Reflection and Object-Oriented Analysis

Walter Cazzola, Andrea Sosio, and Francesco Tisato

DISCO - Department of Informatics, Systems, and Communication,
University of Milano - Bicocca, Milano, Italy
{cazzola|sosio|tisato}@disco.unimib.it

Abstract. Traditional methods for object-oriented analysis and modeling focus on the functional specification of software systems. Non-functional requirements such as fault-tolerance, distribution, integration with legacy systems, and the like, do not have a clear collocation within the analysis process, as they are related to the architecture and workings of the system itself rather than the application domain. They are thus addressed in the system's design, based on the partitioning of the system's functionality into classes as resulting from the analysis. As a consequence of this, the "smooth transition from analysis to design" that is usually celebrated as one of the main advantages of the object-oriented paradigm does not actually hold for what concerns non-functional issues. Moreover, functional and non-functional concerns tend to be mixed at the implementation level. We argue that the reflective design approach whereby non-functional properties are ascribed to a meta-level of the software system may be extended "back to" *analysis*. Reflective Object Oriented Analysis may support the precise specification of non-functional requirements in analysis and, if used in conjunction with a reflective approach to design, recover the smooth transition from analysis to design in the case of non-functional system's properties.

1 Introduction

Traditional methods for object-oriented analysis and modeling focus on the functional specification of software systems. The relevant concepts from the *application domain* are modeled using concepts (classes, object, operation, attributes, associations between classes, and so on) whose scope hardly includes non-functional requirements such as fault-tolerance, distribution, performance, persistence, security, and so on. These are not related to properties of the entities in the "real world", but rather to *properties of the software objects that represent those entities*. Such non-functional requirements play a major role in the contract between customer and developer, and are usually included in analysis documents, maybe in the form of labels or "stereotypes" attached to analysis classes. Nevertheless, their treatment lacks a clear collocation in traditional object oriented processes. As a consequence, they tend to be less precisely specified in analysis, and they do not enjoy the "smooth transition from analysis to design" which is usually ascribed as one of the main advantages of the object oriented paradigm.

In this paper, we argue that a reflective approach is well suited to address this problem. Reflection allows us to address non-functional requirements using traditional OO concepts, although these apply to a *meta-level* computation rather than normal ("base-level") computation. If traditional OO concepts are used, notations, meta-models, methods, and methodologies used for conventional OO analysis can be employed in the analysis of non-functional properties. Then, if we also assume a reflective approach to *design*, the smooth transition from analysis to design is also recovered. The main purpose of this paper is that of illustrating this idea.

Although reflection has gained increasing attention within the last decade, and is now recognized as a relevant and useful feature in OO languages and programming systems, there is still a lack of research efforts devoted to the definition of an OO *process* for reflective systems. As this paper proposes an integrated reflective approach to analysis and design, it can also be regarded as an attempt to shed some light on what such a process might look like.

The outline of the paper is as follows. Section 2 discusses the problems encountered in traditional object oriented analysis for what concerns the treatment of non-functional requirements. Section 3 lists some major concepts from the discipline of (OO) reflection, which provide a basis for solving those problems. Section 4 illustrates our proposal of a *reflective object-oriented analysis*, discussing how non-functional requirements fit into such an approach to OO analysis, and also pointing at the applications of the proposed approach to the design of reflective systems. Section 5 briefly points at related works in the discipline of reflection, and section 6 draws some conclusions.

2 Non-Functional Requirements and Traditional Object-Oriented Analysis

The first step in traditional OO analysis is related to the modeling of the *application domain*, and results in the definition of the collection (hierarchy) of classes corresponding to the relevant concepts in the domain and of the relationships (associations) between those classes. Following a traditional, consolidated style of software engineering, the authors of those methods insist that implementation details ("how") should be (as systematically as possible) ignored when specifying a system ("what"). Once analysis is "complete", the design begins based on the classes found in the analysis, that are refined and progressively enriched with details about "how" they are going to meet their specification. In this process, new classes may be added (sometimes referred to as "architectural" as opposed to "application" classes) whose purpose is that of providing a concrete infrastructure for the analysis classes to work. One of the major benefits coming from object-orientation is the smooth transition from analysis to design; the classes that are found in the analysis phase often preserve their original interfaces and overall interrelationships when they are refined into more concrete design classes, although several details are usually added.

The general lines of this process are of course valuable and could hardly be criticized per se. Nevertheless, we believe that there is a missing piece, namely, *the treatment of non-functional requirements has no clear collocation in most traditional OO processes*. Non-functional requirements on the system, such as issues related to fault-tolerance, distribution, performance, persistence, integration with legacy or pre-existing systems, networking, authentication and security, and so on, maybe as relevant to the customer as functional ones. (Also relevant may be non-functional requirements on the process, such as limitations to budget or time to market, development resources, or the need to reuse COTS components, although we will not discuss this topic in this paper). The problem with non-functional requirements is that they are not easily captured by traditional OO concepts as employed by traditional OO modeling notations and meta-models. While saying that money may be drawn from a bank account may result, say, in the definition of a withdraw operation in class *bank_account*, saying that the information about a bank account is persistent (or fault-tolerant, etc.) has no counterpart in traditional object oriented concepts. Of course, it is possible to specify that the account information must be persistent in the analysis of a banking system; using the UML, for example, a *stereotype* “persistent” could be attached to the relevant class(es). The problem occurs in the *transition from analysis to design*: while functional requirements are relatively easily translated into design elements (concrete classes, methods, attributes), this does not hold for non-functional properties. Thus, the celebrated property of OO techniques of supporting a smooth transition from analysis to design (and to implementation) does not hold for many relevant features of a software system.

The immediate consequence of this state of facts is that implementing non-functional properties is a less clearly constrained and guided activity than implementing functional properties, i.e., such properties are less clearly specified and tend to be more obscurely implemented. Moreover, they are necessarily tackled *based on the functional partitioning that resulted from analysis*. This may be misleading and contribute to bad design, especially for those non-functional requirements that aren’t naturally related to *any specific object*, but rather require a system-wide infrastructure. A typical result is that code related to such issues gets intermixed with “functional” code in the implemented system, thus reducing modifiability and reusability.

Other drawbacks of this situation can be highlighted. For example, the fact that non-functional properties are not clearly specified in analysis documents reduces the effectiveness of requirements’ analysis as a contract between customer and developer.

An example that can be mentioned to clarify these points is that of designing authentication features for secure transactions in a banking system, an issue that was studied by one of the authors (W. Cazzola [2]). Consider the case of a *withdraw* operation to be implemented for ATM transactions. Functionally, the withdrawal is just a movement of money. Nevertheless, it requires a complex “non-functional” infrastructure including concurrency control, fault-tolerance support, authentication, and possibly more. In a traditional OO process, the designer may receive, as an outcome of analysis, a class ATM providing an operation *withdraw* labeled as *atomic*, *secure*, *reliable*, and so on. The designer will probably cope with these additional properties refining the *withdraw* operation into a very complex activity (perhaps described by a state diagram with dozens of states and transitions). The resulting implementation is necessarily one where the basic semantics of *withdraw* is obscured and dispersed in the midst of a plethora of additional code that has little to do with the movement of money *per se*, thus making the ATM object harder to reuse and modify. Changes in the analysis documents (e.g., if the customer asks for a higher level of security) provide no hint as to how the system design and implementation should be changed (e.g., changes cannot be traced easily from analysis onto design and implementation).

3 Object Oriented Reflection

3.1 Basic Concepts

Computational reflection (or *reflection* for short) is defined as the activity performed by an agent when doing computations about itself [12]. The concept applies quite naturally to the OOP paradigm [5, 12, 7]. Just as objects in conventional OOP are representations of "real world" entities, computation objects can themselves be represented by other objects, usually referred to as *meta-objects*, whose computation is intended to observe and modify their *referents* (the objects they represent). Meta-computation is often performed by meta-objects by *trapping* the normal computation of their referents; in other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either substituting or encapsulating the referent's actions. Of course, meta-objects themselves can be represented, i.e., they may be the referents of meta-meta-objects, and so on. A reflective system is thus structured in multiple levels, constituting a *reflective tower*. The objects in the base level are termed *base-objects* and perform computation on the entities of the application domain. The objects in the other levels (termed *meta-levels*) perform computation on the objects residing in the lower levels.

There is no need for the association between base-objects and meta-objects to be 1-to-1: several meta-objects may share a single referent, and a single meta-object may have multiple referents. The interface between adjacent levels in the reflective tower is usually termed a *meta-object protocol* (MOP). Albeit several distinct reflection models have been proposed in the literature (e.g., where meta-objects are coincident with classes, or instances of a special class *MetaObject*, and so on), such a distinction is not relevant for this discussion and will be omitted.

In all reflective models and MOPs, an essential concept is that of *reification*. In order to compute on the lower levels' computation, each level maintains a set of data structures representing (or, in reflection parlance, a *reification of*) such computation. Of course, the aspects of the lower levels' system that are reified depend on the reflective model (e.g., structure, state and behavior, communication). In any case, the data structure comprising a reification are *causally connected* to the aspect(s) of the system being reified; that is, any change to those aspects reflects in the reification, and vice versa. It is a duty of the reflective programming language framework to preserve the causal connection link between the levels (depending on the reflective model, this infrastructure may operate at compile- or at run-time): the designers and programmers of meta-objects are insulated from the details of how causal connection is achieved. Meta-objects can be programmed in exactly the same programming paradigm as conventional computation. It is in fact possible, and most usual, that all levels of the reflective tower be programmed in the same *programming language*. The fact that all the levels of the tower be implemented in a single language is qualified by some author as one of the characterizing features of reflection proper [7].

Another key feature of all reflective models is that of *transparency* [16]. In the context of reflection, this term is used to indicate that the objects in each level are completely *unaware* of the presence and workings of those in the levels above. In other words, each meta-level is added to the referent level without modifying the referent level itself. The virtual machine of the reflective language, in other words, manages causal connection link between a meta-level and its referent level in a way that is transparent *both* to the programmer of the meta-level and to the programmer of the referent level.

3.2 Reflection and non-functional properties

An application of reflection, supported by the feature of *transparency*, is the (non-intrusive) *evolution* of a system: the behavior or structure of the objects in a system can be modified, enriched, and/or substituted without the need to modify the original system's code. In principle, this may have interesting applications to the evolution of non-stopping systems or systems that are only available in black-box form.

Another well-known application, which is the one that will be considered in this paper, is that of adopting a reflective approach to separate functional and (possibly several distinct) non-functional features in the design of a system. In a typical approach, the base-level objects may be entrusted to meet the application's functional requirements, while meta-levels augment the base-level functionality ensuring non-functional properties (e.g., fault tolerance, persistence, distribution, and so on). Depending on the specific support provided by the reflective language virtual machine, the evolution of a system through the addition of a meta-level may require recompilation or maybe done dynamically. With reference to this partitioning of a system, in the following we will refer to the base-level objects as "functional objects" and to meta-level objects as "non-functional objects". While functional objects model entities in the real world (such as *bank_account*), non-functional objects model properties of functional objects (to reflect this, non-functional classes may have names that correspond to properties, e.g., *fault_tolerant_object*).

There are several reasons why a design could take advantage from such an approach. Of course, separation of concerns (in general, hence also in this case) enhances the system's modifiability. Depending on whether a required modification of the system involves functional or non-functional properties, functional objects alone or non-functional objects alone need to be modified. If a new security policy is adopted for the transactions related to the accounts in a bank, the (functional) class *bank_account* will be modified; if, say, a higher level of fault-tolerance is required, the (non-functional) class *fault_tolerant_object* will be changed. This approach also enhances reusability in two ways: first, the very same "functional" object (e.g., a *bank_account* object) can be reused with or without the additional properties implemented by its associated meta-objects, depending on context. Any additional feature of an object (e.g., fault-tolerance, the capability to migrate across platforms, persistence, and so on) has an associated overhead. In a reflective approach, all such features are not hardwired into the code of the object itself but implemented by separated meta-objects; whenever the additional features are not required, the corresponding meta-objects are simply not instantiated.

As a second form of reuse, many non-functional properties lend themselves to be implemented in a way that is essentially independent of the specific class of objects for which they are implemented. As an example, support for persistence is usually independent of the specific type of object being made persistent, as demonstrated by the adoption of "persistent" classes in Java and other mainstream OO programming languages. In our opinion, this is likely to hold for several typical non-functional properties; some examples are provided by the works on reflective approaches to fault-tolerance [1, 6], persistence [11], atomicity [18], and authentication [17, 15, 2]. Based on this fact, it is reasonable to expect that the same meta-object can be reused to attach the same non-functional property to different functional objects.

4 Reflective Object-Oriented Analysis

4.1 General Concepts

Two main points from the considerations of the previous section can be highlighted:

❶ the property of *transparency* of reflection allows for functional and non-functional concerns to be clearly separated in the design of a system, being respectively entrusted to the base-level and to the meta-levels;

❷ the concept of *reification*, and the transparent application of causal connection by the virtual machine of a reflective programming language, allows for meta-levels to be programmed in the same paradigm as the base-level.

Based on these two points we propose a novel approach to the treatment of non-functional properties in OO analysis. As we discussed in section 2, non-functional properties have no clear collocation in traditional OO analysis because they have no counterpart in the vocabulary of OO concepts. Fault-tolerance cannot be represented as a class, an object, an operation, an attribute, an association between classes, and so on. Nevertheless, in a reflective approach, those non-functional properties are represented by meta-objects. Meta-objects are objects themselves, and lend themselves to be described in OO terms. The transition to the "*meta*", in a sense, transforms something that is "about" an object (a property of an object) *into an object itself*; it "reifies" a property. As a consequence of this transformation, object properties themselves are absorbed into the scope of notations and meta-models for OO modeling and hence become natural subjects for OO analysis.

This can be further illustrated by the following parallelism. Just like the concept of reification allows for meta-levels (that implement *computation on computation*) to be programmed in the same language as used for the base-level (that implements *computation on the domain*), so it allows for the computation performed by the meta-level to be analyzed and modeled using the same concepts and techniques that are used to analyze and model the computation in the base level. When applied to base-level objects, these concepts model properties of the real-world entities that those objects model (e.g., operations model the dynamics of real-world objects, such as drawing money from a bank account). When applied to meta-level objects, the same concepts model properties of the *software objects that represent real-world entities within the system* (e.g., operations model the dynamics of software objects, such as their ability to be saved onto, or restored from, files).

4.2 Reflective Object-Oriented Analysis

We argue that the considerations made insofar suggest a novel approach to OO analysis, namely *reflective object-oriented analysis* (ROOA). In ROOA, the requirements of the system are partitioned, in the analysis phase, into concepts related to the domain and concepts related to the software system operating in that domain (i.e., into functional and non-functional). The concepts related to the software system may, themselves, be partitioned according to the properties they deal with (fault-tolerance, persistence, distribution, reliability, and so on). Observe that this partitioning is orthogonal to the traditional partitioning of the functional requirements of a system, namely that guided by OO concepts applied to the application domain. It is then natural to speak of additional "levels" of specification, which complement the traditional (functional) level. Note that we are not necessarily proposing a *method*; it is not our assumption, as for now, that this partitioning into levels be a step that must be taken "before" or "after" traditional analysis. Our perception is simply that such a partitioning should *complement* the traditional functional partitioning as assumed, explicitly or implicitly, by current OO methods. Each level of this partitioning can be analyzed using standard OO analysis techniques, and specified within a traditional OO meta-model (e.g., the UML meta-model). All the concepts from the standard OO vocabulary can be used when modeling non functional levels (class, inheritance, association, attribute, operation, and so on), albeit, as mentioned in the previous subsection, these levels are about properties of the system rather than the domain. This approach, *per se*, has the advantage of allowing for a more precise specification of non-functional requirements. To take full advantage of the approach, nevertheless, a reflective approach to *design* should also be applied, whereby the non-functional levels of analysis are mapped onto meta-levels (in the reflective sense) in the implemented system. In this case, the additional benefit of ROOA is that a smooth transition from analysis to design for non-functional requirements is feasible.

Let us consider the ATM example again. In the ROOA approach, the *withdraw* operation is specified, in the analysis, simply as a movement of money. Nevertheless, it is also specified (at another, nonfunctional level) that some properties, such as *atomicity* and *security*, hold for some actions in the system, including the *withdraw* operation. These properties are analyzed using standard OO analysis, and result in the definition of a set of classes describing them precisely (e.g., describing what an "atomic operation" is). In the transition to design, these properties are refined into meta-objects' classes charged with preserving the atomicity (security, reliability) of some of the operations of their referents. While a detailed discussion of how this could be done cannot be included here, the reader may refer to [2] for more information on a reflective approach to authentication.

The main objective of this work is that of suggesting that adopting reflective point of view may be useful to address the analysis and specification of non-functional requirements, and especially so when the resulting system is *a reflective system* separating non-functional from functional concerns via reflection. Nevertheless, we also believe that the general lines of ROOA apply, in general, to the design of *any* reflective system. To the best of our knowledge, few efforts have been made to propose extensions or adaptations of OO methods, methodologies, and processes to OO reflective systems. In our view, the best way to design a reflective system is that of considering it in a reflective perspective from the outset, i.e., from analysis. This means that the analysis phase should include a partitioning of the system's requirements into levels as that proposed in the previous section.

5 Related Work

Several authors within the Reflection field have considered the application of reflective techniques to address non-functional software requirements. Hürsch and Videira-Lopes [9] highlight the relevance of an approach that separates multiple concerns (including functionality as one specific concern, as opposed to other non-functional concerns) both at the conceptual and implementation level. They provide a tentative classification of the concerns that may be separated in general software systems, and encompass the major techniques that may be used to separate them, namely *meta-programming*, *pattern-oriented programming*, and *composition filters*. Their discussion is somewhat less specific than that provided by this paper, as their concept of separation of concerns is not necessarily achieved via the use of reflective techniques (i.e., meta-programming). Most of other related efforts propose *design approaches* (rather than analysis approaches) for structuring a software system in such a way that non-functional requirements are addressed by a system's meta-level(s) and thus cleanly separated from functional (base level) code. As we basically aim at supporting a smooth transition from analysis to design via reflection, reflective design approaches to the enforcement of non-functional properties provide us with some hints as to what the *result* of this process (that is, the resulting design) should look like. Stroud and Welch [17] discuss a reflective approach to the dynamic adaptation of security properties of software systems. In their approach, security properties are transparently added to an existing system (even available in black-box form) and easily tailored to any specific environment onto which the system is downloaded. Their paper explicitly tackles the issue of separating functional and non-functional requirements and reusing meta-objects implementing non-functional properties. Several authors addressed the transparent addition of fault-tolerance features to a software system via reflection, e.g. [1] (that applies channel reification to communication fault-tolerance) [14] (that employs reflective N-version programming and recovery blocks) [13] (that employs reflective server replication) and [10] (that employs reflective checkpointing in concurrent object-oriented systems). As mentioned above, other non-functional issues that were demonstrated to be effectively tackled via reflection include persistence [11], atomicity [18], and authentication [17, 15, 2].

Also related to the topic of this paper is our work on Architectural Reflection (AR) [3, 4]. In AR, a reflective approach is adopted for reifying the *software architecture* of a software system. While the definition of the architecture of a system is usually regarded as belonging to (early) design, there are cases where *requirements* on the architecture of a system should be considered from the outset (i.e., the need for integration with legacy systems, the need to reuse COTS components, and so on). Architectural properties of both the whole system (as actually addressed by AR) and of single objects may be addressed in a reflective approach like that suggested in this paper. Other authors have considered addressing architectural properties of objects using a meta-level; for example, [8] proposes a reflective object-oriented pattern for the separated definition of an object's behaviour.

6 Conclusions and Future Work

This paper is intended to suggest how traditional OO analysis could be extended in order to cope with non-functional requirements in a cleaner way than supported by current methods. It suggests that a reflective approach could be taken, whereby a system's specification is partitioned into levels (i.e., in way that is orthogonal to a "functional" partitioning), where the base level includes information on the domain, and the other levels include information on the system. This partitioning into levels could then be mapped easily onto a reflective architecture where requirements related to the system are refined into meta-objects that augment base-objects with non-functional properties. Also, applying this partitioning in the analysis phase may be useful, in general, when designing reflective systems. Of course, this is basically a "vision" paper and its intent is that of stimulating further work on the relationships between the concept of reflection and software methods (especially OO methods). We plan to continue this work by providing a more formal description of ROOA and considering in more detail how the approach may fit with mainstream OO notations, meta-models, methods, and methodologies. As a first step, we plan to study how the UML meta-model could be applied, in practice, to analyze the non-functional properties of a real system within the ROOA approach. While our ideas mainly stemmed from our interest to a reflective approach to design, it is most interesting to see if and how reflective concepts can actually be applied in analysis while preserving the valuable separation between "what" and "how" preached by software engineers. Should the outcome of this research be promising, we plan to build a tool to be integrated with a mainstream OO modeling tool such as Rational Rose and supporting ROOA. We also plan to investigate how ROOA can be integrated into mainstream object-oriented methods (e.g., Objectory).

Acknowledgements

The authors wish to thank Sergio Ruocco from DiSCO, University of Milano Bicocca, for his valuable contributions to the ideas presented in this paper.

References

1. Massimo Ancona, Walter Cazzola, Gabriella Dodero, and Vittoria Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th October 1995. ACM. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.
2. Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science 1603, pages 35–51. Springer-Verlag, July 1999.
3. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Bridging the Gap Between a Running System and its Architectural Specification. In *proceedings of 6th Reengineering Forum (REF'98)*, pages 12–1–12–6, Firenze, Italia, on 8th-11th March 1998. IEEE.
4. Walter Cazzola, Andrea Savigni, Andrea Sosio, and Francesco Tisato. Architectural Reflection: Concepts, Design, and Evaluation. Technical Report RI-DSI 234-99, DSI, University degli Studi di Milano, May 1999. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.
5. Pierre Cointe. MetaClasses are first class objects: the ObjVLisp model. In *Proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, Orlando, Florida, October 1987. ACM.
6. Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert J. Stroud, and Zhixue Wu. Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming. In *Proceedings of FTCS-25 "Silver Jubilee"*, Pasadena, CA USA, June 1995. IEEE.
7. Jacques Ferber. Computational Reflection in Class Based Object Oriented Languages. In *Proceedings of 4th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'89)*, volume 24 of *Sigplan Notices*, pages 317–326. ACM, October 1989.
8. Luciane Lamour Ferreira and Cecília M. F. Rubira. The Reflective State Pattern. In Steve Berczuk and Joe Yoder, editors, *Proceedings of the Pattern Languages of Program Design*, TR #WUCS-98-25, Monticello, Illinois - USA, August 1998.
9. Walter Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.
10. Mangesh Kasbekar, Chandramouli Narayanan, and Chita R. Das. Using Reflection for Checkpointing Concurrent Object Oriented Programs. In Shigeu Chiba and Jean-Charles Fabre, editors, *Proceedings of the OOPSLA Workshop on Reflection Programming in C++ and Java*, October 1998.
11. Arthur H. Lee and Joseph L. Zachary. Using Meta Programming to Add Persistence to CLOS. In *International Conference on Computer Languages*, Los Alamitos, California, 1994. IEEE.
12. Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, volume 22 of *Sigplan Notices*, pages 147–156. ACM, October 1987.

13. Juan-Carlos Ruiz-Garcia Marc-Olivier Killijian, Jean-Charles Fabre and Shigeru Chiba. A Metaobject Protocol for Fault-Tolerant CORBA Applications. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 127–134, 1998.
14. Brian Randell. System Structure for Software Fault Tolerant. *IEEE Transaction on Software Engineering*, SE-1(2):220–232, June 1975.
15. Thomas Riechmann and Jürgen Kleinöder. Meta-Objects for Access Control: Role-Based Principals. In Colin Boyd and Ed Dawson, editors, *Lecture Notes in Computer Science*, number 1438 in Proceedings of 3rd Australasian Conference on Information Security and Privacy (ACISP'98), pages 296–307, Brisbane, Australia, July 1998. Springer-Verlag.
16. Robert J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating System Review*, 22:99–103, April 1992.
17. Robert J. Stroud and Ian Welch. Dynamic Adaptation of the Security Properties of Application and Components. In *proceedings of ECOOP Workshop on Distributed Object Security (EWDOS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), pages 41–46, Brussels, Belgium, July 1998. Unité de Recherche INRIA Rhǒne-Alpes.
18. Robert J. Stroud and Zhixue Wu. Using Meta-Object Protocol to Implement Atomic Data Types. In Walter Olthoff, editor, *Proceedings of the 9th Conference on Object-Oriented Programming (ECOOP'95)*, LNCS 952, pages 168–189, Aarhus, Denmark, August 1995. Springer-Verlag.