Safe Mobile Code – CPPCC: Certified Proved-Property-Carrying Code*

Zoltán Horváth, Tamás Kozsik
hz@inf.elte.hu, kto@elte.hu
Department of General Computer Science
University Eötvös Loránd, Budapest, Hungary

Abstract

We propose a hybrid model of safe mobile code exchange. The receiver needs a guarantee that the properties of the received code correspond to certain requirements. The safest solution is to verify run-time just before the execution of the code that the requirements are satisfied. The other extreme is to trust in the sender of the code unconditionally or, using a certificate mechanism, to check merely the identity of the code producer. The first approach is safe, but very resource-consuming, the latter, on the other hand, is fairly effective, but does not provide the required level of safety. We propose an architecture that possesses the advantages of both approaches.

1 Introduction

Languages that support some sort of mobile code have become wide-spread recently. Such languages are e.g. Java, Emerald [9], Dynamic ML [3] and Clean [8].

When an application runs a piece of code received via the net, the application may want to get guarantees that this piece of code does no harm: does not use too much resources, does not read or modify data unauthorised, etc. Proof-carrying code [6] is a technique for providing such assurances. With PCC, the code consumer specifies safety requirements, which tell under what conditions parts of memory can be read or written or how much of a resource is allowed to use. The code producer must also provide a set of program properties and an encoded proof packed together with the code transmitted, to certify that the code really satisfies these conditions. Complex methods may be used to construct these proofs [7, 5], but their verification should be a simple computation.

Java, for example, makes it possible to download code over the internet and execute it on a client machine. This technology, offered by class loading (like in

*Position paper. Supported by the OTKA T37742 and the OMFB 01548 "DemoGRID" projects.

case of applets) and object serialization is supported by the abstract machine approach that Java utilizes. The whole concept is not bound to the object-oriented paradigm. In the functional programming domain the Concurrent Clean system applies a similar concept of mobile code in the form of "dynamics".

In both examples the compiler packages type information with the code. The receiver run-time system can check this type information due to its dynamic typing possibilities. Java's class file verifier mechanism (of which the most complicated phase is performed by the so-called Bytecode Verifier) can prove static and structural constraints and type correctness during linking time – which, in fact, is run-time from the point of view of the overall execution of the program. (In the following we will refer to this as "dynamic-time" as opposed to "static-time".) These checks ensure that the code is well-formed, and the applied data-flow analysis guarantees that there will be no operand stack overflow or underflow, that local variable uses and stores are valid, and that the arguments of the instructions are of valid types. This mechanism does not provide full semantic checking, moreover the data-flow analysis is expensive. Due to this latter, some restrictions have been introduced, e.g. the code size, the number of methods, variables or constants, the size of operand stack, etc. are limited.

The dynamic linker and the run-time system of Concurrent Clean trusts in the integrity of the code and the associated type information. Clean performs run-time type checking (pattern matching) based on this type information when evaluating dynamically loaded code.

It would be very fruitful to include further semanctical information to the code compared to the above approach. This additional information would describe the run-time behaviour of the code and the receiver would be able to decide to which extent is the obtained code appropriate for solving a certain specified problem. For example the receiver could expect a piece of code that can sort a list of integer numbers. Type requirements for such a code would be: take a list of integers and produce a list of integers. These requirements can be type-checked. Structural constraints can prescribe that the sorting operation does not perform illegal memory access – the aforementioned Bytecode Verifier can make such a guarantee. On the other hand the following additional semantic requirement would also be necessary: the result of the operation is such that every element of the list is either the first element or an element that is greater than or equal to the previous element. A safety-critical application must be sure at least that the dynamically linked code, if it terminates, satisfies this requirement. (A time-critical application would also set up a requirement on termination.)

The widely-used way to avoid malicious code is that we accept code only from trusted partners who guarantee the expected behaviour. The identity of the partner can be proved by e.g. public-key cryptography systems or certificates. But our problem remains: we still do not know whether the trusted partner has made a mistake or not while implementing the code. It is possible that the code does not perform sorting in certain cases due to a programming error.

Higher confidence could be achieved, if we kept the idea of needing to check the properties of the code upon receiving, but, in contrast to the Java model, we would check all required semantical properties. For example, we could verify that the list became sorted after performing the dynamically obtained code, or we could verify that the code really implements a sorting algorithm.

To avoid the heavy dynamic-time overhead introduced by an operational analysis of the code or a run-time check of the postcondition we may use proof-carrying code. The code producer develops proofs about some specified properties of the code static-time. This process may be very complicated and may require a lot of time to complete. The proof will then be packaged with the code. The receiver only has to check the proof – and this can be performed fairly efficiently.

In this paper we propose a technology which uses minimal run-time overhead but still makes it sure that the receiver will use code which is proved correct.

2 The Certified Proved-Property-Carrying Code architecture (CPPCC)

In this section we describe how to transmit code in a secure way. There are three main parties involved in the scenario. First, there is the producer of the mobile code, who also produces proofs of properties of the developed code. Then, there is the receiver who receives the code from the code producer and executes it – but only after certain safety checks which ensure that the code satisfies the requirements specified in the receiver's code. Finally, there is the certifying authority between the two, whose task is to reduce the work-load of the receiver by performing the most resource-consuming parts of the checks static-time. The overall view of the protocol, illustrated in figure 1, is the following:

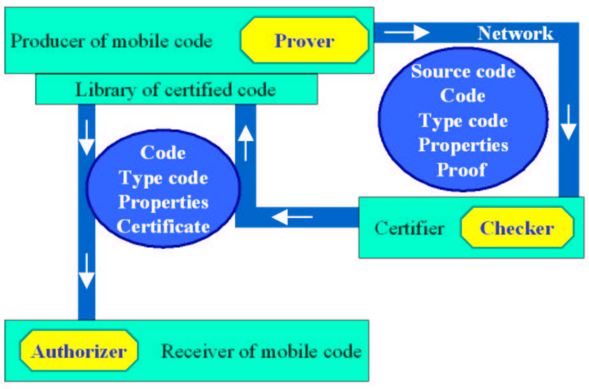

Figure 1: Secure transmission of code – Overview

1. The code producer packages the code with the original source code, type information (type code), some important properties of the code, and the proof of theses properties produced by the help of a (semi)automatic theorem prover.

2. The certifier checks the proofs in respect to the source code and the properties, then it checks whether the abstract code is really the result of a proper compilation of the source code, and finally it checks whether the type information is correct. If everything succeeds, then a certificate is added. The certified code can be placed in a library for further use.

3. The receiver checks dynamic-time the certificate, compares the properties and type information with requirements and accepts or refuses the application of the code.

Certain properties of programs can be very well expressed by attaching "invariants" to types [4]. The propagation of such type invariants can be administrated by the type system. This approach leads to increased efficiency, since less verification is needed by a first-order logic based proof system.

2.1 Production of mobile code

The production of the mobile code (see figure 2) starts from an annotated source code. The intended global properties of the component are added to the program

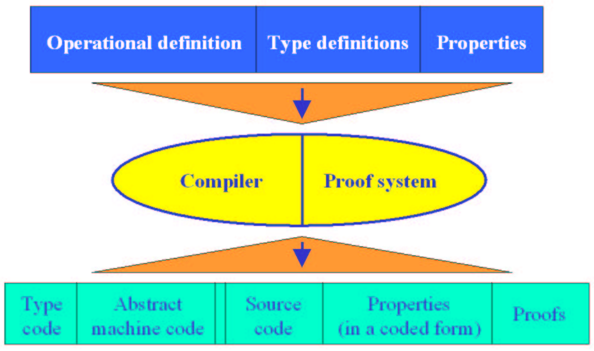

Figure 2: The production of mobile code

text in form of annotations. Type declarations may be obligatory or produced by type inference. The compiler and the theorem prover produce an output, if the properties are valid in respect to the source code. The result contains the original source with the annotated global properties, the proof of these properties (as an encoded sequence of verification steps), type information inferred and the generated abstract machine code. All these pieces of information are needed to perform certification.

2.2 Certification of code

The results from the previous step are sent to a certifying authority whose task is to perform as many checks as possible. Figure 3 demonstrates this idea. The

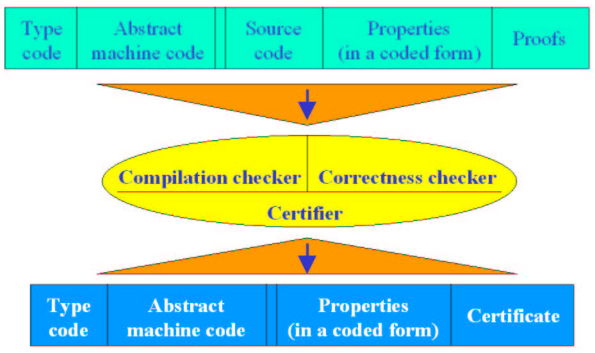

Figure 3: Certification of mobile code

certifier checks the proof in respect of the given properties and the source code. This task is easy to automatize, there is no need for human intelligence, which may be required to support the construction of proofs. After the check is done, the source code is not needed any more, moreover we do not need the proof either. The proved properties are sufficient for the future dynamic-time checks. The other task of the certifier is to check the abstract machine code, whether it is originated from the source. We drop the source, so this is the last moment for such a test to perform. If all checks succeed, the certifier adds a certificate to the abstract machine code, the type code and the encoded properties, and sends these data back to the mobile code producer.

2.3 Checks performed by the receiver of the code

If an application needs a component available at a mobile code producer, it can request and receive a certified proved-properties-carrying code, which can be dynamically linked into the application – of course, after the necessary safety checks. This scenario is presented in figure 4. First, the receiver checks the certificate. If the certificate is valid, the type information and carried properties are compared to the type and semantic requirements specified for the dynamically

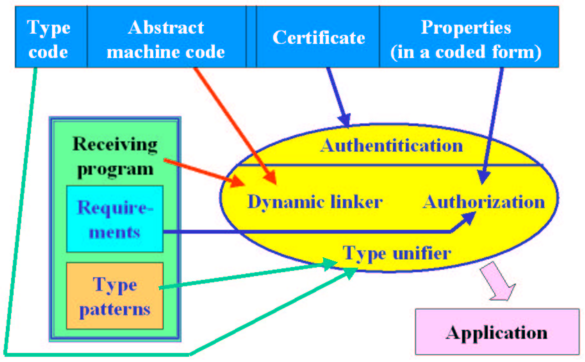

Figure 4: Using the mobile code in the receiver

loaded component. If all these requirements are satisfied, the receiver accepts the code, links it to the application and executes it.

The matching of requirements and properties can be hard in general. To preserve efficiency (by avoiding the use of a resource-consuming proof system for this purpose) the receiver will in most cases make a decision to refuse code whose specified properties are not close enough to the requirements. Thus it will be the responsibility of the mobile code producer to supply the appropriate properties with the code.

This approach can make use of incremental extension of certified proved-properties-carrying-code with further properties.

References

[1] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[2] Horváth Z. et al.: Verification of the Temporal Properties of Dynamic Clean Processes. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages, Lochem, The Netherlands, IFL'99.*, pp. 203-218.

[3] Gilmore, S., Kírlí, D., Walton, C.: Dynamic ML without Dynamic Types. *Technical Report ECS-LFCS-97-378*, Laboratory for Foundations of Computer Science, Department of Computer Science, The University of Edinburgh, 1997.

[4] Kozsik T., van Arkel, D., Plasmeijer R.: Subtyping with Strengthening Type Invariants. In *Proceedings of the 12th International Workshop on*

Implementation of Functional Languages, Aachener Informatik-Berichte, Aachen, Germany, September 2000, pp. 315-330.

[5] de Mol, M., van Eekelen, M.: A Proof Tool Dedicated to Clean. In *Selected papers of Applications of Graph Transformations with Industrial Relevance*, AGTIVE'99, Kerkrade, The Netherlands, Springer-Verlag, to appear in LNCS.

[6] Necula, G.: Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1997.

[7] Owre, S., Rushby, J., Shankar, N.: PVS: a prototype verification system. In *Proc. 11th Intl. Conf. on Automated Deduction, Springer LNCS vol. 607*, pages 748-752, 1992.

[8] Pil, M.R.C.: Dynamic types and type dependent functions. *Proc. of Implementation of Functional Languages, IFL'98*, London, LNCS 1595, pp. 169-185, 1999.

[9] Hutchinson, N.: *The Emerald Distributed Programming Language.* http://www.cs.ubc.ca/nest/dsg/emerald.html

[10] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001. http://www.cs.kun.nl/~clean/Manuals/manuals.html