DPS: An Architectural Style for Development of Secure Software *

Pascal Fenkam, Harald Gall, Mehdi Jazayeri, Christopher Kruegel

Technical University of Vienna, Distributed Systems Group
A-1040 Vienna, Argentinierstrasse 8/184-1
{p.fenkam, h.gall, m.jazayeri,c.kruegel }@infosys.tuwien.ac.at

Abstract. Many claim that software systems must be designed for security. This, however, is far from being an easy task, especially for complex systems. We believe that this difficulty can be alleviated by a set of —preferably rigorous— principles. We propose an architectural style, the *Dual Protection Style (DPS)*, for constructing secure software. This style results from our experience in designing and implementing a distributed, multi-user, medium sized application. We present the applicability and effectiveness of our *DPS* style on the basis of a case study of a distributed software platform for virtual and mobile team collaboration called MOTION. We further elaborate on the description of this architectural style, its formalization and the formal verification of some of its properties.

Keywords: Authorization and Access Control, Security Engineering, Formal Methods, Software Architecture, Architectural Style, Alloy.

1 Introduction

The dependability of software systems is becoming a foremost concern for the whole society. This is exacerbated by the increasing pervasive use of software. These are no longer simply used in a standalone fashion, but have become highly distributed and embedded in many of our working tools. The pervasiveness of software systems is accompanied by a variety of security risks. For instance, it is not necessary anymore to be physically present in front of a computer to attack its security engine: thanks to the Internet revolution. This frightening reality has (or should have) consequences on the way we build software. It is increasingly argued that software must be designed for security right from the beginning instead of security being added as an afterthought [3, 7, 9, 17, 18, 22]. Numerous examples are cited in the literature where applications failed to be "upgraded" from non-secure applications to secure applications.

If designing for security seems to be an acceptable and reasonable recommendation, the challenge, however, resides in how to achieve this goal. This is particularly difficult since the concept of security itself is differently understood. Though security is difficult to define, there are well understood security criteria for distributed applications: identification, authentication, authorization and access control, integrity, auditing, etc. At least for these well understood notions, one would expect some principles guiding their introduction into distributed applications. To our knowledge, little work has been done on illustrating how a secure application can be constructed.

In this paper, we propose an architectural style for constructing access controlled applications. This style is a collection of rules that constrain the topology and the behavior of the components of an application that needs to support access control. Our architectural style prescribes ways of controlling access to resources made available by applications. It is based on two principles, user access control

* This work is supported by the European Commission in the Framework of the IST Program, Key Action II on New Methods of Work and eCommerce. Project number: IST-1999-11400 MOTION (MObile Teamwork Infrastructure for Organizations Networking).

and code access control. User access control is the process of verifying that end-customers only perform actions they are allowed to. Code access control on the other hand ensures that applications only invoke methods they are allowed to. Following this architectural style, only one method in each trace of the application must perform user access control. The paper also presents the formal specification of this architectural style in *Alloy* [11], a formal notation for micromodels of software. The main purpose of this formal specification is to provide a strong foundation to the style, and a precise definition of its invariants.

The remainder of the paper is organized as follows. Section 2 is concerned with related work: some principles proposed by various researchers for constructing secure applications are presented. Section 3 gives an overview of the MOTION platform that inspired the development of this architectural style. This case study also helps to better illustrate the problem solved by our work. We present an informal description of the architectural style in Section 4 and its formal description in Section 5. Section 6 concludes the paper.

2 Related Work

Building secure software is starting to draw well-deserved attention. Up to now, most computer security articles published in industry magazines focussed on fixing security problems [7]. Among the few that propose principles for building secure software systems, we consider [10, 18, 23, 25, 27, 28] particularly interesting.

In [18], McGraw and Felten propose twelve rules for writing secure Java programs: 1) programs should not depend on initialization, 2) Use of visibility restrictors, 3) everything should be made final unless there is a good reason not to, 4) programs should not depend on package scope, 5) programmers should avoid using inner classes, 6) programmers should avoid signing code, 7) if signing code is a must, all should be put in one archive file, 8) classes should be made unclonable, 9) classes must be unserializable, 10) classes must be undeserializable, 11) classes should not be compared by name, 12) secrets should not be stored in code.

Similar recommendations are given in [23]. Besides the controversial nature of some of these guidelines, they are oriented towards Java programs. Thus, they can be viewed as low level principles. Referring to the idea that security considerations must be incorporated throughout the entire product lifecycle [22], we argue that there is a need for more high level principles that can be applied at the architecture and design levels.

Viega and McGraw [27] propose a complement to these principles. Ten recommendations are given to be kept in mind throughout the design and the implementation of a secure software system: 1) secure the weakest link, 2) practice defense in depth, 3) fail securely, 4) follow the principle of least privilege, 5) compartmentalize, 6) keep it simple, 7) promote privacy, 8) hiding secrets is hard, 9) be reluctant to trust, 10) use of community resources.

Besides these principles, the Open Group [25] is currently writing a book on security patterns. A first draft of this book is available on the Internet. For the moment, 32 patterns are indexed and 15 of them are documented. These patterns are divided into 5 categories: entity patterns, structural patterns, interaction patterns, behavioral patterns, and available system patterns. For the flavor of these patterns, we give an overview of four of them.

The *Protected System* pattern also called *Reference Monitor* pattern recommends software developers to delegate all access control checks to a guard which enforces security policies. The *Policy Enforcement Point* pattern complements the first by recommending to separate/isolate the code of the reference monitor from the remainder of the application. The *Secure Communication* pattern ensures that security policies of differents take-holders still be satisfied when they come to communicate with each other. The *Recoverable Component* pattern requires a system to be structured so that its state can be recovered if the system fails.

An important fact to notice is that although developed separately, most principles provided by the open group are refinements of those proposed by Viega and McGraw in [18]. For instance, the *Checkpointed System* pattern and the *Recoverable Component* obviously contribute to ensure the *Fail Securely* principle. The *DPS* style gives a concrete guideline with which to achieve these principles. After the description of this architectural style in Section 4, we show how it helps in securing the weakest link, keeping things simple, increasing usability, and protecting against malicious code. The next section gives an overview of the MOTION project that inspired the development of the *DPS* architectural style.

3 The MOTION Project

MOTION (MObile Teamwork Infrastructure for Organizations Networking) is a platform we designed and prototyped in the MOTION European project [14, 21]. This platform addresses the needs of two well known organizations. The first is a manufacturer of mobile phones and the second is a producer of white goods (e.g. refrigerators, whashing machines) . The platform has a service architecture that supports mobile teamworking by taking into account different connectivity modes of users, provides access support for various devices, supports distributed search of users and artifacts, offers user management facilities in a way where users can be grouped in communities. The MOTION platform also contains an access control mechanism.

3.1 The MOTION Architecture

The MOTION system was constructed by assembling different components: PeerWare [19], DUMAS (Dynamic User Management and Access Control System) [6], an XQL engine [8], an XML parser, a repository (comparable to a file system), an artifact manager (comparable to a shell that provides primitives for accessing the file system), etc. Some of these components are commercial off-the shelf while other were built by the project team. A layered view of the MOTION platform is presented in Figure 1.

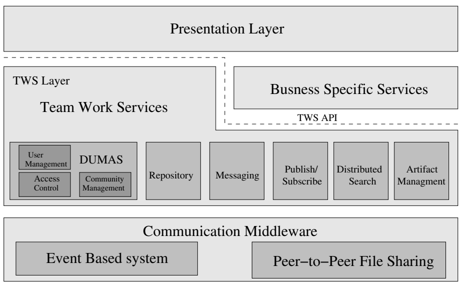

Fig. 1. Overview of the MOTION Architecture

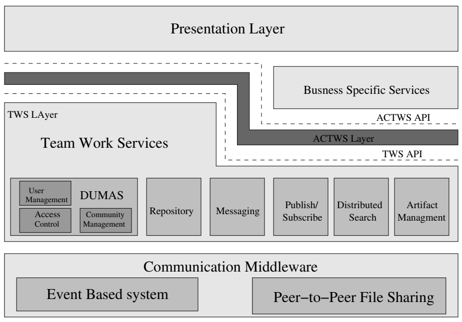

Fig. 2. The Access Controlled MOTION

The bottom layer of the architecture provides the communication infrastructure and is realized by PeerWare [19]. The services made available by this component include: peer-to-peer file sharing, an event based system, and mobile code infrastructure.

```
1       public void addMember(UserID userId,
2                             CommunityID target_community)
3             throws NoSuchEltException, MissingPermissionException,
4                DuplicatedDefinitionException {
5             UserID actor= motionmanager.getCurrentUserID();
6             Permission permission=new Permission("add-member");
7             authorization.checkPermission(actor, permission,
8                                           target_community );
9
10            motionmanager.getDumasController().getUserManager().
11                linkUser(actor, userId,target_community);
12
13            Element elt=temp.createElement("Action");
14            elt.setAttribute("classname","Community" );
15            elt.setAttribute("methodname","addMember" );
16            elt.setAttribute("argument0",actor.getID() );
17            elt.setAttribute("subjecttype","user");
18            elt.setAttribute("actor",actor.getID());
19            elt.setAttribute("target",target_community.getID());
20
21            TUVEvent evt= new TUVEvent(elt);
22            twsps.publish(evt,peer.getPublicationDocument());
23
24      }
```

Listing 1: Adding a Member to a Community

On top of this layer, we constructed the teamwork services layer (TWS layer). This teamwork services layer is composed of DUMAS, a repository, a messaging system, a user oriented publish-subscribe system (TWS P/S), a component for distributed search, and an artifact management component. DUMAS groups user management, community management, and access control functionalities. The repository is a component responsible for storing different MOTION data, including files, meta-data, and access control lists. All methods in the TWS layer are security sensitive and therefore access controlled.

The top-most layer of the MOTION platform consists of the presentation layer and business specific services. This layer includes applications matching the specific requirements of end-customers. They completely rely upon the TWS layer.

3.2 Component Integration

In this section, we discuss the integration of the components listed above. The purpose of this section is to illustrate (1) the need for the DPS style, and (2) the benefits we would gain if we had used this style right from the beginning of the development process. We focus on the integration of the TWS layer components with the access control module. This integration essentially consists of inserting the instructions for access control in each public method of the TWS layer. An example is presented in Listing 1. The method `addMember` is shown. The purpose of this method is to add a user to a community. The access control check is performed at line 7. The lines 3 to 6 prepare the arguments for invoking the operation `checkPermission` performing the permission check. This integration had to be performed for approximatively 110 methods made available by the TWS layer. Initially, 100 person-hours were foreseen for this task. At the end we needed about 320 person-hours, more than 200 % time over-run! What went wrong?

The operation `addMember` presented in Listing 1 is performed by requesting the services of three different components: (1) the DUMAS user management module (line 10), (2) the TWS publish- subscribe facility (line 22), and (3) the repository. The repository is not directly invoked by `addMember`, instead `addMember` invokes `linkUser` of DUMAS that further invokes the repository. The invocation of the TWS publish operation is performed for notifying other users and components. Each public method of the TWS layer performs access control checks. This is because most of these methods are also directly invoked by the presentation layer or business specific services. The implication is that for invoking the method `addMember`, three different access rights are required:

- the access right `add-member` required by the operation `addMember`. This access right ensures that the user invoking the operation has the permission to insert a user into the target community.
- the access right `write` ensures that the user has the permission to write to the repository.
- the access right `publish` is checked by the TWS publish/subscribe for ensuring that the user has the permission to publish the specified event.

We face an embarrassing problem! This multitude of permission-checks does not protect the TWS layer, but makes it unusable. For performing a single operation, too many different access rights are required from the user. We noticed this because the access control component was developed in tandem with a graphical user interface. This user interface allowed users to give us direct feedback. This problem is exacerbated by the fact that these access rights are not directly semantically related to the operation `addMember`. For a customer, there is no obvious reason to require access right for writing to a repository while adding a user to a community. This might even appear as a bug in the system. Besides this usability issue, a second problem, that such an integration of access control into applications faces is the issue of administration. The more complicated an access control system is, the more difficult is its administration and the more likely are security breaches into the system.

To solve this problem, we ensured that users are not asked more than one access right during the execution of a single task. This is achieved by requiring that in each trace of the application, user access rights are checked only once. To improve the TWS layer such that it takes this result into consideration, we had to remove access control checks from all MOTION components first. The next step consisted of building a tiny layer on top of the TWS layer (see Figure 2), that separates the non-access controlled TWS layer from the presentation layer and business specific services. This tiny layer is called access controlled teamwork services layer (ACTWS layer). All methods of the ACTWS layer are access controlled while no method in the TWS layer is access controlled. Further, no operation of the ACTWS layer is allowed to invoke an operation of this layer, be it recursively or not. This would obviously lead to the initial situation.

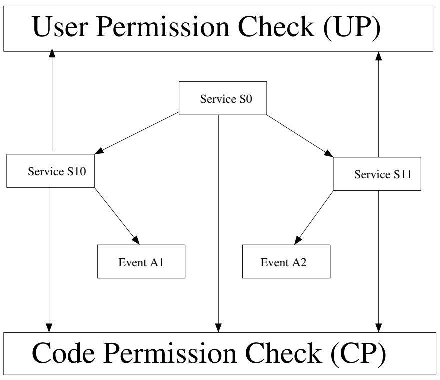

Fig. 3. The Dual Protection Style

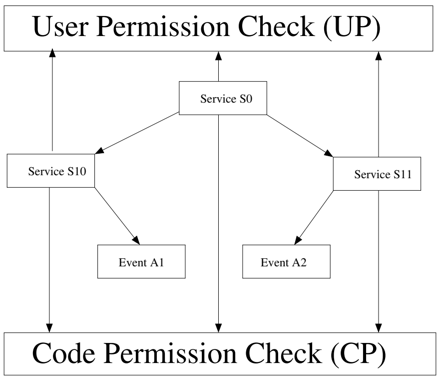

Fig. 4. Example of DPS Violation

While the approach described above solves the problem of multiple access rights for a single task, it raises another issue. The layers underlying the ACTWS layer are completely open, without any access control check, thus, they are vulnerable to various attacks. An important kind of attacks is malicious code. First, this attack seems to be à la mode nowadays. Second, it is particularly relevant to the MOTION platform since the underlying middleware uses mobile code for relocation facilities [19]. The solution to this second challenge is to protect components by means of code access control. This is, for each method in the MOTION system, there exists a list of components (or a list of methods, depending on the granularity) allowed to invoke it. Thus, if some malicious code wants for instance to modify the access control list stored in the repository, the operation fails since this code is not allowed to interact with the repository. The principle is similar to code access control provided in the Java environment [15].

Given these two techniques, we are able to efficiently protect the MOTION platform. We strongly believe that the experience gained in securing this platform can be useful to other software developers. We therefore propose to capture its quintessential properties through an architectural style that abstracts away from MOTION specific details.

4 The DPS Architectural Style

In this section, we give an informal description of the *Dual Protection Style*. The goal of this style is not to guarantee security, but to simplify the construction of access controlled applications. The *DPS* style defines components, connectors and constraints for controlling the access to services provided by an application.

Vocabulary

In its basic version, the *DPS* recognizes three types of components: a user access control component, a code access control component, and the set of components forming a system and interacting with the user and code access control components. In case of MOTION, such components are, for instance, the repository, the artifact manager, or the community manager.

A component is called *protectee* of a user/code access control component if one of its services interacts with this user/code access control component. This user/code access control component is called a *protector* of the protected component. Further, a component that makes available a service interacting with a user access control component is said to be *user access controlled*. If the component rather interacts with a code access controller, it is said to be *code access controlled*.

For instance, all methods of the TWS layer are code access controlled while all methods of the ACTWS layer are user access controlled.

The connectors between *protectees* and *protectors* are called *access control checks* or *permission checks*. There exist two kinds of *permission check: user permission check* and *code permission check*. A *user permission check* is initiated by a *protectee* towards a user access control component. This is a request to the user access control component to check the authorization of a user to access an object. Similarly, a *code permission check* is a request to a code access controller to decide on the authorization of another application to make use of the requested service.

Invariants

The *DPS* style constrains the behaviors of *protectors* and *protectees* by means of two invariants. The first invariant is best explained in terms of CSP traces [2]. A trace is a sequence of events that a process (or a service) allows. An event is an atomic action, i.e. it cannot be decomposed. In this respect, we consider permission checks as events. A permission check cannot be decomposed into other events. The implication is that any process performing permission checks is not atomic. We use the terms service and process equivalently. An application satisfies the *DPS* style if each of its processes satisfies the *DPS* invariants. Such an application will be said to be *DPS* protected.

The first invariant of the *DPS* style ensures that the number of occurrences of user permission checks in each process is equal to 1. The number of occurrences of an event, say a, in a trace T is denoted by $T \downarrow a$.

An example of a system S, consisting of the services S_0, S_{10}, S_{11} is illustrated in Figure 3 together with its call graph. Its processes are defined as $traces(S_{11}) = \langle UP.CP.A_2 \rangle$, $traces(S_{10}) = \langle UP.CP.A_1 \rangle$, $traces(S_0) = \langle CP.UP.CP.A_2, CP.UP.CP.A_1 \rangle$. *UP* and *CP* are user and code permission checks respectively and A_1 and A_2 are actions other than permission check.

This system satisfies the first invariant of the *DPS* style. In fact, $UP.CP.A_1 \downarrow UP = 1$, $UP.CP.A_2 \downarrow UP = 1$, $CP.UP.CP.A_1 \downarrow UP = 1$ and $CP.UP.CP.A_2 \downarrow UP = 1$.

Informally, this invariant prescribes that user permissions should be checked only once for each task that a user performs. It does not prevent methods from performing various permission checks, but ensures that all permission checks must be performed in an exclusive manner so that only one of them can be executed each time.

For a program that has no permission checks in **if-else** blocks, the invariant can be expressed in terms of its call graph. In this case, there should be one method performing user permission check in each path of the call graph. An example is depicted in Figure 3. On each of the paths of this call grapth, there exists exactly one service that initiates a user permission check.

In general, this invariant cannot be expressed in terms of a call graph since a method whose body includes the instruction
`if(condition) user-permission-check-1 else user-permission-check-2` satisfies the first invariant of the *DPS* style but is said to contain two permission checks (following the definition of call graphs [26]).

The second invariant of the *DPS* style ensures that each service performs at least one code permission check. In other terms, all methods must be *protectee* of some code access controller. Considering the example of Figure 3, all services S_0, S_{11}, S_{10} perform code permission checks. Unlike the first invariant, the second invariant is easily expressed in terms of methods instead of traces.

Note that the way how the access controller decides on whether users or code have permissions depends on each architecture (thus instance of the style).

The *DPS* style supposes that the identities of users and code can be verified. That is, there exists a reliable mechanism for determining which user/service is requesting a service. Though there exist different mechanisms for authenticating users, this is not the case for code. Considering two services hosted by two different computers and communicating by means of a handshake protocol it is difficult for each of them to verify the identity of the application running at the other end. This problem, however, does not exist when the components are co-located and one is accessing the other through a direct method call. In Java, for instance, the stack inspection algorithm provides the facility of determining which methods are currently being executed.

Specializations and Instantiations

The *Dual Protection Style* can be specialized in several interesting ways.

We say that a permission check connector is open for a resource S if the connector authorizes any access to this resource. We say that a permission connector is open if it is open for all resources.

Depending on which connectors are open in a system, the following specializations of the *Dual Protection Style* can be distinguished. In the first case, the *user permission connector* is open, in the second case the *code permission connector* is open, and in the third case both connectors are open.

Strictly speaky, the *DPS* style is violated when one of the above invariants is not satisfied. For instance, a system that has no code access controller can theoretically be said to violate the *DPS* style. On the other hand, this system behaves exactly like a system with the same services but with an open *code permission connector*. A system that theoretically doesn't satisfy the *DPS* style but behaves in practice exactly like a specialization of the *DPS* style is said to have an architecture that is of a degenerate *DPS* style. We do not consider such systems as violation of the *DPS* style.

This fuzzy boundary of architectural styles is not specific to the *DPS* style. It has also been recognized for well known styles such as Pipe-and-Filters. Though *batch sequential* is an architectural style in its own right, it also results from a pipeline (a pipeline is a specialization of pipes-and-filters) where each filter processes all of its input data as a single entity. Another example is provided by the peer-to-peer architectural style that behaves as client-server in some configurations. As recommended by Shaw and Garlan [16], the fact that boundaries of styles overlap should not deter us from identifying the main features of a style with its central example of use. Thus, in practice, we will not consider systems that practically behave as specializations of the *DPS* style as violation of this style.

The only case that intriguingly violates the *DPS* style is when there exists a trace of the application where the number of occurrences of the user permission checks is greater than 1. Such a case is shown for instance in Figure 4. The branch of the call graph composed of the invocation of S_0, S_{01} contains two user permission checks. In terms of traces, we have
$traces(S_0) = \langle CP.UP.CP.UP.A_2, CP.UP.CP.A_1 \rangle$ and $CP.UP.CP.UP.A_2 \downarrow UP = 2$. The `mv` command of the UNIX/LINUX operating system is an example of an application that does not satisfy the *DPS* style (although an equivalent command can be implemented that satisfies DPS). For moving a file to a directory, one needs to have the permission to read that file and the permission to write to the target directory . The command `less` on the other hand satisfies the *DPS* style. These commands are applications that are constructed using the system calls of the operating system such as `sys_read` and `sys_write` [13]. A question that can be raised is if the UNIX/LINUX operating system itself satisfies the *DPS* style . We are currently investigating this issue.

Code access control is not specific to Java. The concept of domain type enforcement [1] originally proposed for Unix ensures that code modules are permitted to access only a controlled set of resources (including other modules).

Advantages

Our style provides a number of advantages. Most of them are oriented towards facilitating the development of access controlled applications. Therefore, we concentrate on these features first and not on the security capabilities of our style. The *DPS* style proposes a clear architecture for easily constructing usable, and secure applications. We are not aware of a similar architectural style for building secure systems.

Starting with the *DPS* style, the developer knows from the beginning of the development process exactly what actions are to be carried out. He can decide exactly where to perform security checks in his components, decide on which protectors to use, and what the implications are in terms of dependability, architecture, and maintenance. In general such an architectural style forces the software developer to take security into consideration very early in the design of its architecture.

Applications based on the *DPS* style are easier to administer. In fact, the administrator knows that for each task, there is at most one access right that must be owned. This is obviously much simpler than administrating an application where tasks require many different permissions. Moreover, the behavior of applications where operations require many access rights might sometimes be unexpected. Considering the case of `addMember` presented above, it is difficult to assign to a user the permission to add users to a community without assigning him the permission to publish events related to the profile of this user such as “user is online”, or “user is expert”. An undesirable side effect! This side effect takes a particular significance in information flow control. We claim that a system can more easily be proven to satisfy an information flow constraint when its architecture is an instance of the *DPS* style than when it isn’t.

The *DPS* style promotes securing the weakest link by minimizing the dependency on customers. The weakest link in an application designates a component, an algorithm, a protocol, or anything else that is likely to be weakly secure. Attackers will obviously look for the weakest link of an application. Social engineering was shown to be one of the most effective means to bypass security restrictions. Security engineers often make assumptions that, though reasonable, do not hold in practice (an example is the famous question asked by most browsers: “do you trust this application?”). It is therefore very probable that users will have difficulties in using complex security mechanisms. The consequence is that either they ignore them, thus being completely vulnerable, or they use them with difficulties. Thus, the less a security mechanism relies on customers, the less it is vulnerable to social engineering efforts and therefore the lower is the probability of being attacked through this link. By reducing the number of access rights required by a user, the developer reduces the difficulties that users might encounter.

In systems where access control is performed at different levels, thus not satisfying the *DPS* style, the software/security engineers will spend significant time in investigating which permissions/access rights must be checked at each stage and in each component. The assumption is thus made that programmers have the necessary background in access control models. This is unfortunately not always the case, as security engineering doesn’t even belong in most software engineering curriculums. These difficulties are exacerbated if we take into consideration the iterative nature of the software development process that probably also implies iteration on the access control model(s) and high degree of interference between the software engineers and the security engineers. The *DPS* style allows reducing the interaction between security engineers and software engineers. It finally frees software engineers from dealing with access control by delegating this task to a few people who know exactly where there is a need for access control check in the application. This can be done by constructing a layer as we did in the MOTION case study. Though software engineers need to integrate code permission checks in components, this is a minor issue since there is nothing more than invoking a method that another method can do. The interface to the access control component is thus simple and can easily be specified by security engineers and the API made available to software engineers.

The *DPS* style promotes protecting applications against malicious code. It is not enough for an application A to be started by a user to access a resource r made available by a DPS protected application C. The application A must be explicitly allowed by the code access controller of C to invoke the method leading to r.

The *DPS* style achieves a clear decoupling between permission check connectors; one can check for user permissions without checking for code permissions and vice versa. This clear separation is missing in the Java security architecture [15] where the two connectors are strongly coupled. The separation between these connectors obviously allows security engineers to finer control security mechanisms to match their needs.

Shortcomings

The *DPS* style relies on the identity of client components. This leads to a subtle limitation in the application of the *DPS* style in distributed environments. This limitation occurs when it is not possible to verify the identity of the requesting service. Let us consider for instance an architecture based on services S_{01}, S_{02}, and S_1 where S_{01} and S_{02} both request S_1. All communication between these services is performed by means of handshake protocols. This means that these services have no methods for verifying the identity of the code with which they communicate. Following the *DPS* style, user permission checks must be performed either by S_{01} and S_{02} or by S_1. In the first case, however, S_1 is totally vulnerable to attacks since there is no way to establish the authenticity of its callers and thus to perform code permission checks.

In the second case, user permission checks must be done by S_1 (following *DPS*). This however is not always feasible. S_1 might be a commercial service made available by a third party while S_{01}, and S_{02} are independent services constructed upon S_1. The recommendation of the *DPS* style would thus mean that S_1 must manage a user access control component and policy for S_{01} and S_{02} which is not practicable. A concrete example of this scenario is where S_1 is a search engine such as Google and S_{01} is a MOTION service. There is no reason why Google should perform access control for the MOTION platform.

We propose two solutions to this limitation. The first solution consists of relaxing the second invariant of the *DPS* style. This relaxation consists of identifying remote applications through the name of the computer hosting them (possibly combined with some other authentication information). The second solution relies on using middleware such as RMI or CORBA. They give applications the possibility to question the identity of a caller. In both solutions, the application of the *DPS* style is risky. The risk in the first solution is obviously bigger than in the second solution. In the first solution, some malicious code can still attack a remote service if it is located on a host authorized to interact with this service. In the second case, one needs to fake the implementation of RMI/CORBA that is used to communicate with the remote service. The fake implementation will thus masquerade the identity of the caller. Obviously one is more protected in the second case, whose dependability can be improved by using further authentication information.

A second shortcoming that can be attributed to the *DPS* style is the lack of defense in depth. The idea behind defense in depth is to provide diverse defensive strategies, so that if one of them fails the other can still prevent a full breach. If we consider the example shown in Figure 3, if a security breach succeeds in S_0, by means of buffer overflow for instance, the attacker gains control of the system. He can perform everything that the services that S_0 is normally allowed to perform. There exists a solution to this limitation that even produces the same results as the *DPS* style. This solution consists of reformulating the first invariant of the *DPS*. This reformulation proposes not to limit the number of user permission checks in traces to one, but requires that all user permission checks in a trace be equivalent. We say that two user permission checks are equivalent if the related user access controllers return the same answers given similar input. We, however, prefer the actual version of the invariant since it is much simpler and intuitive enough to be understood and successfully applied by the average programmer.

5 An Alloy Specification of the DPS Style

In this section, we present a formal specification of the *DPS* architectural style and verification of some of its properties. As recognized in [12], if it is proven that an architectural style verifies some properties, there is no need to show that its instances verify these properties as well. Formally specifying architectural styles is therefore worthwhile since the cost of specifying and analyzing a style is amortized across instances. Furthermore, a style specification abstracts from many of the details of the instances allowing a style analysis to focus more easily on fundamental properties that might be obscured in the analysis of an instance. This formal specification helped us deepen our understanding of the style. It is only after a formal specification of this style that we could give a clear and simple formulation of the invariants. The DPS style is specified in the Alloy specification language. We give a brief overview of this language below. The different *Alloy* constructs used in the specification are explained when they are used.

5.1 The Alloy Specification Language

The *Alloy* specification language was designed by Daniel Jackson [11]. It is a first order language that can be viewed as a subset of Z [4]. *Alloy* is a declarative language similar to the formal specification languages Z [4], and VDM [20]. Unlike these languages, *Alloy* is automatically analyzable in the style of model checking. Models described using this specification language are called micromodels. They are intended to be tiny models that focus on the risky parts of a system.

It might be argued that a model should intrinsically abstract from details and thus be an order of magnitude smaller than the system it models. This observation holds! From our experience [5], however, the kind of verification mechanism available for a specification language plays an important role. Considering the example of VDM, the automatic verification facility available up to now is testing (facility provided by the IFAD VDM Tools [24]). To be amenable to testing, however, a VDM-SL specification has to be interpretable and thus written in an executable subset of VDM-SL. Thus, the designer is distracted from the goal of producing a tiny model to the goal of producing an interpretable model. Though interpretable models are an order of magnitude smaller than the systems they model, they are also an order of magnitude bigger than non-interpretable models. With its automatic analyzer, the *Alloy* specification language allows the designer to indeed concentrate on abstracting from details and thus, on producing tiny models that are easy to understand. Just as important, the analyzer of the *Alloy* specification language gives the designer the kind of benefit of immediate feedback that the execution brings.

5.2 The DPS Style in Alloy

As the reader must have noticed, the two invariants of the *DPS* styles are based on two different views of applications. The first is based on CSP traces and the second is based on a methods view.

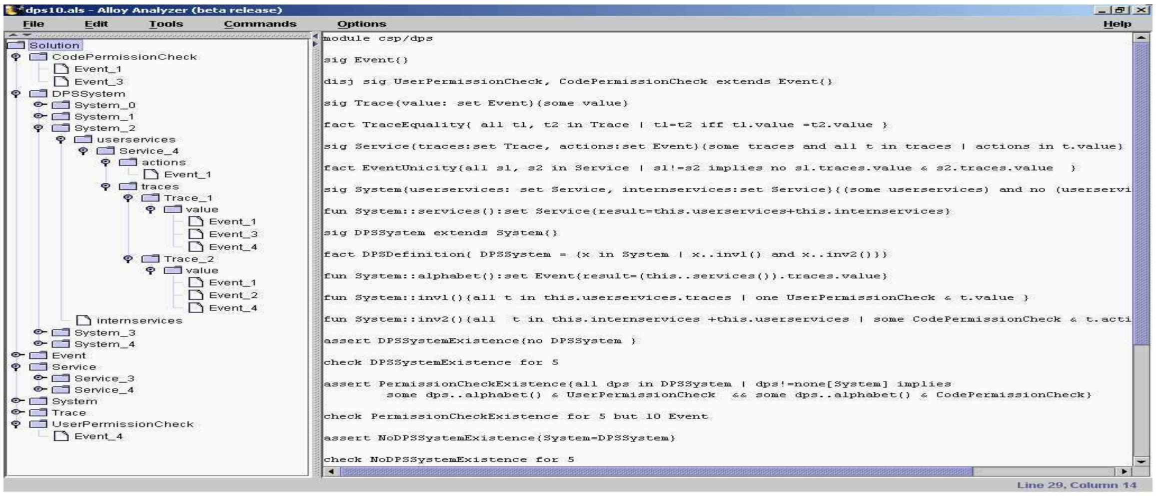

Fig. 5. An Automatic Instantiation of the DPS Style

```
1   module csp/dps
2
3   sig Event{}
4
5   disj sig UserPermissionCheck, CodePermissionCheck extends Event{}
6
7   sig Trace{value: set Event}{some value}
8
9   fact TraceEquality{ all t1, t2 in Trace | t1=t2 iff t1.value =t2.value }
10
11  sig Service{traces:set Trace, actions:set Event}{some traces and all t in traces | actions in t.value}
12
13  sig System{userservices: set Service, internalservices:set Service}{(some userservices+internalservices) and
14          no (userservices & internalservices) }
15
16  fun System::services():set Service{result=this.userservices+this.internalservices}
17
18  sig DPSSystem extends System{}
19
20  fact DPSDefinition{ DPSSystem = {x in System | x..inv1() and x..inv2()}}
21
22  fun System::alphabet():set Event{result=(this..services()).traces.value}
23
24  fun System::inv1(){all t in this.userservices.traces | one UserPermissionCheck & t.value }
25
26  fun System::inv2(){all  t in this.internalservices +this.userservices | some CodePermissionCheck & t.actions}
27
28  assert DPSSystemExistence{no DPSSystem }
29
30  check DPSSystemExistence for 5
31
32  assert PermissionCheckExistence{all dps in DPSSystem | dps!=none[System] implies
33          some dps..alphabet() & UserPermissionCheck  && some dps..alphabet() & CodePermissionCheck}
34
35  check PermissionCheckExistence for 5 but 10 Event
36
37  assert NonDPSSystemExistence{System=DPSSystem}
38
39  check NonDPSSystemExistence for 5
40
41  fun layer1():set Service{result={x in Service | (some UserPermissionCheck & x.traces.value) and
42          (no UserPermissionCheck & x.actions)}}
43
44  fun layer2():set Service{result={x in Service | some UserPermissionCheck & x.actions}}
45
46  fun layer3():set Service {result={x in Service | no UserPermissionCheck & x.traces.value}}
47
48  assert LayersDisjonction{ (no layer1() & layer2()) and (no layer2() & layer3()) and (no layer1() & layer3())}
49
50  check LayersDisjonction for 5
51
52  assert DPSLayeredView{ DPSSystem..services() in layer1() +layer2()+layer3()}
53
54  check DPSLayeredView for 5
```

Listing 2: The Alloy Specification of the DPS Style

The following specification is based on these two views. In this specification, we restrict the notion of trace to the alphabet of this trace. Thus, considering for instance the trace $\langle a, b, c, d\rangle$ we denote it as $\{a, b, c, d\}$. This simplification has no influence on our invariants since they only constrain the number of occurrences of some actions, but not their order.

The line 1 of the specification defines the name of the module including the style. The line 3 declares the type `Event`. An event is decomposable. Two kinds of events are of particular interest, `CodePermissionCheck` and `UserPermissionCheck` declared at line 5. These two kinds of events are disjoint (denoted with `disj`), no event can simultaneously be user permission check and code permission check. A trace is defined by a set of events (line 7). Empty traces are explicitly excluded from our specification. The fact `TraceEquality` ensures that two traces are equal if and only if the have the same alphabet. This is not true in general (see [2]). We define a service using two different views, a method view and a trace view (line 9).

In the first view, a service simply consists of a set of traces. In the second view, a service is defined as a service that further invokes other services and actions. The set of services a service invokes is not of interest to us but the set of actions it invokes is. The set of actions invoked by a service must be a subset of each of its traces.

We further define a system as composed of user services and internal services. User services are services that are avaialble to users of the applications. Internal services are services that are not directly accessible to users but probably to other services. Systems that define service are not of interest to us (line 15). Besides this restriction, we require that no service be simustaneouly user service and internal service. A *DPSSystem* is a kind of system (line 22) whose behavior is constrained by invariants *Invariant 1*, and *Invariant 2* (line 24). Invariant 1 (line 25) requires that any user service of a *DPSSystem* performs exactly one user permission check —be it directly or indirectly. That is, given a *DPSSystem*, any trace of any of its user services must contain exactly one user permission check. Invariant 2 (line 27) ensures that each service —be it user service or internal service— be protected by means of code permission checks.

The remainder of the section explains some assertions on the *DPS* style. The *Alloy* automatic analyzer is used to verify these assertions.

Theorem 1 (DPS System Existence). *There exists a DPS System.*

This theorem is formulated in the specification at line 29. We pretend that there exists no *DPSSystem*. Next, we request the analyzer to verify the assertion (line 31). The number 5 in the check command represents the scope of the analysis. The skolemization of each of the basic types defined in the specification is limited to 5 elements. For instance, the type `Event` is converted by the analyzer to the set { `event_0`,..., `event_4` }.

After checking the assertion, the analyzer replies with a counter-example (à la model checking) that proves the negation of the assertion, thus the existence of some *DPSSystem*. Some instances of *DPSSystem* found by the analyzer are shown in Figure 5. Among the 5 events, `event_4` is a user permission check while `event_1` and `event_3`, are code permission checks. Five systems are also generated by skolemization. All of them are *DPS* systems. Let us analyze `System_2`. It contains a single user service: `Service_4` and no internal service. This unique service has two traces composed of the events 1,3,4 and 1,2,4 respectively. One can easily verify that this service indeed has 1) a code permission check in the set of actions it invokes, and 2) each of its traces performs exactly one user permission check.

Theorem 2 (Permission Check Existence). *Each DPS System contains at least one user permission controller and one code permission controller.*

Considering the implicit fact that each permission check is oriented towards an access controller the existence of a user/code permission check implies the existence of a user/code access controller. We thus prove that each *DPS* system has a user/code permission check. The formulation of this theorem is given at line 33. It stipulates that the alphabet of any *DPSSystem* includes some user permission checks and some code permission checks. The verification of this theorem is done by instructing the analyzer to check it (line 36). The analyzer fails to find a counter-example.

Theorem 3 (Non-DPS System Existence). *Not every system is a DPS System.*

We already showed that there exists some DPS system. However, in the counter example given by the analyzer, all systems are *DPS* systems. This raises the question of whether there exists a system that is not a *DPS* system. We pretend that the set of all systems is equal to the set of all DPS systems (line 38). The analyzer is able to find a system that is not a DPS system, thus falsifying our assertion.

Theorem 4 (DPS Layered View). *Any DPS System has a 3 layer architecture.*

We classify the set of services in three categories. The first category consists of services that contain a user permission check in their traces, but not in their actions. In other words, this is the set of services that do not directly invoke user permission checks, but invoke other services that eventually perform user permission check (line 41). The second category of services consists of services that directly perform user permission checks thus have a user permission check in their set of actions (line 44). The third category of services is the set of services that has no user permission check in their traces (line 46). We show at lines 48-50 that these three layers are disjoint each from the others. We finally show at lines 52-54 that any *DPS* system is composed of these three disjoint layers. This theorem is illustrated by the MOTION layered architecture. The second layer is the ACTWS layer, the first layer is composed of all components below the ACTWS layer, and the third layer is composed of the business specific services and the presentation layer (see Figure 2).

We have formulated and analyzed other theorems on the *DPS* style. These theorems, however, are based on some other views, and thus other specifications and are not presented here.

6 Conclusion

We presented the *Dual Protection Style*, an architectural style for building access controlled software systems. This style distinguishes three kinds of components: user access control components, code access control components and protectees. Protectees are components whose services are protected by access controllers. The connectors between protectees and access control components are called permission checks. The *DPS* style constricts the interaction between protectees and protectors by means of two invariants. Besides an informal description of this style, we presented some of its advantages and shortcomings. We finally presented a formal specification of the *DPS* style using the *Alloy* specification language and verification of some of its properties.

A question that can be raised is how are applications constructed without the *DPS* style. We have presented a class of applications whose architectures are said to be degenerate forms of the *DPS* style that behave practically like instances of the *DPS* style. We gave examples of applications in this class. The notion of code permission recently raised attention (due to Java) although it was already investigated in the UNIX environment under the concept of domain type enforcement. We presented a framework of how to combine it with user access control.

The *DPS* style is not a panacea that is applicable in all cases. We presented an example application that does not satisfy the *DPS* style. Most advantages that we presented concerning this style are related to usability and easy applicability. The next step is to investigate how to specify the security properties of this style and how to prove them.

Another issue that needs to be clarified is that of composition. Is DPS compatible to software development approaches such as component based development? The answer to this question is yes. Components can still perform different user permission checks and being composed in a system that satisfies DPS. For this, the access control system must intentionally be built to support DPS. We are currently constructing such an access control system. The idea behind it is to have a global context variable that records if a user permission check was already performed for a given trace. If so, the permission check is ignored and otherwise performed.

References

1. Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *USENIX Computing Systems*, 9(1):47–83, 1996.
2. C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

3. Premkumar Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *ICSE 2000 Special Volume on The Future of Software Engineering*, 2000.
4. Antoni Diller. *Z:An Introduction To Formal Methods*. Oreilly, Mai 1996.
5. Pascal Fenkam, Harald Gall, and Mehdi Jazayeri. Visual Requirements Validation: Case Study in a Corba-supported environment. In *Proceedings of the 10th IEEE Joint International RequirementsEngineering Conference, Essen, Germany*, page to appear, September 2002.
6. Pascal Christian Fenkam. Dynamic user management system for web sites. Master's thesis, Graz University of Technology and Vienna University of Technology, September 2000. Available from http//www.ist.tu-graz.ac.at/publications.
7. Anup K Ghosh. Building software component from the ground up. *IEEE Software*, 19(1):14–16, January 2002.
8. GMD. Xql ipsi, http://xml.darmstadt.gmd.de/xql/, 2002.
9. Anthony Hall and Roderick Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, pages 18–25, February 2002.
10. Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2001.
11. Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methododlogy*, 11(2), April 2002.
12. Daniel Jackson. Automatic analysis of architectural styles. Technical report, MIT Laboratory for Computer Sciences, Software Design Group, Unpublished Manuscript. Available at http://sdg.lcs.mit.edu/ dnj/publications.html.
13. Kernighan and Pike. *The Unix Programming Environment*. Prentice Hall, April 1984.
14. Engin Kirda, Pascal Fenkam, Gerald Reif, and Harald Gall. A service architecture for mobile teamwork. In *Proceedings of the 14th International Conference on Software Engineering Conference and Knowledge Engineering Ischia, ITALY*, July 2002.
15. Charlie Lai, Li Gong, larry Koved, Anthony Nadalin, and Roland Schemers. User Authentication and Authorization in The Java Platform . In *Proceedings of the 15th Annual Computer Security Conference, Phoenix, AZ*, December 1999.
16. Marry Shaw and David Garlan. *Software Architecture-Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
17. Gary McGraw. Penetrate and patch is bad. *IEEE Software*, pages 15–16, February 2002.
18. Gary McGraw and Edward W. Felten. *Securing Java, Getting Down to Business with Mobile Code*. John Wiley and Sons, Inc, 1999.
19. Gian Pietro Picco and Gianpaolo Cugola. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems. Technical report, Dipartimento di Electronica e Informazione, Politecnico di Milano, 2001.
20. Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. In *ACM SIGPLAN Notices*. ACM SIGPLAN, September 1992.
21. Gerald Reif, Engin Kirda, Harald Gall, Gian Pietro Picco, Gianpaola Cugola, and Pascal Fenkam. A web-based peer-to-peer architecture for collaborative nomadic working. In *10th IEEE Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), Boston, MA, USA*. IEEE Computer Society Press, June 2001.
22. Michael P. Ressler. Security sensitive software development. In *IEEE International Carnahan Conference on Security Technology (ICCST)*, 1989.
23. Sun Microsystem. Security code guidelines. Technical report, Sun Microsystem, February 2000. Available at http://java.sun.com/security/seccodeguide.html.
24. The Institute of Applied Computer Science, IFAD. *The IFAD VDM Toolbox*. IFAD Danemark, 1999. Available from www.ifad.dk.
25. The Open Group. *Guide to Security Patterns, Draft 1*. The Open Group, April 2002. Available at www.opengroup.org.
26. Frank Tip and Jens Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*. ACM Press, October 2000.
27. John Viega and Gary McGraw. *Building Secure Software, How to Avoid Security Problems the Right Way*. Addison Wesley Professional Computing Series, 2002.
28. Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *Proceedings of the Pattern Languages of Programming (PLoP) Workshop*, September 1997.