

In Components We Trust – Programming Language Support for Weak Protection

Franz Puntigam

Technische Universität Wien, Institut für Computersprachen

Argentinierstraße 8, A-1040 Vienna, Austria

E-mail: franz@complang.tuwien.ac.at

Abstract

For many reasons we trust components just to some degree. Component users apply a number of methods to improve protection against malicious components. In this paper we briefly analyze some of these concepts and their relationships to our trust in components. It turns out that weak protection methods (those with potential security holes) can be beneficial for components we partially trust especially if potential holes are clearly visible. Visible holes build a basis for extending responsibility from the component user to the component supplier.

1 Introduction

Programming by composing software components has been a major research topic for several decades, and there is still growing interest in it from both industry and academia. Promises of component technologies are very convincing: Instead of developing new software from scratch we just compose existing components and get the same results at a lower price and in shorter time. However, experience shows that we are still far away from this ideal situation. Technical problems as well as management decisions and market forces make it difficult to compose already existing components in a desired way. Many difficulties have their roots in the fact that we do not trust components. We cannot know if a component actually does what it is supposed to do; it could be erroneous or even act as Trojan horse. Technical solutions like software testing and type checking can improve our confidence. The final decision whether and to which degree we trust a component remains at the management level.

In the present paper we take the position that no component is completely trustworthy. Nonetheless we want to compose components. We shield components from one another and from critical resources such that malicious components cannot cause too much harm. The focus is on programming language concepts. Shielding program parts is a quite old idea exploited many times: Data hiding by means of import and export declarations is a central concept in all languages with modules and already plays an important role in software composition. This concept (now often in the form of public, protected, and private declarations) together with encapsulation is essential to data abstraction. We reflect on some variants of these concepts and make proposals to improve the granularity of shielding.

2 Trust

There are many reasons why and to which degree we trust specific components:

Experience and Reputation: Often we trust components because they were developed by companies or groups we regard as trustworthy (especially our own group). Our trust can be based on business connections, personal contacts, previous experience, or just hearsay and personal preferences. We have specific expectations about the current and future quality of components – their life cycles. Usually we trust components just to some degree. We may be in fear of specific problems that possibly show up in the future or just of the uncertainty of available information.

Legal Contracts: Trust can be established by agreements between suppliers and users of components that may have legal consequences. Contracts can clearly distribute responsibilities between users and suppliers. Unfortunately, many contracts exclude liability for broken software and (even worse) restrict software analysis. It is an open question whether certification can circumvent this problem in practice.

Technical Superiority: Our trust can be based on careful analysis of components. We use testing methods and apply type checkers and other tools to ensure that components cannot do specific kinds of harm. Thereby we trust our own technical superiority in detecting potential problems. While current tools are quite effective in this regard they are by far not perfect, and they cannot say anything about future versions. Hence, some fear remains even if all tests and checks succeed.

Self-Confidence: It is probably a good idea not to blindly trust ourselves, especially our abilities to protect us against spying and system corruption. We can do much to protect us. This effort must be organized quite well to be effective. Because of missing feedback we usually do not know how good the protection actually is.

Potential Harm: In applications that do not much harm if broken we are often ready to apply components we do not really trust. Here is the problem that we may underestimate the potential harm of an application and do not see the danger for the whole system. There are not many harmless applications.

To improve our trust we can work on each of these topics. From a programming language point of view mainly two of them give chances for improvement: The integration of tools for component analysis (technical superiority) into programming languages prevents the use of obviously broken and inappropriate components, while additional organizational support for protection (self-confidence) helps in keeping the danger of spying and system corruption through (partially) trusted components at a low level.

3 Rejecting Broken Components

Programming languages and systems offer support for checking whether components are broken. At least interfaces (as far as specified) have to be consistent. This is a necessary requirement for cooperation between components and the hosting system. We differentiate between three approaches that can be used also in combination:

Dynamic Checking: The run-time or operating system prevents execution of illegal operations (like memory access outside of assigned memory pages) and thereby enforces access to critical resources only through published interfaces. Interfaces can restrict access in many ways. For example, programs on a Linux system can get access to a file only by invoking system functions when running with sufficient privileges for the permissions of the file. Such techniques are in use for decades and probably will remain to be important in the foreseeable future. On some systems we can run badly trusted programs with low privileges shielding most critical resources from them. Security does not (completely) depend on the quality of compilers producing executable code. However, for component composition this approach has disadvantages: To protect components from one another all communication must go through system interfaces. This communication is slow and essentially unspecified at the system level – just byte streams. We apply techniques for component interface specification and compatibility enforcement (as in CORBA [10]) possibly also based on dynamic checking, just at a higher level. Heavy-weight communication favors large components with loose coupling.

Static Checking: Compilers statically check interface consistency even for advanced interfaces at all levels. A combination of static interface checking with dynamic checks by the operating system seems to be an ideal solution: Communication between components need not go through system interfaces because compilers can ensure the required shielding statically. Light-weight components become useful. We know up-front whether components fit together, not just at run time. Compilers can still produce separate pieces of code for components linked together only at load time or run time. However, also this approach has its disadvantages: A compiler can pretend that interfaces fit together although actually they do not, and compiled components can pretend to being checked against interface specifications by a compiler although they are not. Electronically signing checked components reduces the latter problem, but we have to trust the compiler. Furthermore, not all desired properties can be checked statically (e.g., array access within bounds).

Load-Time Checking: Run-time systems of Java-like languages can repeat simple checks at load time to ensure type consistency [2]. This approach reduces dependence on the compiler. Checks shall not take too much time since this time belongs to the run time of an application. Thus, code has to exist in a form which makes it easy to be re-engineered, and we have all disadvantages of static checking.

In conclusion, up-front interface checking is necessary especially for light-weight components. Interfaces are abstractions of object behavior. There is a tendency toward more and more complex interface specifications going far beyond simple signatures of available routines [1, 3, 5, 8, 9, 11, 14] for following reasons: In object-oriented programming we create abstractions of real-world objects in the application domain. Because of this connection to the application domain we understand relationships without detailed specifications. Substitutability (subtyping) depends on often implicit relationships. However, when composing components developed with a more general or different application domain in mind, implicit domain knowledge vanishes. To compensate for it we make relationships more explicit in interfaces. Technically speaking, within a homogeneous

system we have common supertypes of related concepts, while there may be no or even several (more or less equivalent) common supertypes of related concepts in heterogeneous systems composed of components. As an alternative to using named common abstractions (nominal supertypes) we can work with anonymous structural interfaces which specify (otherwise implicit) behavioral constraints in a rather detailed way. It is undesirable to need overly complicated interface specifications, but sometimes we cannot avoid them. Today, advanced type systems can statically check very complicated interface specifications. However, in practice nobody wants to specify and maintain complicated interfaces. There are many problems with automatically inferred interfaces mainly because there is no separation between intended behavior and implementation detail in program code. Inferred interfaces likely change with new versions.

While there are already approaches to include some non-functional properties like performance requirements and limits on resource consumption into interfaces, other properties like maximal failure rates will hardly ever be automatically verifiable. Hence, technical superiority can never be the only reason for trust.

An important point worth to be repeated is that static interface checking can only reject obviously inappropriate components. Components passing all checks can still be harmful. It is probably no good idea to replace interfaces based on common named abstractions with complex anonymous interfaces just to allow the compiler to do a little more checking. Introducing and making use of common named abstractions should be considered as a design goal in component design and composition although this goal cannot be achieved as easily as with conventional object-oriented programming. It is much easier to trust components with common abstractions.

4 Protection through Interfaces

If we ensure interface compatibility up-front and patch up all holes, interfaces are a powerful mechanism for shielding resources from access by components. All communication has to go through interfaces. A resource usable only through a specific interface can be protected by controlling the communication through this interface.

Languages provide several mechanisms of protection through controlled interfaces:

Protected Access: In virtually all object-oriented languages currently in use we differentiate between publicly accessible methods and those accessible only within restricted environments. Under protected access we understand in this paper access restricted to specific components. Other components cannot invoke protected methods even if they know the objects containing them. Unfortunately, the situation is quite involved in practice: If we consider components just to be classes, then private access is a perfect choice. Otherwise (components are of larger granularity) we can select for example protected visibility in C++ or default visibility in Java. When using protected visibility we have to be careful not to open holes through subclassing; protected methods are accessible within subclasses even if belonging to other components. It is always possible to introduce holes: Even if another component cannot access a method directly, it may be able to access a public method that performs the access in question. Values in protected variables can become available to other components through public variables or method invocations.

Hidden Types: In systems based on nominal types, static type checking, and modules we can restrict the visibility of types in other components. Methods and variables specified by invisible types (but not by public supertypes) are not usable. Careless design can cause components to get access through other public methods. Nonetheless, hidden types have a potential advantage over protected access: We can design systems where several components use a common type while others cannot: There are possibilities like file access restrictions to control the visibility of modules in a detailed way. Types usually are not passed around at run time.

Hidden Objects: Recently new techniques were proposed that prevent object references from escaping from protected areas [15]. With these techniques other components cannot get access to object references they need for method invocation. This approach is reliable because it is possible to show that hidden objects actually cannot escape. Anyway programmers can introduce holes by propagating messages to hidden objects. This approach is quite restrictive by providing an all-or-nothing strategy: Either no other component can get access to the object reference or all of them can. Protected access uses another kind of all-or-nothing strategy: Either no other component can access methods of visible objects or all of them can. Sometimes it is useful to combine these kinds of protection.

Protection Proxy: More flexible protection mechanisms do not just rely on programming language features, but mainly on programming techniques. A protection proxy is probably the most important software design pattern in this respect. Careful pattern applications can lead to quite good shielding with very fine granularity of control. However, a protection proxy usually cannot see where a method invocation comes from – a trusted or untrusted component. Therefore, we need language-based protection even when programming with proxies.

Partial Hiding: Some programming techniques in general use protected access, hidden types, or hidden objects, but purposefully open holes to give other components selective access to needed resources. These techniques are inherently dangerous because in principle a component can use visible objects and methods in arbitrary ways. Some people argue that partial hiding does not exist; if another component has access to an object, then this component may allow further components to access the object. Nonetheless, partial hiding is a useful approach to protection if we trust components not to use critical information in an undesirable way.

Limited Use: Tokens were proposed as a language mechanism for the specification of dependences between method invocations in advanced interfaces [12, 13]: Method invocations can consume tokens (representing limited resources similar to tokens in a token net) to be provided by the caller and issued only by the object containing the method. A mechanism allows us to dynamically distribute and use tokens statically checked as part of the type system. Thereby we can, for example, limit the number of simultaneous invocations or enforce specific orders of invocations (by issuing a token needed for method *b* only after executing method *a*). While this concept is not supposed to be a protection mechanism, it can help to improve protection of (partially hidden) objects by selectively restricting their use.

In conclusion, there exist several language mechanisms for the desired purpose. However, each mechanism can easily lead to holes if not used carefully. Furthermore, except of hidden types all approaches either use some kind of all-or-nothing strategy or depend on programming techniques rather than language features.

Now we analyze how these protection mechanisms can be used in a typical situation: We consider a component interacting with two further components, where the considered component has two resources and wants to allow each connected component to access one of the resources. In this situation we can neither hide the objects representing the resources nor protect access to corresponding methods since other components need access to the objects and methods. Because of an all-or-nothing strategy it is not possible to selectively open one resource for one component and the other resource for the other component. Hidden types cannot solve this problem if both resources are of the same type (which is usually the case). We cannot rely on language features and must use programming techniques to implement the desired protection. A protection proxy cannot reliably differentiate between components invoking a method; this information is usually not available. Of course we can require components to identify themselves through method arguments, but components can cheat by giving wrong arguments. Hence, we must trust components when using such techniques. Since we have to trust the components anyway we do not lose anything by using partial hiding instead of a protection proxy. We need further information on access patterns of the resources to lower the level of required trust by limiting use through tokens.

As in this example we quite often run into the problem that only weak protection (that requires a high level of trust in components) gives us the necessary freedom. The only alternatives would be to have completely unprotected resources or not to use the components. We have to decide whether we trust components sufficiently.

5 Weak Protection and Responsibility

Usual program code does not immediately show if and how methods circumvent protection mainly because they are spread all over the code. Beside of its unreliability this is the major disadvantage of partial hiding. Paradoxically we can improve the state of the art by introducing further holes into protection methods, where syntactic restrictions ensure that it is easy to find these holes in program code. For example, explicit import statements for modules or classes taking (generic) parameters are appropriate for this purpose. Here are some proposals of what the holes on such parameters may look alike:

Protected Access: Protected methods and variables are accessible through formal parameters of modules and classes although they belong to another component (the one providing corresponding actual parameters in import statements).

Hidden Types: Constraints on formal type parameters (for example, as specified by where-clauses or required structural supertypes) of modules and classes are regarded as satisfied if method and variable declarations in corresponding actual type parameters are protected (and not public as usually required).

Hidden Objects: Actual parameters of modules or classes in import statements can be hidden objects, thereby allowing hidden objects to escape in a controlled way.

Additional holes can improve protection if they allow us to apply a more restrictive protection method than possible without holes – weak protection instead of no protection at all. The essential point in these proposals is the use of explicit import statements easily found in program code and obviously belonging to glue code. When writing glue code programmers know that they are composing possibly unreliable components. For example, `import ModuleA<T>(x)` at the top of a file makes clear that we give `ModuleA` protected access to `x` and `T`, and a definition `module ModuleA<T>(x:AType){...}` makes clear that the module is responsible for keeping `x` and `T` protected. In contrast, when just writing public methods accessing protected methods or variables, programmers may not be aware of the existence of possibly unreliable components and holes in the protection. Similarly, when just using `ModuleA<T>` or even `ModuleA` (a raw type as in Java, not specifying a type for the type parameter) as type somewhere in a program, readers of the program cannot immediately see the holes.

Import statements and parametric modules have been in use for many years (for example, in Ada [4] and Haskell [6]), and parameters of modules are very good in specifying connections between components in a rather static way. In general, static specifications are usually more readable in program code than dynamic interactions. However, establishing connections between components dynamically and step by step (one invocation of an initialization method or constructor after another) gives us more flexibility. Opening holes just in import statements is not sufficient if components must be composed in a very flexible way or component compositions change dynamically. For the dynamic case we can adapt this approach by introducing the same kinds of holes to arbitrary parameters, not just parameters of modules and classes. Of course it is necessary to clearly specify the parameters in concern. Corresponding statements have to occur in the code of the component owning a resource as well as in the component needing access to this resource. It must be clear that such statements belong to glue code.

By making use of such holes through designated parameters we establish a contract [7] between the component owning a resource and the one needing access to the resource: The owner allows the other component to access the resource in a protected way if the other component agrees to keep the resource protected. If the component accessing the resource through a hole allows further components to get protected access to this resource (intentionally or accidentally), then the supplier of this component is responsible for corresponding security problems, not the owner of the resource. Explicit and visible specifications of such holes are important to make clear who is responsible.

Who shall be blamed if we trust a component and the component turns out not to be trustworthy? The answer depends on the reason of the problem, but in most cases the user of the component has to be blamed. Only one of the reasons for trust given in Section 2 can shift the blame to the component supplier: Legal contracts specify the supplier to be responsible. There are good reasons to regard contracts being part of program code also to be legal contracts. Under this assumption the proposed concept of clearly specified holes in protection mechanisms provides a means to shift trust based on self-confidence to trust based on legal contracts. Thereby, this concept extends responsibility from the user of a component to the component supplier. Of course this way we make components responsible only for a specific and rather simple aspect (compared to the many aspects relevant for trust). Unfortunately, responsibility for most aspects (especially non-functional properties) cannot be treated in such simple way.

6 Conclusions

It is to a large extent the obligation of component users to protect themselves against harm caused by malicious components. Programming language concepts help to protect users in two ways: They reject obviously inappropriate or broken components and shield critical resources from potentially harmful components. Protection through language concepts cannot be perfect and in practice nearly always has holes. We must trust components to not make undesirable use of such holes. Controlled holes in shielding can even increase overall protection by allowing trusted components to access resources while other components have no access to these resources. Clearly visible holes extend responsibility for not allowing public access to resources from component users to components.

References

- [1] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.
- [2] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 1996.
- [3] Dirk Heuzeroth and Ralf Reussner. Meta-protocol and type system for the dynamic coupling of binary components. In *OORASE'99: OOSPLA'99 Workshop on Reflection and Software Engineering*, Bicocca, Italy, 1999.
- [4] ISO/IEC 8652:1995. Annotated ada reference manual. Intermetrics, Inc., 1995.
- [5] H.-Arno Jacobsen and Bernd J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *IEEE International Conference on Automated Software Engineering (ASE'98)*, pages 63–72, Honolulu, Hawaii, USA, 1998.
- [6] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries; The Revised Report*. Cambridge University Press, 2003.
- [7] Bertrand Meyer. The grand challenge of trusted components. In *ICSE-25 (International Conference on Software Engineering)*, Portland, Oregon, May 2003. IEEE Computer Press.
- [8] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand modularization. In *OOPSLA 2002 Conference Proceedings*, pages 52–67, Seattle, Washington, November 2002. ACM.
- [9] Oscar Nierstrasz. Regular types for active objects. *ACM SIGPLAN Notices*, 28(10):1–15, October 1993. Proceedings OOPSLA'93.
- [10] Object Managment Group. *The Common Object Request Broker: Architecture and Specification (CORBA)*, 1993. OMG DOCUMENT Number 93-12-43.
- [11] Frantisek Plasil and Stanislav Visnovsky. Behavioral protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.
- [12] Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Der Andere Verlag, Osnabrück, Germany, 2000.
- [13] Franz Puntigam. State information in statically checked interfaces. In *Eighth International Workshop on Component-Oriented Programming*, Darmstadt, Germany, July 2003.
- [14] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.
- [15] Tian Zhao, Jens Palsberg, and Jan Vitek. Lightweight confinement for featherweight Java. In *OOPSLA 2003 Conference Proceedings*, pages 135–148, Anaheim, California, October 2003. ACM.