PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 09-11

Towards Secure Distributed Computations

Felipe Zipitría

2009

Towards Secure Distributed Computations
Zipitría, Felipe
ISSN 0797-6410
Reporte Técnico RT 09-11
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, 2009

Towards Secure Distributed Computations

Felipe Zipitría

fzipi@fing.edu.uy
Grupo de Seguridad Informática
Instituto de Computación, Facultad de Ingeniería
Universidad de la República
J. Herrera y Reissig 565, Montevideo, Uruguay

Abstract In this paper, we present an infrastructure for securing distributed computations between hosts, using a novel technique called Proof Carrying Results. This technique is based on Necula's proof carrying code. Basically, the result of some computation comes equipped with a certificate, or witness, showing that the computation was made correctly. This witness can be used to verify that the value was generated in a good way. We will show how to add the PCR technique and its supporting infrastructure to a distributed programming language. This will make the language more robust against active adversaries, when the returned values of a computation are of abstract types. Finally, to check the values and associated witnesses produced by some host, we use the COQ proof checker for a precise and reliable verification.

1 Introduction

There are plenty of networks that work in a cooperative way and form what we know as *grids* of computers. These grids serve a lot of purposes, and they are used with good results for intensive calculation, because the joined computing power aids in solving any kind of complex functions. To cope with these new requirements and facilities, programming languages had to evolve into new paradigms, including facilities to do distributed computing in a straightforward way. These are known as *distributed programming languages.* Using this approach, how the network is accessed and where processes are located should be avoided as a concern for the programmer.

Type-safety is a property concerned with *how much* a programming language protects its own abstractions. It is usually viewed as two properties of the semantics of the programming language: *progress* and (type-) *preservation.* Not every programming language has this property, and in the ones that are distributed it is more difficult to conceive.If the programming language semantics and the type system guarantee that the encapsulation provided by type abstraction can never be breached, then the language is called abstraction safe. On a single computer it is usually guaranteed by the compiler.

However, this changes radically when we begin to transmit data over the network. Data can be modified, lost, or attacked. After the creation and utilisation of such languages, an aspect remaining to be introduced is the *security* properties of these computations. The security properties of languages that execute on a single host are hard to maintain. We must take increased precautions when dealing with lots of hosts and complex networks.When dealing with remote computations, receiving the correct values for our computations is a must: if not, we are wasting time and effort. The correctness of the received values is independent of its type, such if it is concrete or abstract. For values of concrete types, we could have its representation available for checking that the value received is of that type. This is not the case for abstract types: its representation even could be not available at all. Therefore, how can values of abstract types be secured, in the context of a distributed programming language?

We propose the use of a novel technique, called Proof Carrying Results [BP06] (PCR). Basically, the result of some computation comes equipped with a certificate, or witness, that can be used to check the correctness of these values of abstract types. There are many ways for a host to check the correctness of this value, by using the associated witness. On one hand, it can be verified using a custom made checker for the particular abstract type. This way is prone to errors, in the creation of a correct checker for these values. On the other hand, a proof checker can be used as a reliable tool for checking correctness.

Throughout this work, we will use a language called Acute [SLW$^+$04]. Its main features are a large part of what is needed to produce a typeful distributed programming, and involve type-safe marshalling of arbitrary values by using two primitives*, marshal* and *unmarshal.* In this work we show how to add the PCR technique to the Acute distributed programming language. The supporting infrastructure for the technique is introduced along with it. For checking the values and associated witnesses produced by some host, we use a proof checker for a precise and reliable verification.

Therefore, our contribution is threefold:

- an infrastructure has been defined and implemented for supporting the technique of proof carrying results,
- the Acute distributed programming language has been extended, with a mechanism that permits the exchange of abstract values in a certified way, and
- for doing the verification of the results, this infrastructure has been connected with the COQ proof checker.

This paper summarises part of the work that has been developed in our Master thesis [Zip08b]. The document is organised in this way: section 2 has a description of the general problem and the particular case we focused on. We continue with the introduction of the Proof Carrying Results technique in section 3, and how this technique can be used to help us solve the problem presented. In section 4 we show the infrastructure that was developed to support PCR in Acute. Finally, we detail related work, and some concluding remarks and further work.

2 (Un)Trusted remote computations

In this section we present the problem of remote computations between untrusted hosts. Figure 1 shows a basic interaction between hosts. There, some host sends a value a to an untrusted part. It asks for the computation of a function f with the value sent.

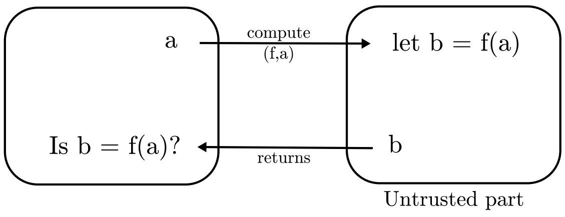

Figure 1. Remote computation problem

Afterwards, the untrusted part obtains b as the result of applying the function f to a. Then b is returned to the initial host. The question now is: how can this host be sure that b is effectively the result of applying function f to a?. For this to work we must require that the result b comes along with an additional information that shows that the computation was made correctly. A solution is proposed in section 3.

In particular, the language Acute has a problem related to this one. Acute has powerful properties: type and abstraction safety are guaranteed along the distributed system. The language has documented security problems when unmarshaling values of abstract types. If the value marshalled is of an abstract type, the representation could not be available at unmarshal time. Therefore, in this case: how can it be assured that this value was generated by a well-behaved Acute run-time?

What happens when there are entities that can tamper with data transmitted between hosts? The presence of active adversaries in the networks changes the scenario. If this situation occurs, safety can be no longer guaranteed. In this case, an additional check is that the marshalled value is a well-formed representation of something of that type. Nevertheless, the implementation of that type could be used for checking types at unmarshal time. But this implementation is not available, and especially in the case of abstract types. This limits the language to a simple decision to handle this case: to work only in a trusted scenario, or to marshal only values of concrete types. For a complete and detailed presentation of this we refer the interested reader to [Zip08b].

We will extend the language to support marshalling of abstract types in a non-trusted scenario. For this purpose, we will introduce the PCR technique into the language, and a supporting infrastructure for it that will end increasing the properties of safety in a distributed context. The next section summarises the contributions of this work.

3 Proof Carrying Results

This technique was introduced in [BP06]. The authors propose that some of the concepts introduced by [NL96] in proof carrying code (PCC) be reused. In that work, Necula put formal mechanisms to solve the problem of remote code execution in a behaviour-controlled way. As in PCC, there are two parts exchanging information, but here the exchange is applied in a different scenario. Instead of a code consumer, we have a results consumer: some host consumes a remote function, and the host that makes the computation sends its results with additional information that allows the consumer to validate this result.

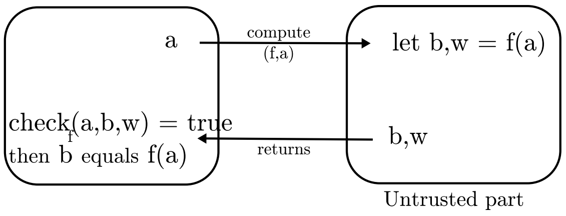

Figure 2. Basic PCR approach

The approach is that a consumer host sends an untrusted part some computation to be done remotely. The untrusted part then returns the result of this computation, with a *certificate* that the computation has been done in a correct way. Then this certificate can be used to verify if the result was computed the correct way.

Certificates provide additional data to check the correctness of computations. They can contain *witnesses*, which are generated in the process of computing, and *proofs*, that state properties about the result. These witnesses are like a trace of the computation made, and are closely related to the final result returned. Proofs, which can be part of the additional data carried, can be there to establish properties about the result itself, or about the witnesses.

Formalisation

We have a function f and we want to delegate its computation, using a as argument:

- $f(a)$ is delegated to an untrusted party, $f \in A \rightarrow B, a \in A$
- $b \in B$ is the expected value

To verify that this value is the value we sent to compute, we must have some function *check*:

$$check_f \in A \times B \to bool \mid \forall (a,b) \in A \times B, check_f(a,b) = true \Rightarrow b = f(a)$$

In general, PCR allows the untrusted part to provide additional data H intended to ease the checking process. Thus, one may have a checker function $check_R \in A \times B \times H \to bool$ such that $check_R(a,b,h) \Rightarrow R(a,b)$. In this way, the functional specification f is generalised to an input-output specification R.

This technique challenges the traditional algorithms, because more information could be needed from them, not only their result. In the next section we introduce a new kind of algorithm that is the foundation of this approach. This kind of algorithms are called Certifying Algorithms. The term was introduced in [KMMS03], and a more general approach was given in [MEK$^+$05]. For a complete and detailed presentation of this we refer the interested reader to [Zip08b]. An algorithm or program is certifying when, along with the result it was supposed to give, it returns a certificate or proof that this result is indeed the correct for the given input. This is a pragmatic approach to program correctness.

4 Infrastructure for PCR in Acute

In this section we present a generic infrastructure for doing Proof Carrying Results. To this aim, we begin introducing primitives that describe flows in distributed computations. We show how the infrastructure is implemented in the Acute language, for certified result communication. Furthermore, it will be used for verifying the witnesses of the computations. Finally, we show how the certificate checking process is made, connecting the Acute language with the COQ proof checker.

4.1 Primitives for distributed computation

The primitives introduced here are operations, or a sequence of operations, that describe different ways of obtaining a certified result. Each primitive will then be shown as an interaction flow between entities.

In this case, these entities will be 3 hosts interacting with each other, which are called *Alice*, *Bob* and *Trent*.

The first flow is from *Bob* to *Alice*. *Bob* sends *Alice* a value v of type T. In this case, *Alice* did not directly ask for this value. There is no guarantee that *Alice* may ever receive or process the value sent by *Bob*. Thereafter, we could say that this value has been pushed from *Bob* to *Alice*.

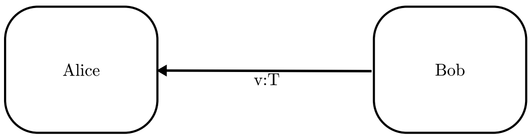

Figure 3. First flow

The second flow is from *Alice* to *Bob*. Now *Alice* `asks` *Bob* for some value of type T. Then *Bob* returns that value of type T.

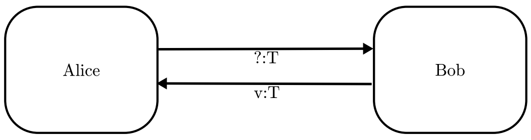

Figure 4. Second flow

The last flow happens between *Alice* and *Trent*. *Alice* has a value v and asks *Trent* for a proof that v is of type T. If *Trent* can construct such proof $\boldsymbol{x}$, it returns $\overrightarrow{x}$ to *Alice*. When some exception happens, for example, if no proof can be found at all, an error is returned.

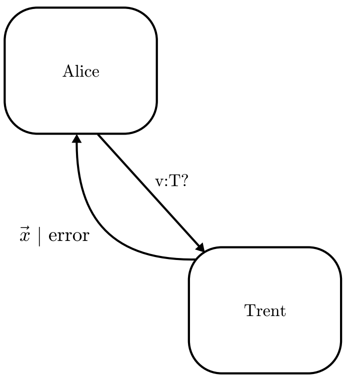

Figure 5. Asking for a proof

These primitives can be combined to increase the types and complexity of flows in the system. We will be interested in the particular flows (or sub-flows) that include proofs of the value sent along the wire. In the next subsection we show how these primitives, combined, are used as a PCR infrastructure model.

4.2 Infrastructure model

Using the primitives and flows previously described, we will model an infrastructure for performing result certification. The infrastructure is modelled by the interaction of the 3 hosts shown in Figure 6.

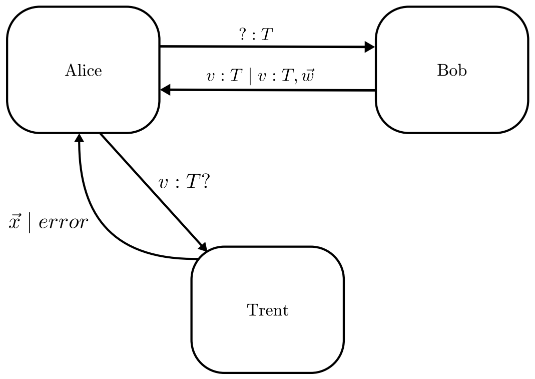

Figure 6. Infrastructure model

A simplification of this infrastructure is that *Alice* ≡ *Trent*. The full infrastructure considers all the flows and primitives presented. In our implementation, we will focus on the primitive $v : T, \boldsymbol{w}$.

4.3 Implementing the infrastructure in Acute

Now that we have introduced a general infrastructure for using in distributed computations, we show how it is applied to the Acute language for obtaining certified result communication.

We extend the way that we use the marshal primitive by adding a witness, obtaining a new syntax for it

marshal e1 e2:T <u>witness</u>

which may now include the information that can be used by the receiver to check an invariant. The use of witnesses for certification is optional: if there are no witnesses the primitive will behave as the old one. In the case that certificates are required by the receiver, the sender cannot choose: the value must be sent with a witness or it will not be used by the receiver, and an *exception* will be raised.

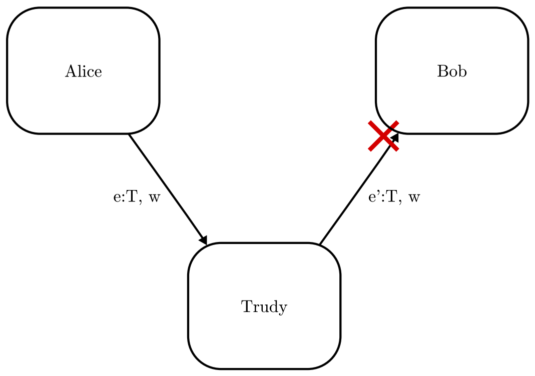

Figure 7. Adversary can not modify values only

The semantic of the primitive is changed accordingly. With this extension, our system is more robust against active adversaries. It is important to note that the adversary could modify the value *and* recalculate the witnesses for this new value. This case does not directly affect the problem we are trying to solve. This happens because, even when the original value has been modified, it complies with the invariants the new value must have. The witnesses will be used to check this compliance, and all we should care about is that the witnesses are well generated to certify the value. A trivial example is that we ask a remote host for a prime number, and the remote host sends us some number. In the middle of this transaction, an adversary changes this number. As long as the new number is also prime and it is sent with a proof, things will keep working and abstraction safety is kept.

Verifying witnesses

To complete the process of result certification, the receiver must perform some actions when receiving the value. We suggest the use of the received witness to check that the invariant holds for the value received. Adding up, the primitive for unmarshal

unmarshal e:T'

will be changed to reflect the new extension. The new primitive

unmarshal e:T' witness

will support the use of witnesses. In the unmarshal process we will check that the invariant holds for the module by using that witness. Therefore, our main contribution is to add functionality to Acute that allow us to send values of an abstract type along with a certificate that proves that the value complies with the type invariant. Enforcing the check of witnesses must be supported, because for some types we want to receive certified values for continuing our computation. The unmarshal, as in the marshal case, alters its semantics in a similar fashion to include the witness.

4.4 Certificate checking

After receiving the values, with extended information about its computation, the infrastructure uses the certificate to check the value just received. For the process of checking the certificate we decided to use the COQ proof assistant [dt08]. Even thought COQ has many impressive functionalities, in this case it is used only as a proof checker for the received certificate.

In general, to interact with COQ, the user has a command-line interface. This is because it is commonly used interactively. We have that a simple proof contains definitions, declarations, and the actual proof. This proof is surrounded by the "Proof" and "Qed" words. At first sight, we did not have the prerequisites of interaction, we just needed to have certain proof approved or not. To use it in this way, the value and its certificate, along with other pre-defined constants and COQ commands, were written into a file. After that, we call the COQ compiler passing this file to be compiled. This was easy to implement, and we could obtain basic results with it: the compiler returned a value 0 if the proof was successful or $\neq 0$ if it wasn't. This solution requires forking a new process, the COQ compiler. Figure 8 shows the architecture of the connection with COQ.

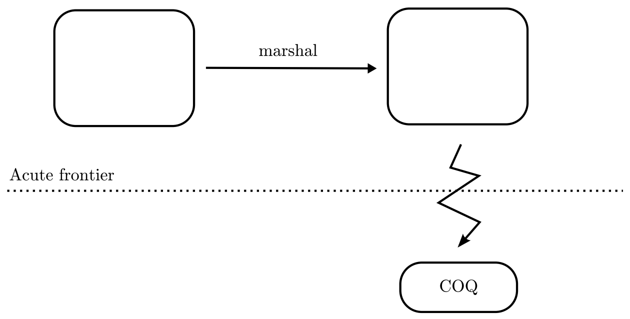

Figure 8. Architecture of the connection with COQ

When writing the file to be sent to COQ we had to take several things into account. First of all, a valid proof had to be constructed with the assertions and proofs that were present in our certificate, including the received value. In the end, this procedure resulted in a concatenation of strings in the file, ordered by some criteria. Following the Figure 9 we provide a description of the process.

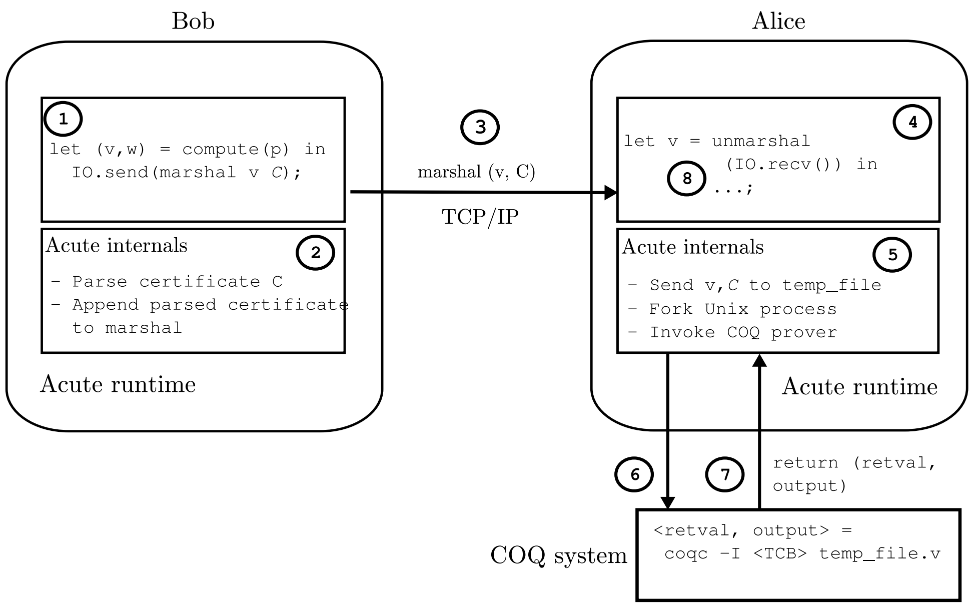

Figure 9. Full proof sequence

In this Figure there are three different parts represented:

– the Acute run-time, responsible of executing Acute source code,
– the Acute internals, and
– COQ.

We describe the process using eight steps. Bob generates some value v with a witness C of that computation, and sends this value to Alice using the marshal primitive. In step (1), there is sample code that *Bob* uses to create the corresponding value v, and its certificate. Next (2), the Acute internal run-time adds the certificate to the value to be marshalled. After that (3), the marshal is transmitted over some network medium (e.g. TCP/IP). Alice, at (4), expects this value after invoking the unmarshal primitive. After the low-level network transport has delivered the message, the Acute internal run-time uses the values received to construct a valid proof sequence to be sent to COQ. This is represented in step (5). Here, a new Unix process forks and the COQ compiler is executed with the specified parameters, the value and its certificate (step (6)). After this fork, the system waits until a response is generated by COQ. When we have a successful compilation of the proof sequence, COQ ends returning a value of 0 (no error), as usual in Unix systems. If any kind of error occurs, the COQ process will returns a value different from 0, and some string output is shown at the standard output of the process (step (7)). We catch this string output, and return it to the Acute run-time, to be used by the language Acute if needed, for example, to raise an exception. This is shown in step (8).

For a complete and detailed presentation of this we refer the interested reader to [Zip08b].

5 Related Works

There are two basic types of work related to ours. First, there are distributed languages with properties similar to the ones in the language we worked with, Acute, that try to solve similar kinds of problems. We will not be exhaustive with programming languages that support typed distributed communications.

In the context of object oriented languages, Java [AG96] boosted the distributed programming paradigm, permitting the execution on lots of hosts by using its virtual machine as a basis. Primitives for doing distributed computations included the notion of marshalling (serialisation). This facility included version identifiers with the class definitions of serialised objects, as we can see in the Remote Method Invocation facility [WRW96]. The main problem for Java programs is that classes are only identified by their (syntactic) name and class loader. This introduces problems when trying to use two classes with the same name but different methods.

For the Microsoft .NET framework, we have languages that rely on the virtual machine that are functional or object oriented. An example of a functional language for .NET is F# [Res07]. This language is like a porting of OCaml for the .NET framework, with some additions. F# was developed as a pragmatically-oriented variant of ML that shares a core language with OCaml. Unlike many type-inferred, statically-typed languages it also supports many dynamic language techniques, such as property discovery and reflection where needed. F# includes extensions for working across languages and for object-oriented programming, and it works seamlessly with other .NET programming languages and tools.

Second, there is work on other techniques, which are probabilistic instead of being a verification technique like the one presented in this work. In this document we used one technique for verification of distributed computations to make them more secure. There are other techniques that introduce the utilisation of probabilistic methodology for detecting possible alterations (or cheating) in the context of grid computations. The work by Wenliang Du et. al [DJMM04] on grid computing and on distributed computations by Phillipe Golle and Ilya Mironov [GM01] are examples of these other techniques. The main approaches are different, and even when they have very low probabilistic numbers of not being caught cheating with the results, they are not one hundred percent reliable. This is, of course, dependent on what kind of adversaries you have, and the threat analysis you made for your system. Following the strategy of checking the results of computations, Grid result checking [GRMR05] focuses on statistical checking.

6 Conclusions and further work

In this section we present our conclusions, together with lines of further work.

We have defined and implemented an infrastructure for result certification. This infrastructure can be used in other places where there is a need for using the proof carrying results technique and not only in this particular case. It is independent from the language, and also, with little or no effort, from the proof assistant used.

There is a successful implementation, for the Acute language, of a solution for the detected problem. This solution uses the infrastructure defined here, and its implementation is available for further testing [Zip08a]. In order to achieve this, we used the technique of proof carrying results, where values transmitted to other hosts carry a witness that proves the computation has been done in a correct way. For the process of verifying that the witness certifies these transmitted values, we used the COQ proof assistant. The proofs are made over these values of abstract types and their properties, carried in a certificate.

An important aspect of the definition of the infrastructure, is that it can be used in other places where there is a need for using the proof carrying results technique and not only in this particular case. It should not be difficult to extend the framework to any other programming language which suffers from similar problems. This means that if we can run a COQ proof assistant, and either 1) we have access to the source code of the programming, or 2) the programming language has an API for interfacing in a similar way with the system; then it should be a reasonable amount of work to add this mechanism to the language/system.

The proof checker is used as a reliable tool to prove that the certificates can be verified correctly. In this step we used the COQ proof assistant, but without interacting with it; only as a verification tool. Any other trustworthy proof checker can be considered. The proofs are made over these values of abstract types and their properties, carried in a certificate. Working with a proof assistant is a good way of delegating the checking process. It is important to use a well-designed, well proven tool for doing this step.

Proof Carrying Results is a promising new approach. The progress on this subject is directly bound to the progress on Certifying Algorithms. This happens because it is difficult to find an algorithm that works for a general case. Despite this consideration, there are many applications of this technique in a number of places, given that we do not trust other hosts. In particular, in global grids where we could have many distributed computations, this technique can obtain good results. Finally, we have made a proof of concept that implementations can be carried to extend systems in a successful way. The source code of the Acute language with support for PCR can be found in [Zip08a]. There are many improvements that can be made that complement the work done up to now, and these are presented in the next section.

There are a number of lines that can be followed starting from this point. Throughout this work we have opened many possible interesting lines of research. We will try to summarise them here:

– Proof Carrying Results can be implemented in other distributed languages, with the same problems that we introduced in Acute. If we have few resources for performing computations (e.g. Java cards, cellular phones, etc.) it is an ideal place to be used. This is because computations can be performed in a server with lots of computing resources, and just checked by the client application, given a clear API for doing result certification.

– For the COQ proof assistant, it could be interesting to have a *Proof* service: this should be a server, that listens in a TCP/IP port and forwards the terms and lemmas received to an executing COQ process.

References

AG96. Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996. ISBN 0-201-63455-4.

BP06. Gilles Barthe and Fernando Pastawsky. Notes on proof carrying results. INRIA Sophia-Antipolis Technical Report, 2006.

DJMM04. Wenliang Du, Jing Jia, Manish Mangal, and Mummoorthy Murugesan. Uncheatable grid computing. In *24th International Conference on Distributed Computing Systems (24th ICDCS'2004)*, pages 4–11, Tokyo, Japan, March 2004. IEEE Computer Society.

dt08. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2008. Version 8.1pl3. Available from: `http://coq.inria.fr` [cited 2008.10.01].

GM01. Philippe Golle and Ilya Mironov. Uncheatable distributed computations. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 425–440. Springer, 2001. ISBN 3-540-41898-9. Available from: `http://link.springer.de/link/service/series/0558/bibs/2020/20200425.htm` [cited 2008.02.20].

GRMR05. Cécile Germain-Renaud and Dephine Monnier-Ragaigne. Grid result checking. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-019-1. `doi:10.1145/1062261.1062280`.

KMMS03. Dieter Kratsch, Ross M. McConnell, Kurt Mehlhorn, and Jeremy P. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 158–167, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0-89871-538-5.

MEK$^+$05. Kurt Mehlhorn, Arno Eigenwillig, Kanela Kanegossi, Dieter Kratsch, Ross McConnel, Uli Meyer, and Jeremy Spinrad. Certifying algorithms (a paper under construction). Max-Planck-Institut für Informatik working paper, 2005. Available from: `http://www.mpi-inf.mpg.de/~mehlhorn/ftp/CertifyingAlgorithms.pdf` [cited 2007.04.24].

NL96. George C. Necula and Peter Lee. Proof-carrying code, October 28 1996. Available from: `http://citeseer.ist.psu.edu/50371.html;http://www.cs.cmu.edu/~necula/tr96-165.ps.gz` [cited 2007.08.30].

Res07. Microsoft Research. F#: A succinct, type-inferred, expressive, efficient functional and object-oriented language for the .net platform, 2007. Available from: `http://research.microsoft.com/fsharp/fsharp.aspx` [cited 2008.10.01].

SLW$^+$04. Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute - high-level programming language design for distributed computation: Design rationale and language definition. Technical report, University of Cambridge and INRIA Rocquencourt, October 2004. Available from: `http://www.cl.cam.ac.uk/users/pes20/acute` [cited 2007.01.04].

WRW96. Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

Zip08a. Felipe Zipitría. PCR extension to the acute source code, 2008. Available from: `http://www.fing.edu.uy/inco/grupos/gsi/sources/acute-pcr/index.html` [cited 2008.10.01].

Zip08b. Felipe Zipitría. Towards secure distributed computations. Master's thesis, nov 2008. Technical Report. Available from: `http://www.fing.edu.uy/~fzipi/tesis/tesis.pdf` [cited 2009.04.01].