in: Proceedings of the Tenth International Workshop on Abstract State Machines, 2003, pp. 356–372.
[24] Object Management Group, UMLTM Profile for Schedulability, Performance, and Time Specification, OMG Documents ptc/2003-03-02, 2003.
[25] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, second ed., Addison-Wesley, 2004.
[26] Security Patterns Home Page, achieved at http://www.securitypatterns.org/.
[27] Software Design Group, the Alloy Analyzer, archived at http://alloy.mit.edu, 2002–2005.
[28] M. Wand, G. Kiczales, C. Dutchyn, A semantics for advice and dynamic join points in aspect-oriented programming, ACM Transactions on Programming Languages and Systems 26 (5) (2004) 890–910.
[29] B. Win, W. Joosen, F. Piessens, Developing secure applications through aspect-oriented programming, in: Aspect-Oriented Software Development, 2005, pp. 633–651.