RAPPORT DE RECHERCHE

L R I

DESIGN OF A PROOF ASSISTANT : COQ VERSION 7

FILLIATRE J C

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

09/2003

Rapport de Recherche N° 1369

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Design of a proof assistant: Coq version 7

Jean-Christophe Filliâtre

LRI – CNRS UMR 8623
Université Paris-Sud, France
filliatr@lri.fr

October 2000

Abstract

We present the design and implementation of the new version of the Coq proof assistant. The main novelty is the isolation of the critical part of the system, which consists in a type checker for the Calculus of Inductive Constructions. This kernel is now completely independent of the rest of the system and has been rewritten in a purely functional way. This leads to greater clarity and safety, without compromising efficiency. It also opens the way to the "bootstrap" of the Coq system, where the kernel will be certified using Coq itself.

Keywords: Proof assistant, Calculus of Inductive Constructions, Coq

Résumé

Nous présentons la conception et la réalisation de la nouvelle version de l'assistant de preuve Coq. La principale nouveauté réside dans l'isolation de la partie critique du système, qui consiste en un vérificateur de type pour le Calcul des Constructions Inductives. Ce noyau est désormais complètement indépendant du reste du système et a été réécrit de manière purement fonctionnelle. Il en résulte une plus grande clarté et une plus grande sécurité, sans perte d'efficacité. Ceci ouvre également la voie à un possible *bootstrap* du système Coq, où le noyau serait certifié par Coq lui-même.

Mots clés : Assistant de preuve, calcul des constructions inductives, Coq

1 Introduction

In the design of reliable proof assistants, we can distinguish two main approaches:

- The LCF-approach, where theorems belong to an abstract data type and are built using a (small) set of correct functions;
- The proof-checking approach, where derivations can be obtained by any means, but are checked by a small piece of code implementing the rules of the logic.

We are not going to compare the advantages and drawbacks of these approaches in this paper; this is already discussed, with many other aspects of machine-checked proofs, by R. Pollack in [11]. We only notice that both approaches rely on a trusted piece of code. Hence the design and implementation of a proof assistant must be done with care to guarantee the greatest possible reliability.

The Coq system [2] is a proof assistant belonging to the second category. It provides mechanisms to write definitions, statements and to do formal proofs. It can be used interactively or as a batch checker. Its logical formalism is the Calculus of Inductive Constructions (CIC for short), an extension of the original Calculus of Constructions [3] with universes [4, 8] and inductive definitions [10]. The CIC is a typed λ-calculus where, following the Curry-Howard isomorphism, types are seen as propositions and terms as proofs. The proof engine builds terms by means of tactics written in ML. The safety of the Coq system is a consequence of the following two facts:

- The CIC is *consistent* (a proof of strong normalization is given in [13]);
- Once a proof is completed, it is type checked by a *trusted* piece of code implementing the typing rules of the CIC.

The typing rules of the CIC are given in appendix. The reader can check that they are not so numerous. Therefore, it should not be (too) complicated to implement them correctly. However, a proof assistant must also satisfy real world requirements, which include:

- an *interactive* system, which means some imperative state somewhere;
- an *undo* mechanism;
- an *extensible* system, where the user can write tactics in ML, add tables to be used by these tactics, etc;
- *efficiency*, which means that tactics should not spend all their time doing type checking.

During fall 1999, the author designed and implemented a new architecture for the Coq system reconciling a safe and independent kernel with all the above requirements. Then several other people in the Coq team started working on this new basis. In particular, H. Herbelin and D. Delahaye improved several parts of the system. This paper, however, only concentrates on the new architecture of the kernel, whose design and partial re-implementation is the work of the author.

This paper is organized as follows. Section 2 presents the global architecture of the system, and how the kernel forms part of this design. Section 3 presents the design of the kernel itself. Section 4 gives some implementation details. Pieces of code quoted in this paper are given in Objective Caml syntax [9].

2 The big picture

The first implementation of the Calculus of Constructions was realized by G. Huet and T. Coquand in 1984. The first public implementation of the Coq system, version 4.10, was

released in 1989. It was written in Caml, a dialect of ML developed at INRIA. Inductive types were then added by C. Paulin in 1991. In 1993, D. de Rauglaudre ported the Coq system to Caml Light, a new implementation of Caml by X. Leroy and D. Doligez. In 1994–95, C. Murthy designed and implemented a completely new architecture centered on high-level library management, released as version 5.10. Co-inductive types were introduced in this version by E. Giménez [7]. It was ported the next year to Objective Caml [9], the current implementation of Caml. The basic architecture of Coq has not changed since Murthy's reorganization.

The new version of Coq, presented in this paper, is the version 7 of the system.

2.1 Architecture of versions 5.10 and 6.x

The architecture of the previous implementation of the Coq system (versions 5.10 and 6.x) is due to C. Murthy. This implementation satisfies all the "real world requirements". In particular, the undo mechanism is implemented as follows. Any kind of *object* can be declared to the system, with methods[1] to perform its effect on the system, interactively or when loading it from the disk. Global *tables* can be registered with methods to freeze and unfreeze their contents. Then the system keeps a stack of all operations and provides undo by unfreezing previous states and redoing operations up to the backtrack point.

In this context, terms of the CIC are just a particular kind of objects. But this design is *unsafe*: it is possible to delete, to modify or to add constants without any check—at the ML level only, not in the interactive system. Moreover, the type checker code is not clearly isolated. First, there is no data type for typing environments, since CIC objects are spread among all other objects on the stack of operations. Second, the type checker does not come first, since its code relies on all the backtracking mechanism code (to access definitions in the environment).

2.2 An ideal kernel

The critical part of the Coq system consists of the type checker for the CIC. Indeed, once the proof of a lemma p is completed, the corresponding λ-term t is type checked and a new constant $p := t$ is added in the environment. To guarantee the consistency of the environment, the type checking of this definition has to be correct. Therefore, the kernel of the system should ideally provide an abstract data type for well formed typing environments, together with functions to insert new elements in a safe way. The signature of such a kernel would look like

```
type safe_env
val empty : safe_env
val push_var : safe_env → identifier × constr → safe_env
val add_constant : safe_env → constant_declaration → safe_env
...
```

[1] It is not implemented in an object oriented way, strictly speaking, since Caml did not support object oriented programming by the time of this implementation. However, C. Murthy's code is really simulating objects, using records containing functions.

```
val lookup_var : safe_env → identifier → constr
...
```

where `constr` is the data type of CIC terms. Notice that the kernel does not have to provide a typing function: typing is done internally when insertion functions are called. Only the invariant of having a well formed typing environment is needed, and this can be maintained if the above data type is abstract.

2.3 Coq version 7: a reconciliation

In Coq version 7, we provide the best of both worlds: an ideal kernel on one hand, as described in the previous section, and Murthy's general backtracking mechanism on the other. We proceed as follows:

1. First, we write a purely functional type checker for the CIC;
2. Second, we introduce the backtracking mechanism, but CIC terms are no longer a particular kind of object;
3. Finally, we reconcile both of them by declaring a *reference* on the current typing environment as a global table.

If `Safe` is the kernel module implementing the signature sketched in Section 2.2, then the code realizing the connection looks like

```
let r = ref Safe.empty

let push_var (id,c) = r := Safe.push_var !r (id,c)
let add_constant cd = r := Safe.add_constant !r d
...
let lookup_var id = Safe.lookup_var !r id
...
```

In order to have the typing environment correctly backtracked during undo operations, we declare the above reference `r` as a global table. This is particularly simple, since typing environments are implemented with a purely functional data type: freezing and unfreezing the value of `r` is then immediate.

```
let freeze () = !r
let unfreeze f = r := f
let _ = declare_table "typing env" freeze unfreeze
```

For all the rest of the system, there is no visible difference with respect to previous implementations. Whatever the implementation of the typing environment is, an imperative environment is still provided with the following signature:

```
val push_var : identifier × constr → unit
val add_constant : constant_declaration → unit
...
val lookup_var : identifier → constr
...
```

And indeed, all the upper parts of the system were reestablished almost "as is".

3 Design of the kernel

As we explained in the previous section, we succeeded in isolating the critical part of the Coq system, that is a type checker for the CIC. For greater clarity and safety, it is implemented in a purely functional way, following the signature sketched in Section 2.2.

3.1 Architecture of the kernel

The typing rules for CIC are given in appendix. If we try to implement them in a direct (and naive) way, we immediately face a problem of circular dependency:

Adding a term in an environment requires to type check it (rule Push)

and

Type checking a term requires to lookup in the environment (rule Var)

With definitions and inductive types, the definition of the convertibility also needs access to the environment, hence introducing another circularity issue. Defining the environment operations, the conversion and the typing function in a mutually recursive way would not be a good design. It would result in a single module with thousands of lines of code. Instead, we choose the following three steps solution:

1. First, we define a data type `env` for typing environments. This type is "unsafe": it may contain values that do not represent well formed environments, since it does not perform type checking on the elements which are inserted.

2. Then we define the conversion $=_c$ and the typing rules over this data type `env`. These functions make assumptions about the well formedness of the typing environments they take as arguments.

3. Finally, in a single module `Safe`, we define the type of typing environment `safe_env`, we write the type checking algorithm and we define the operations over type `safe_env` so that they perform type checking. Internally, the type `safe_env` is defined as equal to the type `env` but it is abstract (its identity to `env` is not exported in the interface of module `Safe`) and therefore we maintain the invariant that all the values of type `safe_env` are well formed. In other words, any value Γ of type `safe_env` is such that $\mathcal{WF}(\Gamma)$ holds.

To summarize, the kernel has the following structure:

1.	Data type for terms
2.	Data type for environments
3.	Typing rules / Conversions
4.	Safe environments (module `Safe`)

In the next section, we give implementation details about the various parts of the kernel.

3.2 Implementation of the kernel

Terms. Terms of the CIC are encoded using explicit names for global objects (constants, inductive types and constructors) and de Bruijn's indices for bound variables. The type `constr` for CIC terms is an abstract type, for easier modifications of this type. A view [12] on type `constr` is provided to allow pattern matching on values of this type[2].

Universes. Coq implements the Extended Calculus of Constructions with universes (ECC), following the work initiated by T. Coquand [4] and studied in the Ph.D. of Z. Luo [8]. The typing rules given in the appendix assume that explicit universes are statically assigned indices. However, this would be a drawback in practice, since this would often necessitate a repositioning of indices each time type checking requires the insertion of a new universe sort between two existing ones. Therefore, Coq uses instead "floating universes": new universes are created when needed and inserted in a dependency graph which is checked to be acyclic. (If necessary, universes indices as used in the typing rules could be retrieved at any time using a topological sort of the universes graph.) The data types for universes and universes graphs are functional and abstract. In order to save space and time, three universes are statically created which are a type for Set and Prop, a type for this type and a type for this latter type. Hence we get the following signature:

```
type universe
val prop_univ : universe
val prop_univ_univ : universe
val prop_univ_univ_univ : universe
val new_univ : unit → universe

type universes
val initial_universes : universes
```

For efficiency of the type checking, we do not introduce relations between universes one by one in graphs, but we use sets of constraints.

```
type constraint_type = Gt | Geq | Eq
type constraint = universe × constraint_type × universe
```

[2]Ocaml does not support views but this is achieved using subtleties of abbreviation and abstraction in the Ocaml type system.

```
type constraints
val empty_constraints : constraints
val add_constraint : constraint → constraints → constraints
```

Then constraints over universes are merged into graphs using the following function `merge_constraint` It raises the exception `UniverseInconsistency` if a cycle occurs.

```
exception UniverseInconsistency
val merge_constraints : constraints → universes → universes
```

The functional implementation of universes graphs and the introduction of sets of constraints constitute a novelty of Coq version 7.

Typing environments. The data type `env` for typing environments is purely functional. It contains mappings from names (explicit names for constants or de Bruijn indices) to their types and/or values and a universes graph. It uses efficient data structures for dictionaries from the Ocaml standard library (balanced trees). The type `env` is kept abstract, so that its implementation can be easily changed. The main point is that it is reused in many other parts of the system, as well as many functions over it like reduction functions, where safety is not involved. Therefore, most of the operations of the system are efficient, yet using the same data type as the safe kernel.

The signature of the module `Env` implementing typing environments looks like:

```
type env
val empty : env
val push_var : env → identifier × constr → env
val add_constant : env → constant_declaration → env
...
val lookup_var : env → identifier → constr
...
val push_constraints : env → constraints → env
...
```

Notice that this signature is similar to the expected signature for the kernel itself; the only (big) difference is that values of type `env` do not necessarily represent well formed environments.

Reductions and conversions. Reductions are functions from `constr` to `constr` taking an environment as argument. Indeed, the environment is needed to expand definitions or to find information related to inductive definitions. Therefore, reduction functions have the following profile:

```
type reduction_function = env → constr → constr
```

Examples of such functions are for instance

```
val whd_betadeltaiota : reduction_function
val nf_betaiota : reduction_function
```

Conversion functions, implementing $=_c$ and $\leq_c$, do not simply return a boolean but a set of constraints; Indeed, enforcing the convertibility of two terms may generate relations between some universes. Therefore, conversion functions have the following type:

```
type conversion_function = env → constr → constr → constraints
```

Then $=_c$ and $\leq_c$ are implemented with the above profile:

```
val conv : conversion_function
val conv_leq : conversion_function
```

Typing rules and typing algorithm. A data type `judgment` is introduced to represent a term with its type. Then typing rules are implemented as functions taking an environment, some judgments and returning a new judgment together with a set of constraints. The interpretation is that, if the environment is well formed and the given judgments derivable in this environment, then the new judgment is derivable in the environment extended with the new constraints. Notice that the typing rules are introduced at that point independently of the typing algorithm.

Then we can implement the typing algorithm itself in a separate module having the following signature:

```
val typecheck : env → constr → judgment × constraints
```

Main module. Finally, a last module `Safe` realizes the signature given in Section 2.2, in the following way:

```
type safe_env = Env.env

let empty = Env.empty

let push_var e (id,c) =
  let (_,cst) = typecheck e c in
  Env.push_var (Env.push_constraints e cst) (id,c)

let add_constant e d =
  ...
```

Notice how the Ocaml type system is used. Internally to module `Safe`, the type definition `type safe_env = Env.env` introduces `safe_env` as an alias for type `Env.env`. Hence functions over type `Env.env`, like `Env.push_var`, can be used in the definition of functions over type `safe_env`. But the abbreviation `safe_env = Env.env` is not exported in the interface of module `Safe`. Therefore, the type `safe_env` appears externally as a "new" abstract data type.

Part	Description	Modules	Lines of code
Lib	Utility libraries	15	1700
Kernel	Type checker for CIC	18	7800
Library	Undo mechanism and modules	14	2000
Pretyping	Translation from AST to terms	12	2000
Parsing	Parsing and pretty-printing	11	3700
Proofs	Proof engine	11	4000
Tactics	Basic tactics and tacticals	14	4000
Toplevel	Assembling layer	8	2900
	Total	103	28100

Figure 1: The main parts of Coq version 7

4 Implementation details

The Coq system is written in Objective Caml [9], a dialect of ML developed at INRIA. This language provides functional, imperative and object oriented programming styles and is equipped with a powerful module calculus which offers true separate compilation. Two compilers are available: one producing portable bytecode, allowing fast development and easy debugging, and one producing fast executables in native code. Coq heavily uses the rich and portable standard library of Ocaml. Coq also uses the Camlp4 tool [5], which provides data types for dynamic grammars, thus allowing a user-extensible grammar.

The code of Coq is documented using a literate programming tool, Ocamlweb [6]. A single human-readable document is produced from the code, which describes the design and all the interfaces (types, functions with their specifications, exceptions, etc.). This document is 110 page long—but several modules are not yet fully documented—and a 30 page long index is automatically appended by Ocamlweb, giving the definition and use points of all the identifiers of the code.

The total redesign and (partial) re-implementation of the system was realized in less than four months. The whole code is roughly 30,000 lines long, 10,000 being dedicated to the kernel and the undo mechanism. The main parts of Coq version 7 are given in Figure 1. For each, the number of Ocaml modules and the number of lines of code is given. These parts are presented in the order of linking.

5 Conclusion

We have presented the new implementation of the Coq system. It combines the efficiency and safety requirements in a completely new design, where the critical part of the system, a type checker for the CIC, is clearly isolated.

This type checker is now written is a purely functional way. It uses efficient functional data structures and is even slightly faster than the previous type checkers that were partly imperative. Being functional, this critical kernel is now easier to maintain and to reason

about. One can now think of formally certifying it. Following the work of B. Barras [1], it could even be "bootstrapped" *i.e.* certified by Coq itself.

Anyhow, we believe that this new implementation already constitutes the safest of the popular proof assistants.

References

[1] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[2] The Coq Proof Assistant. `http://coq.inria.fr/`.

[3] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[4] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First Symposium on Logic in Computer Science*, Cambridge, MA, June 1986. IEEE Comp. Soc. Press.

[5] D. de Rauglaudre. The Camlp4 Pre Processor. `http://caml.inria.fr/camlp4/`.

[6] J.-C. Filliâtre and C. Marché. Ocamlweb, a literate programming tool for Objective Caml. `http://www.lri.fr/~filliatr/ocamlweb/`.

[7] Eduardo Giménez. Codifying guarded definitions with recursives schemes. In *Proc. of the 1994 Workshop on Types for Proofs and Programs, LNCS 996*, pages 39–59, December 1994. An extended version is available as LIP research report no. 95-07.

[8] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[9] The Objective Caml language. `http://www.ocaml.org/`.

[10] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993. Also LIP research report 92-49.

[11] Robert Pollack. How to Believe a Machine-Checked Proof. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, Oxford Logic Guides. Clarendon Press, August 1998.

[12] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[13] B. Werner. *Une Théorie des Constructions Inductives.* Thèse de doctorat, Université de Paris VII, Mai 1994.

Appendix: Typing rules for the ECC

Although the Coq system implements the full CIC with definitions, (co)inductive types and definitions are omitted here for the simplicity of this paper—the reader must be aware, however, that positivity of inductive types and guardedness of (co)recursive definitions constitute a nontrivial part of the actual kernel. Thus we give here the typing rules for the Extended Calculus of Constructions [8]. Sorts, terms and typing environments obey the following grammar:

Sorts ($\mathcal{S}$)	s	::=	$\mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type}(i)$
Terms	t	::=	$s \mid x \mid [x:t]t \mid (x:t)t \mid (t\ t)$
Environments	Γ	::=	$[] \mid \Gamma, x:t$

Typing rules are given in Figure 2. The convertibility between terms, written $=_c$, is here the $\alpha\beta$-equivalence. In the general case, it is the $\alpha\beta\delta\iota$-equivalence, δ being the unfolding of definitions and ι the reduction associated to inductive types.

Empty
$$\frac{}{\mathcal{WF}([])}$$

Push
$$\frac{\Gamma \vdash t : s \quad s \in \mathcal{S} \quad x \notin \Gamma}{\mathcal{WF}(\Gamma, x : t)}$$

Axioms
$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(i)} \qquad \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}(i)} \qquad \frac{\mathcal{WF}(\Gamma) \quad i < j}{\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(j)}$$

Var
$$\frac{\mathcal{WF}(\Gamma) \quad x : t \in \Gamma}{\Gamma \vdash x : t}$$

Prod
$$\frac{\Gamma \vdash t : s_1 \quad \Gamma, x : t \vdash u : s_2 \qquad s_2 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\Gamma \vdash (x : t)u : s_2}$$

$$\frac{\Gamma \vdash t : \mathsf{Type}(i) \quad \Gamma, x : t \vdash u : \mathsf{Type}(j) \quad i \leq k \quad j \leq k}{\Gamma \vdash (x : t)u : \mathsf{Type}(k)}$$

Lam
$$\frac{\Gamma \vdash (x : t)u' : s \quad \Gamma, x : t \vdash u : u'}{\Gamma \vdash [x : t]u : (x : t)u'}$$

App
$$\frac{\Gamma \vdash u : (x : t')u' \quad \Gamma \vdash t : t'}{\Gamma \vdash (u\ t) : u'[x \leftarrow t]}$$

Conv
$$\frac{t =_c u}{t \leq_c u} \qquad \frac{i \leq j}{\mathsf{Type}(i) \leq_c \mathsf{Type}(j)} \qquad \frac{s \in \{\mathsf{Prop}, \mathsf{Set}\}}{s \leq_c \mathsf{Type}(i)} \qquad \frac{t =_c u \quad m \leq_c n}{(x : t)m \leq_c (x : u)n}$$

$$\frac{\Gamma \vdash u' : s \quad \Gamma \vdash t : t' \quad t' \leq_c u'}{\Gamma \vdash t : u'}$$

Figure 2: Typing rules for the Extended Calculus of Constructions

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1345	FLANDRIN E LI H WEI B	A SUFFICIENT CONDITION FOR PANCYCLABILITY OF GRAPHS	16 PAGES	01/2003
1346	BARTH D BERTHOME P LAFOREST C VIAL S	SOME EULERIAN PARAMETERS ABOUT PERFORMANCES OF A CONVERGENCE ROUTING IN A 2D-MESH NETWORK	30 PAGES	01/2003
1347	FLANDRIN E LI H MARCZYK A WOZNIAK M	A CHVATAL-ERDOS TYPE CONDITION FOR PANCYCLABILITY	12 PAGES	01/2003
1348	AMAR D FLANDRIN E GANCARZEWICZ G WOJDA A P	BIPARTITE GRAPHS WITH EVERY MATCHING IN A CYCLE	26 PAGES	01/2003
1349	FRAIGNIAUD P GAURON P	THE CONTENT-ADDRESSABLE NETWORK D2B	26 PAGES	01/2003
1350	FAIK T SACLE J F	SOME b-CONTINUOUS CLASSES OF GRAPH	14 PAGES	01/2003
1351	FAVARON O HENNING M A	TOTAL DOMINATION IN CLAW-FREE GRAPHS WITH MINIMUM DEGREE TWO	14 PAGES	01/2003
1352	HU Z LI H	WEAK CYCLE PARTITION INVOLVING DEGREE SUM CONDITIONS	14 PAGES	02/2003
1353	JOHNEN C TIXEUIL S	ROUTE PRESERVING STABILIZATION	28 PAGES	03/2003
1354	PETITJEAN E	DESIGNING TIMED TEST CASES FROM REGION GRAPHS	14 PAGES	03/2003
1355	BERTHOME P DIALLO M FERREIRA A	GENERALIZED PARAMETRIC MULTI-TERMINAL FLOW PROBLEM	18 PAGES	03/2003
1356	FAVARON O HENNING M A	PAIRED DOMINATION IN CLAW-FREE CUBIC GRAPHS	16 PAGES	03/2003
1357	JOHNEN C PETIT F TIXEUIL S	AUTO-STABILISATION ET PROTOCOLES RESEAU	26 PAGES	03/2003
1358	FRANOVA M	LA "FOLIE" DE BRUNELLESCHI ET LA CONCEPTION DES SYSTEMES COMPLEXES	26 PAGES	04/2003
1359	HERAULT T LASSAIGNE R MAGNIETTE F PEYRONNET S	APPROXIMATE PROBABILISTIC MODEL CHECKING	18 PAGES	01/2003
1360	HU Z LI H	A NOTE ON ORE CONDITION AND CYCLE STRUCTURE	10 PAGES	04/2003
1361	DELAET S DUCOURTHIAL B TIXEUIL S	SELF-STABILIZATION WITH r-OPERATORS IN UNRELIABLE DIRECTED NETWORKS	24 PAGES	04/2003
1362	YAO J Y	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	72 PAGES	07/2003
1363	ROUSSEL N EVANS H HANSEN H	MIRRORSPACE : USING PROXIMITY AS AN INTERFACE TO VIDEO-MEDIATED COMMUNICATION	10 PAGES	07/2003

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1364	GOURAUD S D	GENERATION DE TESTS A L'AIDE D'OUTILS COMBINATOIRES : PREMIERS RESULTATS EXPERIMENTAUX	24 PAGES	07/2003
1365	BADIS H AL AGHA K	DISTRIBUTED ALGORITHMS FOR SINGLE AND MULTIPLE-METRIC LINK STATE QoS ROUTING	22 PAGES	07/2003