

# **SPECIFICATION AND VERIFICATION OF BLOCK WORLD DOMAIN IN COQ**

## **A DISSERTATION**

*Submitted in partial fulfillment of the  
requirements for the award of the degree  
of*

### **INTEGRATED DUAL DEGREE**

**(Bachelor of Technology & Master of Technology)**

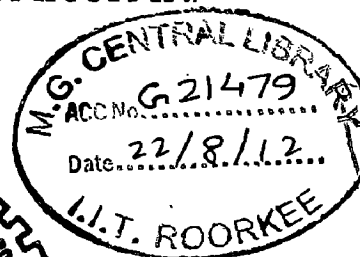
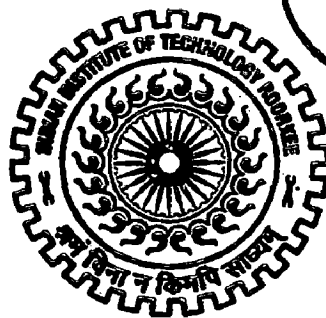
**in**

### **COMPUTER SCIENCE AND ENGINEERING**

**(With Specialization in Information Technology)**

**By**

**GAJRAJ SINGH TANWAR**



**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE  
ROORKEE - 247 667 (INDIA)**

**MAY, 2012**

## CANDIDATE'S DECLARATION

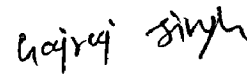
---

I hereby declare that the work being presented in the dissertation work entitled “**Specification and Verification of Block World Domain in Coq**” towards the partial fulfillment of the requirement for the award of the degree of **Integrated Dual Degree in Computer Science and Engineering (with specialization in Information Technology)** and submitted to the **Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, India** is an authentic record of my own work carried out during the period from May, 2011 to May, 2012 under the guidance and provision of **Dr. Rajdeep Niyogi, Assistant Professor, Department of Electronics and Computer Engineering, IIT Roorkee.**

I have not submitted the matter embodied in this dissertation work for the award of any other degree and diploma.

Date: 21 May, 2012

Place: Roorkee



(GAJRAJ SINGH TANWAR)

## CERTIFICATE

---

This to certify that the declaration made by the candidate above is correct to the best of my knowledge and belief.

Date: 22 May, 2012

Place: Roorkee



**Dr. Rajdeep Niyogi**  
Assistant Professor  
E&CE Department  
IIT Roorkee, India

## ACKNOWLEDGEMENTS

---

I would like to take this opportunity to extend my heartfelt gratitude to my guide and mentor **Dr. Rajdeep Niyogi**, Assistant Professor, Department of Electronics and Computer Engineering, Indian Institute of Technology Roorkee, for his trust in my work, his able guidance, regular source of encouragement and assistance throughout this dissertation work. I would state that the dissertation work would not have been in the present shape without his inspirational support and I consider myself fortunate to have done my dissertation under him.

I also extend my sincere thanks to **Dr. Padam Kumar**, Professor and Head of the Department of Electronics and Computer Engineering for providing facilities for the work.

I would like to thank all my friends who supported and encouraged me to finish this work.

Finally, I would like to say that I am indebted to my parents for everything that they have given to me. I thank them for sacrifices they made so that I could grow up in a learning environment. They have always stood by me in everything I have done, providing constant support, encouragement, and love.

*Gajraj Singh*

**GAJRAJ SINGH TANWAR**

## ABSTRACT

---

Automated Theorem Proving (ATP) or Automated Deduction, currently the most well-developed subfield of Automated Reasoning (AR), is the proving of mathematical theorems by a computer program. Depending on the underlying logic, the problem of deciding the validity of a formula varies from trivial to impossible. A simpler, but related, problem is proof verification, where an existing proof for a theorem is certified valid. For this, it is generally required that each individual proof step can be verified by a primitive recursive function or program, and hence the problem is always decidable. Interactive Theorem Prover requires a human user to give hints to the system. Depending on the degree of automation, the prover can essentially be reduced to a proof checker, with the user providing the proof in a formal way, or significant proof tasks can be performed automatically. Interactive prover's are used for a variety of tasks, but even fully automatic systems have proven a number of interesting and hard theorems. There are several types of theorem prover like Coq, which have its own specification languages, in which we can specify a system in a logical form with the help of predicates and write the theorem related to the properties of the system. These can be verified with the help of proof assistant.

## LIST OF FIGURES

---

Figure 2.1	Theorem Proving as a Search.....	6
Figure 2.2	Backwards Proof.....	8
Figure 4.1	Predicates in Coq .....	22
Figure 4.2	Two Block World System .....	24
Figure 4.3	Outline of SwapAB Proof.....	25
Figure 4.4	Coq script for Theorem SwapAB.....	26
Figure 5.1	Working of Coq as Proof Assistant in cmd prompt.....	28
Figure 5.2	Two Block System Coq Specification.....	29

## LIST OF TABLES

---

Table 4.1	Coq syntax for ILL Operators.....	20
-----------	-----------------------------------	----

---

# CONTENTS

---

<b>ABSTRACT.....</b>	<b>iii</b>
<b>LIST OF FIGURES.....</b>	<b>iv</b>
<b>LIST OF TABLES.....</b>	<b>iv</b>
<b>CHAPTER 1: INTRODUCTION.....</b>	<b>1</b>
1.1 Theorem Proving .....	1
1.2 Problem Statement.....	2
1.2.1 Problem Description .....	2
1.3 Organization of Dissertation .....	2
<b>CHAPTER 2: LITERATURE REVIEW .....</b>	<b>4</b>
2.1 Types of ATP Systems.....	4
2.2 Architecture .....	5
2.2.1 A Subsumption Architecture for Theorem Proving.....	5
2.2.2 Constructive Proof.....	7
2.3 Interactive Theorem Prover –Coq.....	9
2.3.1 Coq .....	9
2.3.2 An Overview of specification language of coq GALLINA .....	10
2.3.3 Introduction to the Proof Engine.....	13

<b>CHAPTER 3: LANGUAGE DEFINITION AND PROPOSED SCHEME.....</b>	<b>15</b>
3.1 Linear Logic.....	15
3.2 The Blocks World.....	17
3.2.1 Relationship Predicates over Blocks.....	17
3.2.2 Status Predicates of Robot Arm:.....	18
<b>CHAPTER 4: IMPLEMENTATION DETAILS.....</b>	<b>19</b>
4.1 Codify Intuitionistic Linear Logic (ILL).....	10
4.2 The Linear consequence Operator:.....	20
4.3 Block world codify	
4.3.1 Predicates.....	22
4.3.2 Basic Actions.....	22
4.2.3 Special Cases.....	23
4.4 Two Block World Domain.....	24
<b>CHAPTER 5: RESULT AND DISCUSSION.....</b>	<b>27</b>
<b>CHAPTER 6: CONCLUSION.....</b>	<b>30</b>
<b>REFERENCES.....</b>	<b>31</b>

# CHAPTER 1

## INTRODUCTION

---

### 1.1 Theorem Proving

Theorem Proving (TP) deals with the development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms and hypotheses). It is concerned with the development and use of systems that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. These systems are capable of solving the non-trivial problems. However the search complexity of most interesting problems is enormous, and many problems cannot currently be solved within realistic resource limits. TP systems are used in wide variety of problems.

Theorem proving is mainly of two types. i.e.

1. Automated Theorem Proving
2. Interactive Theorem Proving

The Automated Theorem Proving is also called Automated Deduction is the most well developed subfield of automated reasoning, is the proving of mathematical theorem by a computer program. In this proving depending on the type of underlying logic the problem of deciding the validity of formula varies from trivial to impossible. Since the proofs generated by Automated Theorem Prover's are typically very large, the problem of proof compression is crucial and various techniques aiming at making the prover's output smaller, and consequently more easily understandable and checkable, have been developed.

Interactive Theorem Proving is that which require a human user to give hints to the system. The focus of Interactive Theorem Proving is to present in a fully formalized way, all the axioms, definitions, computations and proofs within any mathematical subject. While the proofs themselves come from humans, the formalizations are meant to be done in such a way that a computer can verify the correctness of the claims. At present there are two major areas of applications of Interactive Theorem Proving. One of them is formalization of mathematics and its proofs, while the other is formal verification of computer systems.



Chapter 3 provides a detailed description of the language definition and mathematical definition of block world.

Chapter 4 gives the brief description of the implementation of the proposed scheme.

Chapter 5 discusses the results and in Chapter 6 concludes the work.

**Soundness:** Sound ATP systems return only correct solutions to problems. Soundness can be assumed for extensively tested systems, at least if a large number of the solutions produced have been verified. Unsound systems are useless, and soundness is therefore required.

**Completeness:** Complete systems always return a solution if one exists. An ATP system may be known to be incomplete from a theoretical perspective, due to known weaknesses in the calculus or search strategy, due to bugs in the implementation.

**Solutions:** There are various responses that an ATP system may make when given a problem. A solution (a proof or a model) may be returned, or the system may give only an assurance that a solution exists. There are some ATP systems that conserve resources by not building a solution during their search, e.g., completion based systems do not build a proof. Such systems retain only enough information to build a solution later if required.

**Resource Limit:** There are three resources used by ATP systems that are of interest, and whose usage should be evaluated. They are CPU time, wall clock time, and memory.

From a user perspective, if there is limited CPU time available, it is of interest to know whether or not an ATP system on average finds its solutions within that CPU time limit. Average CPU time over problems solved is therefore an interesting measure for evaluation.

## 2.2 ARCHITECTURE

1. **A Subsumption Architecture for Theorem Proving:** Work on Automatic Theorem Proving goes back to the earliest days of Artificial Intelligence. A theorem prover for propositional logic was one of the first AI programs. The roots of the field are in mathematical logic. Most early work was based on a theorem of the logician Herbrand, which implicitly defined a procedure for exhaustively searching for a proof[2].

Both resolution theorem proving and many of the alternative mechanisms can be seen as manipulating an initial conjecture by the application of rules of inference. Each rule breaks the conjecture into one or more subgoals. Rules can then be applied to each of these

Search is represented as exploration of a tree with the root at the top and the leaves at the bottom. The root is labeled with the original conjecture. Each rule applying to this conjecture is represented as a circle connected to the conjecture node. Each of these rules gives rise to a (possibly empty) set of subgoals. Each subgoal is represented as a square connected to the rule. Further rules apply to each subgoal, creating further subgoals, etc.

Most work in automatic theorem proving has been directed to controlling the combinatorial explosion.

(a) Resolution-based systems have been refined so that fewer rules apply to each goal or so that each rule application makes more progress.

(b) Alternatives to resolution, requiring fewer searches, have been invented, e.g. term rewriting.

(c) Heuristics have been devised to select the most promising rule applications first or to decide that some rule applications should not be tried at all.

(d) More efficient means of storing and applying rules have been devised so that the effect of the combinatorial explosion is reduced.

(e) Interactive Systems have been developed in which a human user helps to direct the search.

Proving theorems in most areas of mathematics is a semi-decidable problem. This means that: if the conjecture is a theorem then a complete theorem prover will eventually find a proof (though this could take a very long time); if the conjecture is not a theorem then a complete theorem prover might never terminate in its fruitless search for a proof.

**2. Constructive Proof:** A task is a "structured set of activities in which actions are undertaken in some sequence", in order to achieve the user's goals. We decide the user's goal to be formally proving some theorem, by some method of constructive proof.[3]

A constructive proof can be thought of as a tree of statements. At the root node is the subject of the proof: a statement of the theorem. Leaf nodes contain either axioms, or previously proven theorems. Internal nodes (nodes that aren't leaf nodes) are the result of applying inference rules to axioms and theorems, and are theorems themselves. The conjunction of child nodes implies the parent node.

## 2.3 Interactive Theorem Prover –Coq

In this work we have used theorem prover coq which is a formal proof management system. So a brief description of it required.

1. **Coq:** Coq is a formal proof assistant in higher order logic, allowing the development of the computer program in consistent with their specification. It has three main aspects: the logical language in which one can write axiomatizations and specification, the proof assistant which allows the development of verified mathematical proofs, and the program extractor which synthesizes computer programs obeying their formal specifications, written as logical assertions in the language. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. The COQ system is designed to develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct with respect to their specification. It provides a specification language named GALLINA. It also provides an interactive proof assistant to build proofs using specific programs called tactics [4].

### 2. An Overview of specification language of coq GALLINA

GALLINA is the specification language of the proof assistant Coq. It allows to develop mathematical theories and to prove specifications of programs. The theories are built from axioms, hypotheses, parameters, lemmas, theorems and definitions of constants, functions, predicates and sets.[5]

#### a) Lexical Conventions in GALLINA

**Blanks Space:** newline and horizontal tabulation are considered as blanks. Blanks are ignored but they separate tokens.

**Comments:** Comments in COQ are enclosed between (\* and \*), and can be nested. They can contain any character. However, string literals must be correctly closed. Comments are treated as blanks.

**Identifiers and access identifiers:** Identifiers, written ident, are sequences of letters, digits, \_ and ', that do not start with a digit or '.

**Natural numbers and integers:** Numerals are sequences of digits. Integers are numerals optionally preceded by a minus sign.

Pos<sub>n</sub> is assumed

Here `gt` is a function expecting two arguments of type `_nat` in order to build a logical proposition

*b) Definitions*

The initial prelude contains a few arithmetic definitions: `nat` is defined as a mathematical collection (type `Set`), constants `O`, `S`, `plus`, are defined as objects of types respectively `nat`, `nat`→`nat`, and `nat`→`nat`→`nat`. You may introduce new definitions, which link a name to a well-typed value. For instance, we may introduce the constant `one` as being defined to be equal to the successor of zero:

```
Coq < Definition One := (S O).
```

`One` is defined

Here is a way to define the doubling function, which expects an argument `m` of type `nat` in order to build

Its result as `(plus m m)`.

```
Coq < Definition double (m: nat) := plus m m.
```

`double` is defined.

*c) Tactics*

Tactics [5]-[6] are the specific programs using which Coq interactive proof assistant build proofs. A basic set of tactics was predefined, which the user could extend by his own specific tactics.

A deduction rule is a link between some (unique) formula, that we call the conclusion and (several) formulas that we call the premises. Indeed, a deduction rule can be read in two ways. The first one has the shape: —if I know this and this then I can deduce this|. For instance, if I have a proof of `A` and a proof of `B` then I have a proof of `A ^ B`. This is forward reasoning from premises to conclusion. The other way says: —to prove this I have to prove this and this. For instance, to prove `A ^ B`, I have to prove `A` and I have to prove `B`.

This is backward reasoning which proceeds from conclusion to premises. We say that the conclusion is the goal to prove and premises are the subgoals. The tactics implement backward reasoning. When applied to a goal, a tactic

If the goal is neither a product nor starting with a let definition, the tactic intro applies the tactic red until the tactic intro can be applied or the goal is not reducible.

#### 4. apply term

This tactic applies to any goal. The argument term is a term well-formed in the local context. The tactic applies tries to match the current goal against the conclusion of the type of term. If it succeeds, then the tactic returns as many subgoals as the number of non dependent premises of the type of term. If the conclusion of the type of term does not match the goal and the conclusion is an inductive type isomorphic to a tuple type, then each component of the tuple is recursively matched to the goal in the left-to-right order.

#### 5. assert (ident: form)

This tactic applies to any goal. assert (H: U) adds a new hypothesis of name H asserting U to the current goal and opens a new subgoal U. The subgoal U comes first in the list of subgoals remaining to prove.

### 3. Introduction to the Proof Engine

In the following, we are going to consider various propositions, built from atomic propositions A; B; C. This may be done easily, by introducing these atoms as global variables declared of type Prop. It is easy to declare several names with the same specification. [7]

```
Coq < Variables A B C : Prop.
```

We shall consider simple implications, such as  $A \rightarrow B$ , read as A implies B . Let us now embark on a simple proof. We want to prove the easy tautology  $((A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C))$  . We enter the proof engine by the command Goal, followed by the conjecture we want to verify:

```
Coq < Goal (A -> B -> C) -> (A -> B) -> A -> C.
```

```
1 subgoal
```

```
=====
```

```
(A -> B -> C) -> (A -> B) -> A -> C.
```

# CHAPTER 3

## Language Definition and Proposed Scheme

---

This chapter gives the details about the logic system we have used. And the mathematical definition of blocks word.

### 3.1 Linear Logic

Linear Logic is often described as a 'resource-conscious' logic. In the context of mathematical logic, we can consider a proposition to represent a resource of some kind [8]. The logical formulations provide us structural rules to manipulate assumptions: the weakening and contraction.

**Weakening:** this rule says that if from a collection of assumptions  $\Gamma$  we can conclude B, and then certainly from the assumptions  $\Gamma$  and A, we can conclude B.

**Contraction:** The Contraction rule says that if we need an assumption A twice to conclude B, then we can simplify this to A, as A and A is morally the same as A.

Now if we take a resource view of these two rules with a different way. The weakening rules amounts to saying that we might not need a resource after all and the Contraction rule tells us that we might need a resource any number of times.

So Linear logic is a sub-structural logic [9] in that it rejects the use of two of the structural rules of the classical logic, specifically weakening and contraction. Basically weakening allow us to have unused hypothesis, while contraction allows us to disregard the number of times a hypothesis is used.

Thus when we write a deduction in linear logic of the form  $\Gamma \multimap A$  we are stating that all the assumptions in  $\Gamma$  are used exactly used in once in the deduction of A. For this reason the linear logic is also said resource sensitive. [6] We can also say that  $\Gamma \multimap A$  represents a process that consumes the resources of  $\Gamma$  in the production of A.

With the same way in classical disjunction  $A \vee B$  can be established by proving either or both A and B. So in linear logic we write

- $A \oplus B$  for the disjunction where just one of A or B has been proven.
- $A \& B$  for the disjunction where both have been established.

So the linear implication now internalizes the linear deduction and in this way  $A \multimap B$  represents a process whose use will consume a A and produce a B.

### 3.2 The Blocks world

Blocks world is one of the famous domains in planning. The property of this domain is that it consist a set of blocks sitting on the table. The blocks can be stacked but condition is that only one block can fit directly on top of another. In the system there is a robot arm which can pick up a block and move it another position. Block can be placed on the table or on the top of another block [10].

The robot arm can pick up only one block at a time that has nothing on top of it i.e. block's top must be clear so that robot arm can pick it. The robot arm can't pick up a block that has another block on top of it. In this system goal will be build one or more stacks of blocks specified in term of that what blocks are on top of what other block. For example, a goal might be to get block A on B and block C on D.

As we see the block world state at a particular time depends on the relative position of the blocks. So it is a state based system which has goal and change in state can be represented as a transitions.

As we see in block world we assume the existence of some finite set of blocks as well as a robot arm which can be used to move them. The actions of the arm involve only stacking and unstacking of blocks. So we can say that with related to Robot Arm Blocks World is a two-dimensional problem.

We can represent the system in ILL. For that we first decide on the set of predicates to represent the state of the system.. The system can be represented in terms of five basic predicates. These predicates are of two types:

1. **Relationship Predicates Over Blocks:** These describe the relationship of the blocks to each other. These are:



# CHAPTER 4

## IMPLEMENTATION DETAILS

---

In our work we have used coq proof assistant which is based on the calculus of inductive constructions. There are some normal benefits of proof assistant such as uniformity of notations and verification of type correctness. There is some other enhancement of coq such as:

- Coq implements a higher-order constructive logic, facilitating in particular, the description of object logic within the framework.
- Coq has two type hierarchies: Set of constructive type and Prop of classical logic. This allows us to develop in classical logic and then verified in same framework against program written within Set.
- Coq support Inductive definitions ,giving a natural logic extension of definition by class style of programming such as in functional languages

### 4.1 Codify Intuitionistic Linear Logic (ILL)

In order to codify ILL in coq we first have to introduce a type of linear predicate, provide syntax for the linear connectives and define their right and left rules of the sequent calculus presentation of these connectives. Because Coq provides a single homogeneous system with a single built-in deduction. So our definition of ILL will has to exist as an ordinary data type within the system [11].

Linear logic is applied to a state based system along with the classical or Intuitionistic logic. State specific assertion can be phrased in ILL. So it is often useful to express global invariants in classical logic, so we can use them as often as possible.

For the setup of an ILL proof system involve two main steps:

- a) Defining a type of ILL predicate and their associated connectives. And
- b) Then defining a consequence operator and the associated sequent rules.

*Coq* < *LinCons*.

*LinCons*

: (list *ILinearProp*) -> *ILinearProp* -> *Prop*.

As for connectives we can define the infix operator “|-“ to denote this relation.

All the sequent rules [11] can be coded individually, using *coq* implication to represent deduction.

For example *impliesLeft* rule can be coded as:

Inductive *LinCons*: (list *ILinearProp*) -> *ILinearProp* -> *Prop* :=

| *ImpliesLeft*:

(A, B, C: *ILinearProp*)(D1,D2 : (list *ILinearProp*))

((D1 |- A) -> (D2 ^ B |- C) -> (D1 ^ D2 ^ (A -o B) |- C))

In this formula variable A, B, C, D1 and D2 are universally quantified. *list* concatenation is denoted by infix ‘^’ symbol and singleton list containing B is written as ‘B’.

General format of ILL rules in *coq* can be shown by some sequent based rules [13].

(\*here A, B: *ILinearProp*; G, D :( list *ILinearProp*); P: *Prop* \*)

*Axiom: exampl1: Empty |- A.*

*Axiom exampl2: G |- A.*

*Axiom exampl3 (G |- A) |- (D |- B).*

*Axiom example4 P -> (G |- A).*

An axiom in linear logic is represented as an axiom in *coq*, with an empty predicate-list (represented by constant *Empty*) to the left of the consequence relation. For example if we want to state that some predicate A is valid, we can simply assert that like *exampl1*.

The standard condition sequent is represented as a relation between a list of predicate and another predicate as in *exampl2*.

A deduction such as one of the sequent rule may be represented by one linear sequent that is conditional on another. So a rule of the form of

*exampl3 (G |- A) |- (D |- B).exampl3*

can be represented by *coq*’s implication “->” for the deduction as in *exampl3*.

The combination of linear and ordinary classical assumptions can be represented as in *exampl4*. Here the intuitionistic predicate P may share variable with other predicate and thus can assert state-invariant properties of any of the data they might refer to.

### 4.3.3 Special Cases

Now we establish special cases of each action as lemma in coq. That is they can be derived from the axioms get and put. For example it is more useful to have a version of get referring solely to the case where block x was on some other block y. this lemma can be written as:

*(\* Get a block x which is currently on some block y \*)*

*Lemma getXonY :*

*(x,y:Block)*

*( $\wedge$  (empty \*\* (clear x) \*\* (on x y))  $\mid$ - (holds x) \*\* (clear y)).*

There are some other cases too. These are:

If we want to pick up a block which is currently on the table. This special case can be written as lemma as

*(\* Get block x which is currently on the table \*)*

*Lemma getFromTable:*

*(x:Block)*

*( $\wedge$  (empty \*\* (clear x) \*\* (table x))  $\mid$ - (holds x)).*

Two special cases of put actions are:

Put a block x robot arm holding on the top of another block y.

*(\* Put block x (which the arm is holding) onto block y \*)*

*Lemma putXonY:*

*(x,y:Block)*

*( $\wedge$  ((holds x) \*\* (clear y))  $\mid$ - empty \*\* (clear x) \*\* (on x y)).*

Put a block x robot arm holding on the table.

*(\* Put block x (which the arm is holding) onto the table \*)*

*Lemma putXonTable:*

*(x:Block)*

*( $\wedge$  (holds x)  $\mid$ - empty \*\* (clear x) \*\* (table x)).*

All of these four special cases can be proved in coq script.

The statement and proof of lemma getoXnY in coq. This lemma established a special case of use of predicate get, where the block x that is being picked up is on the top of another block y.

This is the two blocks world scenario. Here our goal is to prove that there is a sequence of actions that will reverse the ordering of blocks a and b.

The precondition in this scenario is  $\text{empty} \otimes (\text{on } b \ a) \otimes (\text{ontable } a) \otimes (\text{clear } b) \otimes (\text{ontable } c) \otimes (\text{clear } c)$

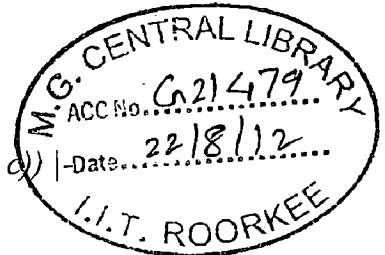
The post condition is  $\text{empty} \otimes (\text{on } a \ b) \otimes (\text{ontable } b) \otimes (\text{on } a \ b) \otimes (\text{clear } a) \otimes (\text{ontable } c) \otimes (\text{clear } c)$

In this case the goal can be achieved with applying these four actions. Here the proof is the list of states in which we can go through as the four actions are applied.

We can write this as a theorem in Coq as follow:

*Theorem SwapAB :*

$(\wedge (\text{empty} ** (\text{clear } b) ** (\text{on } b \ a) ** (\text{table } a) ** (\text{table } c) ** (\text{clear } c)) \rightarrow (\text{on } a \ b) ** \text{Top})$



Here we use T for the sink of unused predicates. Because we know that  $\Gamma \vdash T$  for any  $\Gamma$ .

The outline of the proof can be written as:

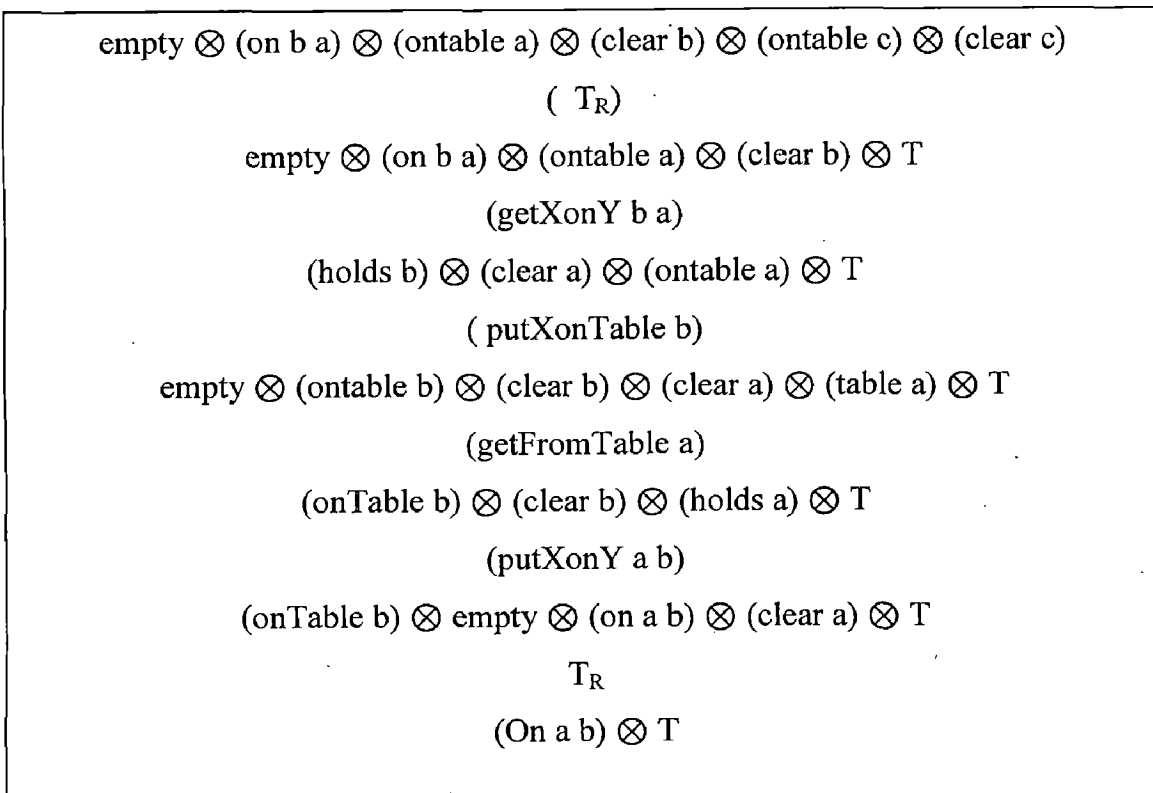


Figure 4.3 An Outline of SwapAB Proof.

# CHAPTER 5

## RESULTS AND DISCUSSION

---

In this work we have given an brief overview of the linear logic and its encoding in the Proof Assistant Coq. This encoding is basically based on the linear consequence operator as a two place predicate over linear logic terms. We have used a state based system i.e. Two Block world domain. We made its specification and verification of some of its properties in Coq. Specifying the system with predicate and on the base of these predicate, writing system properties and verifying is quite good in Coq. Coq is a feasible proof assistant for a linear logic, with its particule advantage are being able to integrate the existing Coq's data types and classical assumptions into the system.

Lets we want to prove a tautology  $((A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C))$ . For that we enter the Proof engine with the command Goal, followed by the conjecture we want to verify. Then we apply the tactic on the subgoals repeatedly until all the subgoals proved. The process is shown in figure 5.1. The proofs can be saved in file of type .v in Coq.

We have written the specification of the Block World in GALLINA and its properties about state. These are saved in file type of .v and successfully compiled. As shown in figure 5.2.

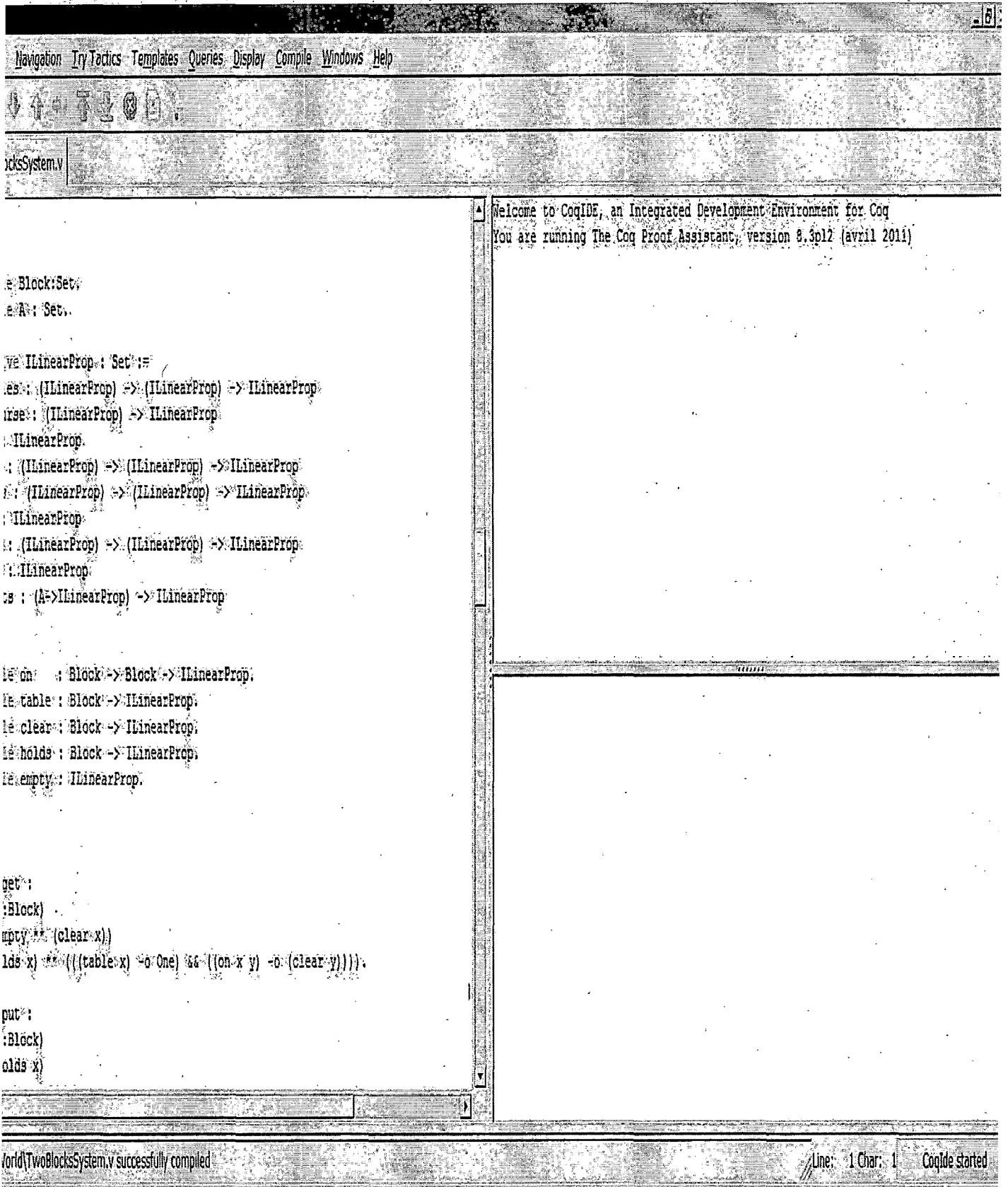


Figure 5.2 Two Block System Coq Specification

# REFERENCES

- [1]. Sutcliffe G. and Suttner C. "Evaluating General Purpose Automated Theorem Proving Systems". Department of Computer Science, University of Miami, P.O. Box 248154, Coral Gables, FL 33124, USA Sept. 2001
- [2]. Bundy A., Dennett D., Sharples M., Brady M, Partridge D. "Subsumption Architecture for Theorem Proving?," *Philosophical Transactions: Physical Sciences and Engineering*, Vol. 349, No. 1689. (Oct. 15, 1994), pp. 71-85
- [3]. Moore M.B., "Interactive Theorem Proving," M.S.Thesis, Dept.Comp.Eng. Monash University, Monash, Australia, November 2003, pp 20-22.
- [4]. Bertot Y. and Cast'eran P. "Interactive Theorem Proving and Program Development" *Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science.* ,Springer Verlag, 2004
- [5]. The Coq proof assistant. Documentation, System Downloads, Vers. 8.0, Coq Development Team,2011
- [6]. Bertot Y. and Cast'eran P."Coq's Art: Examples and Exercises".  
Internet: <http://www.labri.fr/Perso/~casteran/CoqArt>.
- [7]. Gimenez E. and Casteran P., "A Tutorial on Co-Inductive Types in Coq", May 1998 - August 17, 2007
- [8]. Ambrasky S., "Computation interpretations of linear logic," *Theoretical Computer Science*, 111:3-55,1997
- [9]. Dosen K. and Schroder P., "Substructural logic," *Studies in Logic and Computations*, Oxford University Press,1993
- [10]. Russel S.J and Norvig P. "Planning" in *Artificial Intelligence A Modern Approach* 2<sup>nd</sup> Ed. New Jersey: Pearson Education, 2003, pp 410-416
- [11]. Harland J. and Pym D. "A Note on Implementation and Application of Linear Logic Programming Language". In Gupta G., editors. *proceeding of the 17<sup>th</sup> Annual Computer Science Conference*, Christchurch, January 1994
- [12]. Bierman G.M., "On Intuitionistic Linear Logic," Ph.D. dissertation, Dept. Comp. Eng., Wolfson College, Cambridge. December 1993